Programmation en Java

Alexandre Meslé

12 octobre 2018

Table des matières

1 Notes de cours **2**
1.1 Introduction . . . 2
1.1.1 Hello World ! . . . 2
1.1.2 Formats de fichiers . . . 2
1.1.3 Machine virtuelle . . . 2
1.1.4 Linkage . . . 3
1.2 Variables . . . 4
1.2.1 Définition . . . 4
1.2.2 Déclaration . . . 4
1.2.3 Affectation . . . 4
1.2.4 Saisie . . . 5
1.2.5 Affichage . . . 5
1.2.6 Entiers . . . 5
1.2.7 Flottants . . . 6
1.2.8 Caractères . . . 7
1.2.9 Chaînes de caractères . . . 7
1.3 Opérateurs . . . 8
1.3.1 Généralités . . . 8
1.3.2 Les opérateurs unaires . . . 8
1.3.3 Les opérateurs binaires . . . 9
1.3.4 Formes contractées . . . 10
1.3.5 Opérations hétérogènes . . . 11
1.3.6 Les priorités . . . 12
1.4 Conditions . . . 13
1.4.1 Le bloc **if** . . . 13
1.4.2 Si ... Alors ... Sinon . . . 13
1.4.3 Switch . . . 15
1.4.4 Booléens . . . 16
1.4.5 Les priorités . . . 17
1.5 Boucles . . . 18
1.5.1 Définitions et terminologie . . . 18
1.5.2 while . . . 18
1.5.3 do ... while . . . 19
1.5.4 for . . . 19
1.5.5 Accolades superflues . . . 20
1.6 Survol du langage . . . 21
1.6.1 Structure d'une classe . . . 21
1.6.2 Variables . . . 21
1.6.3 Entrées-sorties . . . 21
1.6.4 Sous-programmes . . . 22
1.6.5 Main . . . 22
1.6.6 Instructions de contrôle de flux . . . 22
1.6.7 Exemple récapitulatif . . . 22

1.6.8 Packages . . . 23
1.7 Tableaux . . . 24
1.7.1 Déclaration . . . 24
1.7.2 Instanciation . . . 24
1.7.3 Accès aux éléments . . . 24
1.7.4 Longueur d'un tableau . . . 25
1.7.5 Tableaux à plusieurs dimensions . . . 25
1.8 Objets . . . 26
1.8.1 Création d'un type . . . 26
1.8.2 Les méthodes . . . 26
1.8.3 L'instanciation . . . 26
1.8.4 Les packages . . . 28
1.8.5 Le mot-clé **this** . . . 28
1.9 Encapsulation . . . 29
1.9.1 Exemple . . . 29
1.9.2 Visibilité . . . 31
1.9.3 Constructeur . . . 33
1.9.4 Accesseurs . . . 34
1.9.5 Surcharge . . . 35
1.9.6 Collections . . . 36
1.10 Héritage . . . 39
1.10.1 Héritage . . . 39
1.10.2 Polymorphisme . . . 40
1.10.3 Redéfinition de méthodes . . . 40
1.10.4 Interfaces . . . 40
1.10.5 Classes Abstraites . . . 41
1.11 Exceptions . . . 45
1.11.1 Rattraper une exception . . . 45
1.11.2 Méthodes levant des exceptions . . . 45
1.11.3 Propagation d'une exception . . . 46
1.11.4 Définir une exception . . . 46
1.11.5 Lever une exception . . . 47
1.11.6 Rattraper plusieurs exceptions . . . 47
1.11.7 Finally . . . 47
1.11.8 RuntimeException . . . 48
1.12 Interfaces graphiques . . . 49
1.12.1 Fenêtres . . . 49
1.12.2 Un premier objet graphique . . . 49
1.12.3 Ecouteurs d'événements . . . 50
1.12.4 Premier exemple . . . 50
1.12.5 Classes anonymes . . . 51
1.12.6 Gestionnaires de mise en forme . . . 52
1.12.7 Un exemple complet : Calcul d'un carré . . . 53
1.13 Tests unitaires . . . 55
1.13.1 Exemple . . . 55
1.13.2 Test à la bourrin . . . 56
1.13.3 Test des fonctions . . . 56
1.13.4 Test des fonctions automatisé . . . 57
1.13.5 Tests unitaires . . . 58
1.13.6 Logs . . . 59
1.14 Collections . . . 60
1.14.1 Types paramétrés . . . 60
1.14.2 Paramètres et héritage . . . 61
1.14.3 Collections standard . . . 62

1.15 Threads . . . 70
1.15.1 Le dîner des philosophes . . . 70
1.15.2 Lancement . . . 71
1.15.3 Synchronisation . . . 71
1.15.4 Mise en Attente . . . 73
1.16 Persistance . . . 75
1.16.1 Fichiers . . . 75
1.16.2 Serialization . . . 76
1.16.3 JDBC . . . 77
1.16.4 L'attaque par injection . . . 78
1.17 Hibernate . . . 81
1.17.1 Introduction . . . 81
1.17.2 Un premier exemple . . . 81
1.17.3 Un gestionnaire de contacts en quelques lignes . . . 84
1.17.4 Un exemple de relations entre les classes . . . 88

2 Exercices **99**
2.1 Variables . . . 99
2.1.1 Saisie et affichage . . . 99
2.1.2 Entiers . . . 100
2.1.3 Flottants . . . 100
2.1.4 Caractères . . . 100
2.2 Opérateurs . . . 101
2.2.1 Conversions . . . 101
2.2.2 Opérations sur les bits (difficiles) . . . 101
2.2.3 Morceaux choisis (difficiles) . . . 101
2.3 Conditions . . . 103
2.3.1 Prise en main . . . 103
2.3.2 Switch . . . 103
2.4 Boucles . . . 104
2.4.1 Compréhension . . . 104
2.4.2 Utilisation de toutes les boucles . . . 104
2.4.3 Choix de la boucle la plus appropriée . . . 105
2.4.4 Morceaux choisis . . . 105
2.4.5 Extension de la calculatrice . . . 106
2.5 Tableaux . . . 107
2.5.1 Exercices de compréhension . . . 107
2.5.2 Prise en main . . . 108
2.5.3 Indices . . . 108
2.5.4 Matrices . . . 108
2.6 Les sous-programmes . . . 110
2.6.1 Initiation . . . 110
2.6.2 Géométrie . . . 110
2.6.3 Arithmétique . . . 113
2.6.4 Tableaux . . . 114
2.6.5 Pour le sport . . . 114
2.7 Objets . . . 115
2.7.1 Création d'une classe . . . 115
2.7.2 Méthodes . . . 115
2.8 Encapsulation . . . 116
2.8.1 Prise en main . . . 116
2.8.2 Implémentation d'une pile . . . 116
2.8.3 Collections . . . 118
2.8.4 Refactoring de la pile avec des ArrayList . . . 118
2.8.5 Refactoring de la pile avec des listes chaînées . . . 119

2.9 Héritage . . . 121
2.9.1 Héritage . . . 121
2.9.2 Polymorphisme . . . 121
2.9.3 Interfaces . . . 124
2.9.4 Classes abstraites . . . 127
2.9.5 Un dernier casse-tête . . . 128
2.10 Exceptions . . . 135
2.11 Interfaces graphiques . . . 139
2.11.1 Prise en main . . . 139
2.11.2 Maintenant débrouillez-vous . . . 139
2.12 Tests unitaires . . . 140
2.12.1 Prise en main . . . 140
2.12.2 Pour aller plus loin . . . 141
2.13 Collections . . . 142
2.13.1 Types paramétrés . . . 142
2.13.2 Collections . . . 142
2.13.3 Morceaux choisis . . . 142
2.14 Threads . . . 145
2.14.1 Prise en main . . . 145
2.14.2 Synchronisation . . . 145
2.14.3 Débrouillez-vous . . . 145
2.15 Persistance . . . 146
2.15.1 Remember my name . . . 146
2.15.2 Hachage . . . 146
2.15.3 Morceaux choisis . . . 146
2.16 Hibernate . . . 147
2.16.1 Prise en main . . . 147
2.16.2 Contacts . . . 147

Chapitre 1

Notes de cours

1.1 Introduction

1.1.1 Hello World !

Copiez le code ci-dessous dans un fichier que vous enregistrerez sous le nom `HelloWorld.java`.

```
package introduction;

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

Ensuite, exécutez la commande

```
javac HelloWorld.java
```

L'exécution de cette commande doit normalement faire un apparaître un fichier `HelloWorld.class`. Saisissez ensuite la commande :

```
java HelloWorld
```

Théoriquement, ce programme devrait afficher

```
Hello World
```

1.1.2 Formats de fichiers

Les programmes java contiennent ce que l'on appelle des **classes**. Pour le moment nous ne mettrons qu'une seule classe par fichier et nous donnerons au fichier le même nom que la classe qu'il contient. Les fichiers sources portent l'extension `.java` tandis que les programmes compilés portent l'extension .class. Le compilateur que nous avons invoqué en ligne de commande avec l'instruction `javac`, a généré le fichier .class correspondant au fichier source passé en argument.

1.1.3 Machine virtuelle

Le fichier .class n'est pas un exécutable, il contient ce que l'on appelle du **pseudo-code**. Le pseudo-code n'est pas exécutable directement sur la plate-forme (systèle d'exploitation) pour laquelle il a été compilé, il est nécessaire de passer par un logiciel appelé une **machine virtuelle**. La machine virtuelle lit le pseudo-code et l'interprète (i.e. l'exécute). Le grand avantage de Java est que le pseudo code ne dépend pas de la plate-forme mais de la machine

virtuelle. Un programme compilé peut être exécuté par n'importe quel OS, la seule nécessité est de posséder une machine virtuelle Java (JVM) adaptée à cette plate-forme.

1.1.4 Linkage

En java, le linkage se fait à l'exécution, si vous modifiez un fichier, vous aurez une seule classe à recompiler, et vous pourrez immédiatement exécuter votre application.

1.2 Variables

1.2.1 Définition

Une variable est un emplacement de la mémoire dans lequel est stockée une valeur. Chaque variable porte une nom et c'est ce nom qui sert à identifier l'emplacement de la mémoire représenté par cette variable. Pour utiliser une variable, la première étape est la déclaration.

1.2.2 Déclaration

Déclarer une variable, c'est prévenir le compilateur qu'un nom va être utilisé pour désigner un emplacement de la mémoire. En Java, on déclare les variables à l'intérieur du bloc formé par les accolades du `main`. Il faut toujours déclarer les variables avant de s'en servir.

Nous ne travaillerons pour le moment que sur les variables de type numérique entier. Le type qui y correspond, en Java est `int`.

On déclare les variables entières de la manière suivante :

```
public static void main(string[] args)
{
        int variable1, variable2, ..., variablen;
        ...
}
```

Cette instruction déclare les variables `variable1, variable2, ..., variablen` de type entier. Par exemple,

```
int variable1, variable2;
int autrevariable1, autrevariable2;
```

1.2.3 Affectation

Si on souhaite affecter à la variable `v` une valeur, on utilise l'opérateur `=`. Par exemple,

```
int v;
v = 5;
```

Cet extrait de code déclare une variable de type entier que l'on appelle `v`, puis lui affecte la valeur 5. Comme vous venez de le voir, il est possible d'écrire directement dans le code une valeur que l'on donne à une variable.

Il est aussi possible d'initialiser une variable en même temps qu'on la déclare. Par exemple, l'extrait ci-dessus se reformule

```
int v = 5;
```

Les opérations arithmétiques disponibles sont l'addition (`+`), la soustraction (`-`), la multiplication (`*`), la division entière dans l'ensemble des entiers relatifs (quotient : `/`, reste : `%`). Par exemple,

```
int v, w, z;
v = 5;
w = v + 1;
z = v + w / 2;
v = z % 3;
v = v * 2;
```

1.2.4 Saisie

Traduisons en Java l'instruction `Saisir <variable>` que nous avons vu en algorithmique. Pour récupérer la saisie d'un utilisateur et la placer dans une variable `<variable>`, on utilise l'instruction suivante :

```
Scanner <s> = new Scanner(System.in);
<variable> = <s>.next<type>();
```

Il faut placer `import java.util.Scanner;` au début du fichier source pour que cette instruction fonctionne. et `<s>.close();` une fois les saisies terminées.

Ne vous laissez pas impressionner par l'apparente complexité de cette instruction, elle suspend l'exécution du programme jusqu'à ce que l'utilisateur ait saisi une valeur et pressé la touche `return`. La valeur saisie est alors affetée à la variable `<variable>`. Par exemple, pour déclarer une variable `i` et l'initialiser avec une saisie, on procède comme suit :

```
Scanner tutu = new Scanner(System.in);
int i = tutu.nextInt();
```

Il est possible par la suite de saisir d'autres variables sans avoir à redéclarer le `Scanner`.

```
float f = tutu.nextFloat();
```

1.2.5 Affichage

Traduisons maintenant l'instruction `Afficher <variable>` en Java :

```
System.out.println(<variable>);
```

Cette instruction affiche la valeur contenue dans la variable `variable`.

Nous avons étendu, en algorithmique, l'instruction `Afficher` en intercalant des valeurs de variables entre les messages affichés. Il est possible de faire de même en Java :

```
System.out.println("la valeur de la variable v est " + v + ".");
```

Les valeurs ou variables affichées sont ici séparés par des `+`. Tout ce qui est délimité par des double quotes est affiché tel quel. Cette syntaxe s'étend à volonté :

```
System.out.println("les valeurs des variables x, y et z sont "
                    + x + ", " + y + " et " + z);
```

Il est possible sous Eclipse, d'utiliser le raccourci `sysout` suivi de `Ctrl+space`.

1.2.6 Entiers

Quatre types servent à représenter les entiers :

nom	taille (t)	nombre de valeurs (2^{8t})
`byte`	1 octet	2^8 valeurs
`short`	2 octet	2^{16} valeurs
`int`	4 octets	2^{32} valeurs
`long`	8 octets	2^{64} valeurs

Plages de valeurs

Il nécessaire en programmation de représenter des valeurs avec des 0 et des 1, même si Java s'en charge pour vous, il est nécessaire de savoir comme il procède pour comprendre ce qu'il se passe en cas de problème. On retrouve donc dans la mémoire la représentation binaire des nombres entiers. Ainsi la plage de valeur d'un `byte`, encodée en binaire, est :

$$\{0000\ 0000, 0000\ 0001, 0000\ 0010, 0000\ 0011, \ldots, 1111\ 1110, 1111\ 1111\}$$

Les nombres entiers positifs sont ceux qui commencent par un 0, ils sont représentés sur l'intervalle :

$$\{0000\ 0000, 0000\ 0001, 0000\ 0010, 0000\ 0011, \dots, 0111\ 1100, 0111\ 1101, 0111\ 1110, 0111\ 1111\}$$

Les valeurs entières correspondantes sont :

$$\{0, 1, 2, \dots, 125, 126, 127\}$$

Et les nombres négatifs, commençant par un 1, sont donc représentés sur l'intervalle :

$$\{1000\ 0000, 1000\ 0001, 1000\ 0010, 1000\ 0011, \dots, 1111\ 1100, 1111\ 1101, 1111\ 1110, 1111\ 1111\}$$

Les nombres négatifs sont disposés du plus éloigné de 0 jusqu'au plus proche de 0, l'intervalle précédent code les valeurs :

$$\{-2^7, -(2^7-1), -(2^7-2), -(2^7-3), \dots, -4, -3, -2, -1\}$$

Par conséquent, on représente avec un `byte` les valeurs

$$\{-2^7, -(2^7-1), -(2^7-2), -(2^7-3), \dots, -4, -3, -2, -1, 0, 1, 2, \dots, 125, 126, 127\}$$

Les opérations arithmétiques sont exécutées assez bêtement, si vous calculez $0111\ 1111 + 0000\ 0001$, ce qui correspond à $(2^7 - 1) + 1$, le résultat mathématique est 2^7, ce qui se code $1000\ 0000$, ce qui est le codage de -2^7. Soyez donc attentifs, en cas de dépassement de capacité d'un nombre entier, vous vous retrouverez avec des nombres qui ne veulent rien dire. Si vous souhaitez faire des calculs sur des réels, un type flottant sera davantage adapté.

Le principe de représentation des entiers est le même sur tous les types entiers. On résume cela dans le tableau suivant :

nom	taille (t)	nombre de valeurs (2^{8t})	plus petite valeur	plus grande valeur
`byte`	1 octet	2^8 valeurs	-2^7	2^7-1
`short`	2 octet	2^{16} valeurs	-2^{15}	$2^{15}-1$
`int`	4 octets	2^{32} valeurs	-2^{31}	$2^{31}-1$
`long`	8 octets	2^{64} valeurs	-2^{63}	$2^{63}-1$

1.2.7 Flottants

Les flottants servent à représenter les réels. Leur nom vient du fait qu'on les représente de façon scientifique : un nombre décimal (à virgule) muni d'un exposant (un décalage de la virgule). Deux types de base servent à représenter les flottants :

nom	taille
`float`	4 octet
`double`	8 octets

Notez bien le fait qu'un point flottant étend le type à point fixe en permettant de "déplacer la virgule". Cela permet de représenter des valeurs très grandes ou très petites, mais au détriment de la précision. Nous examinerons dans les exercices les avantages et inconvénients des flottants.

Un littéral flottant s'écrit avec un point, par exemple l'approximation à 10^{-2} près de π s'écrit `3.14`. Il est aussi possible d'utiliser la notation scientifique, par exemple le décimal 1 000 s'écrit `1e3`, à savoir 1.10^3. Nous nous limiterons à la description du type `float`, le type `double` étant soumis à des règles similaires. Attention, les littéraux de type `float` s'écrivent avec un f en suffixe.

Représentation en mémoire d'un `float`

Le codage d'un nombre de type `float` (32 bits) est découpé en trois parties :

partie	taille
le bit de signe	1 bit
l'exposant	8 bits
la mantisse	23 bits

Le nombre est positif si le bit de signe est à 0, négatif si le bit de signe est à 1. La mantisse et l'exposant sont codés en binaire, la valeur absolue d'un flottant de mantisse m et d'exposant e est $\frac{m}{2^{23}}.2^e$. Le plus grand entier qu'il est possible de coder sur 23 octets est $(2^{23}-1)$, comme les bits de la mantisse représentent la partie décimale du nombre que l'on souhaite représenter, on obtient le plus grand nombre pouvant être représenté par la mantisse en divisant $(2^{23}-1)$ par 2^{23}, soit $\frac{(2^{23}-1)}{2^{23}}$. Comme le plus grand exposant est 2^7 , le plus grand flottant est $\frac{(2^{23}-1)}{2^{23}}.2^{(2^7)}$, donc le plus petit est $-\frac{(2^{23}-1)}{2^{23}}.2^{(2^7)}$.

1.2.8 Caractères

Un `char` sert à représenter le code `UNICODE` d'un caractère. Il est donc codé sur 2 octets. Il est possible d'affecter à une telle variable toute valeur du code `UNICODE` entourée de simples quotes. Par exemple, l'affectation suivante place dans `a` le code `UNICODE` du caractère `'B'`.

```
char a;
a = 'B';
```

Si une variable numérique entière `e` contient une valeur `UNICODE`, on obtient le caractère correspondant avec `(char)e`. Inversement, on obtient la valeur `UNICODE` d'une variable de type caractère `c` avec l'expression `(int)c` (où `int` peut être remplacé par n'importe quel type numérique entier d'au moins 2 octets).

1.2.9 Chaînes de caractères

Une chaîne de caractères, se déclarant avec le mot-clé `String`, est une succession de caractères (aucun, un ou plusieurs).

Les littéraux de ce type se délimitent par des double quotes, et l'instruction de saisie s'écrit sans le `next()` sans préciser le type. Par exemple,

```
String s = "toto";
String k = sc.next();
```

1.3 Opérateurs

1.3.1 Généralités

Opérandes et arité

Lorsque vous effectuez une opération, par exemple $3 + 4$, le $+$ est un opérateur, 3 et 4 sont des opérandes. Si l'opérateur s'applique à 2 opérandes, on dit qu'il s'agit d'un opérateur **binaire**, ou bien d'**arité** 2. Un opérateur d'arité 1, dit aussi **unaire**, s'applique à un seul opérande, par exemple $-$`x`, le `x` est l'opérande et le $-$ unaire est l'opérateur qui, appliqué à `x`, nous donne l'opposé de celui-ci, c'est-à-dire le nombre qu'il faut additionner à `x` pour obtenir 0. Il ne faut pas le confondre avec le $-$ binaire, qui appliqué à x et y, additionne à x l'opposé de y.

En Java, les opérateurs sont unaires ou binaires, et il existe un seul opérateur ternaire.

Associativité

Si vous écrivez une expression de la forme `a` $+$ `b` $+$ `c`, où `a`, `b` et `c` sont des variables entières, vous appliquez deux fois l'opérateur binaire $+$ pour calculer la somme de 3 nombres `a`, `b` et `c`. Dans quel ordre ces opérations sont-elles effectuées ? Est-ce que l'on a (`a` $+$ `b`) $+$ `c` ou `a` $+$ (`b` $+$ `c`) ? Cela importe peu, car le $+$ **entier** est **associatif**, ce qui signifie qu'il est possible de modifier le parenthèsage d'une somme d'entiers sans en changer le résultat. Attention : l'associativité est une rareté ! Peu d'opérateurs sont associatifs, une bonne connaissance des règles sur les priorités et le parenthèsage par défaut est donc requise.

Formes préfixes, postfixes, infixes

Un opérateur unaire est **préfixe** s'il se place avant son opérande, **postfixe** s'il se place après. Un opérateur binaire est **infixe** s'il se place entre ses deux opérandes (`a` $+$ `b`), **préfixe** s'il se place avant ($+$ `a b`), **postfixe** s'il se place après (`a b` $+$). Vous rencontrez en `C` des opérateurs unaires préfixes et d'autres postfixes (on imagine difficilement un opérateur unaire infixe), par contre tous les opérateurs binaires seront infixe.

Priorités

Les règles des priorités en Java sont nombreuses et complexes, nous ne ferons ici que les esquisser. Nous appelerons **parenthèsage implicite** le parenthèsage adopté par défaut par le Java, c'est à dire l'ordre dans lequel il effectue les opérations. La première règle à retenir est qu'un opérateur unaire est **toujours prioritaire** sur un opérateur binaire.

1.3.2 Les opérateurs unaires

Un opérateur unaire est toujours prioritaire sur un opérateur binaire ou ternaire.

Négation arithmétique

La négation arithmétique est l'opérateur $-$ qui à une opérande x, associe l'opposé de x, c'est-à-dire le nombre qu'il faut additionner à x pour obtenir 0.

Négation binaire

La négation binaire ~ agit directement sur les bits de son opérande, tous les bits à 0 deviennent 1, et vice-versa. Par exemple, ~127 (tous les bits à 1 sauf le premier) est égal à 128 (le premier bit à 1 et tous les autres à 0).

Priorités

Tous les opérateurs unaires sont de priorité équivalente, le parenthèsage implicite est fait le plus à droite possible, on dit que ces opérateurs sont **associatifs à droite**. Par exemple, le parenthèsage implicite de l'expression `~`$-$`~i` est `~`($-$(`~i`)). C'est plutôt logique : si vous parvenez à placer les parenthèses différemment, prevenez-moi parce que je ne vois pas comment faire...

1.3.3 Les opérateurs binaires

Les opérateurs binaires sont généralement associatifs à gauche. Et contrairement aux opérateurs unaires, ils ne sont pas tous de même priorité.

Opérations de décalages de bits

L'opération `a` $>>$ 1 effectue un décalage des bits de la représentation binaire de `a` vers la droite. Tous les bits sont décalés d'un cran vers la droite, le dernier bit disparaît, le premier prend la valeur 0. L'opération `a` $<<$ 1 effectue un décalage des bits de la représentation binaire de `a` vers la gauche. Tous les bits sont décalés d'un cran vers la gauche, le premier bit disparaît et le dernier devient 0. Par exemple, 32 $<<$ 2 associe à 0010 0000 $<<$ 2 la valeur dont la représentation binaire 1000 0000 et 32 $>>$ 3 associe à 0010 0000 $>>$ 3 la valeur dont la représentation binaire 0000 0100.

Opérations logiques sur la représentation binaire

L'opérateur & associe à deux opérandes le ET logique de leurs représentations binaires par exemple 60 & 15 donne 12, autrement formulé 0011 1100 `ET` 0000 1111 = 0000 1100. L'opérateur | associe à deux opérandes le OU logique de leurs représentations binaires par exemple 60 | 15 donne 63, autrement formulé 0011 1100 `OU` 0000 1111 = 0011 1111. L'opérateur ^ associe à deux opérandes le OU **exclusif** logique de leurs représentations binaires par exemple 60 ^ 15 donne 51, autrement formulé 0011 1100 `OU EXCLUSIF` 0000 1111 = 0011 0011. Deux ^ successifs s'annulent, en d'autres termes `a^b^b`=`a`.

Affectation

Ne vous en déplaise, le = est bien un opérateur binaire. Celui-ci affecte à l'opérande de gauche, qui doit être une variable, une valeur calculée à l'aide d'une expression, qui est l'opérande de droite. Attention, il est possible d'effectuer une affectation pendant l'évaluation d'une expression. Par exemple,

```
a = b + (c = 3);
```

Cette expression affecte à `c` la valeur 3, puis affecte à `a` la valeur `b` + `c`.

Priorités

Tous les opérateurs binaires ne sont pas de priorités équivalentes. Ceux de priorité la plus forte sont les opérateurs arithmétiques (∗, /, %, +, −), puis les opérateus de décalage de bit ($<<$, $>>$), les opérateurs de bit (&, ^, |), et enfin l'affectation =. Représentons dans une tableau les opérateurs en fonction de leur priorité, plaçons les plus prioritaire en haut et les moins prioritaires en bas. Parmi les opérateurs arithmétiques, les multiplications et divisions sont prioritaires sur les sommes et différence :

noms	opérateurs
produit	∗, /, %
sommes	+, −

Les deux opérateurs de décalage sont de priorité équivalente :

noms	opérateurs
décalage binaire	$>>$, $<<$

L'opérateur & est assimilé à un produit, | à une somme. Donc , & est prioritaire sur |. Comme ^ se trouve entre les deux, on a

noms	opérateurs
ET binaire	&
OU Exlusif binaire	^
OU binaire	|

Il ne nous reste plus qu'à assembler les tableaux :

noms	opérateurs
produit	*, /, %
somme	+, −
décalage binaire	>>, <<
ET binaire	&
OU Exlusif binaire	^
OU binaire	|
affectation	=

Quand deux opérateurs sont de même priorité le parenthèsage implicite est fait le plus à **gauche possible**, on dit que ces opérateurs sont **associatifs à gauche**. Par exemple, le parenthèsage implicite de l'expression `a - b - c` est

```
(a - b) - c
```

et **certainement pas** `a - (b - c)`. Ayez donc cela en tête lorsque vous manipulez des opérateurs non associatifs !

Attention : la seule exception est le =, qui est associatif à droite. Par exemple,

```
a = b = c;
```

se décompose en `b = c` suivi de `a = b`.

1.3.4 Formes contractées

Le Java étant un dérivé du C, qui est un langage de paresseux, tout à été fait pour que les programmeurs aient le moins de caractères possible à saisir. Je vous préviens : j'ai placé ce chapitre pour que soyez capable de décrypter la bouillie que pondent certains programmeurs, pas pour que vous les imitiez ! Alors vous allez me faire le plaisir de faire usage des formes contractées avec parcimonie, n'oubliez pas qu'il est très important que votre code soit **lisible**.

Unaires

Il est possible d'**incrémenter** (augmenter de 1) la valeur d'une variable `i` en écrivant `i++`, ou bien `++i`. De la même façon on peut **décrémenter** (diminuer de 1) `i` en écrivant `i--` (forme postfixe), ou bien `--i` (forme préfixe). Vous pouvez décider d'incrémenter (ou de décrémenter) la valeur d'une variable pendant un calcul, par exemple,

```
a = 1;
b = (a++) + a;
```

évalue successivement les deux opérandes `a++` et `a`, puis affecte leur somme à `b`. L'opérande `a++` est évaluée à 1, puis est incrémentée, donc lorsque la deuxième opérande `a` est évaluée, sa valeur est 2. Donc la valeur de b après l'incrémentation est 3. L'incrémentation contractée sous forme postfixe s'appelle une **post-incrémentation**. Si l'on écrit,

```
a = 1;
b = (++a) + a;
```

On opère une **pré-incrémentation**, `++a` donne lieu à une incrémentation avant l'évaluation de `a`, donc la valeur 4 est affectée à `b`. On peut de façon analogue effectuer une **pré-decrémentation** ou a **post-decrémentation**. Soyez très attentifs au fait que ce code n'est pas portable, il existe des compilateurs qui évaluent les opérandes dans le désordre ou diffèrent les incrémentations, donnant ainsi des résultats autres que les résultats théoriques exposés précédement. Vous n'utiliserez donc les **incrémentation** et **decrémentation** contractées que lorsque vous serez certain que l'ordre d'évaluation des opérandes ne pourra pas influer sur le résultat.

Binaires

Toutes les affectations de la forme `variable = variable operateurBinaire expression` peuvent être contractées sous la forme `variable operateurBinaire= expression`. Par exemple,

avant	après
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`
`a = a % b`	`a %= b`
`a = a >> i`	`a >>= i`
`a = a << i`	`a <<= i`
`a = a & b`	`a &= b`
`a = a ^ b`	`a ^= b`
`a = a | b`	`a |= b`

Vous vous douterez que l'égalité ne peut pas être contractée...

1.3.5 Opérations hétérogènes

Conversions implicites

Nous ordonnons de façon grossière les types de la façon suivante : **double** > **float** > **long** > **int** > **short** > **byte**. Dans un calcul où les opérandes sont de types hétérogènes, l'opérande dont le type T est de niveau le plus élévé (conformément à l'ordre énoncé ci-avant) est sélectionné et l'autre est converti dans le type T.

Le problème

Il se peut cependant que dans un calcul, cela ne convienne pas. Si par exemple, vous souhaitez calculer l'inverse $\frac{1}{x}$ d'un nombre entier x, et que vous codez

```
int i = 4;
System.out.println("L'inverse de " + i + " est " + 1/i);
```

Vous constaterez que résultat est inintéressant au possible. En effet, comme i et 1 sont tout deux de type entier, c'est la division entière est effectuée, et de toute évidence le résultat est 0. Ici la bidouille est simple, il suffit d'écrire le 1 avec un point :

```
int i = 4;
System.out.println("L'inverse de " + i + " est " + 1./i);
```

Le compilateur, voyant un opérande de type flottant, convertit lors du calcul l'autre opérande, `i`, en flottant. De ce fait, c'est une division flottante et non entière qui est effectuée. Allons plus loin : comment faire pour appliquer une division flottante à deux entiers ? Par exemple :

```
int i = 4, j= 5;
System.out.println("Le quotient de " + i + " et " + j + " est " + i/j + ".");
```

Cette fois-ci c'est inextricable, vous pouvez placer des points où vous voudrez, vous n'arriverez pas à vous débarrasser du warning et ce programme persistera à vous dire que ce quotient est -0.000000 ! Une solution particulièrement crade serait de recopier `i` et `j` dans des variables flottantes avant de faire la division, une autre méthode de bourrin est de calculer `(i + 0.)/j`. Mais j'espère que vous réalisez que seuls les boeufs procèdent de la sorte.

Le `cast`

Le **cast** est une conversion de type explicite dans une opération. Il se note en plaçant entre parenthèse le type vers lequel on veut convertir avant l'expression à convertir.

Le seul moyen de vous sortir de là est d'effectuer un `cast`, c'est à dire une conversion de type sur commande. On caste en plaçant entre parenthèse le type dans lequel on veut convertir juste avant l'opérande que l'on veut convertir. Par exemple,

```
int i = 4, j= 5;
System.out.println("Le quotient de " + i + " et " + j +
        " est " + (float)i/j + ".");
```

Et là, ça fonctionne. La variable valeur contenue dans `i` est convertie en **float** et de ce fait, l'autre opérande, `j`, est aussi convertie en **float**. La division est donc une division flottante. Notez bien que le `cast` est un opérateur unaire, donc prioritaire sur la division qui est un opérateur binaire, c'est pour ça que la conversion de `i` a lieu avant la division. Mais si jamais il vous vient l'idée saugrenue d'écrire

```
int i = 4, j= 5;
System.out.println("Le quotient de " + i + " et " + j +
        " est " + (float)(i/j) + ".");
```

Vous constaterez très rapidement que c'est une alternative peu intelligente. En effet, le résulat est flottant, mais comme la division a lieu avant toute conversion, c'est le résultat d'une division entière qui est converti en flottant, vous avez donc le même résultat que si vous n'aviez pas du tout casté.

1.3.6 Les priorités

Ajoutons le `cast` au tableau des priorités de nos opérateurs :

noms	opérateurs
opérateurs unaires	`cast`, −, ~, ++, −−
produit	∗, /, %
somme	+, −
décalage binaire	>>, <<
ET binaire	&
OU Exlusif binaire	^
OU binaire	|
affectation	=

1.4 Conditions

On appelle traitement conditionnel une portion de code qui n'est pas exécutée systématiquement.

1.4.1 Le bloc if

Principe

En pseudo-code un traitement conditionnel se rédige de la sorte :

> **Si** < *condition* > **alors**
> | < *instructions* >
> **Fin si**

Si la condition est vérifiée, alors les instructions sont exécutées, sinon, elles ne sont pas exécutées. L'exécution de l'algorithme se poursuit alors en ignorant les instructions se trouvant entre le `alors` et le `finSi`.

En java un traitement conditionnel se formule de la sorte :

```
if (<condition>)
{
        <instructions>
}
```

Notez bien qu'il n'y a pas de point-virgule après la parenthèse du **if**.

Opérateurs de comparaison

La formulation d'une condition se fait souvent à l'aide des opérateurs de comparaison. Les opérateurs de comparaison disponibles sont :

- == : égalité
- != : non-égalité
- <, <= : inférieur à, respectivement strict et large
- >, >= : supérieur à, respectivement strict et large

Par exemple, la condition `a` == `b` est vérifiée si et seulement si `a` et `b` ont la même valeur au moment où le test est évalué. Par exemple,

```
System.out.println("Saisissez une valeur");
int i = scanner.nextInt();
if (i == 0)
{
        System.out.println("Vous avez saisi une valeur nulle.");
}
System.out.println("Au revoir !");
```

Si au moment où le test `i` == 0 est évalué, la valeur de `i` est bien 0, alors le test sera vérifié et l'instruction `System.out.println("Vous avez saisi une valeur nulle.");` sera bien exécutée. Si le test n'est pas vérifié, les instructions du bloc sous la portée du **if** sont ignorées.

1.4.2 Si ... Alors ... Sinon

Il existe une forme étendue de traitement conditionnel, on la note en pseudo-code de la façon suivante :

> **Si** *condition* **alors**
> | instructions
> **Sinon**
> | autresinstructions
> **Fin si**

Les instructions délimitées par `alors` et `sinon` sont exécutées si le test est vérifié, et les instructions délimitées par `sinon` et `finSi` sont exécutées si le test n'est pas vérifié. On traduit le traitement conditionnel étendu de la sorte :

En java un traitement conditionnel étendu se formule de la sorte :

```
if (<condition>)
{
        <instructions1>;
}
else
{
        <instructions2>;
}
```

Par exemple,

```
System.out.println("Saisissez une valeur");
int i = scanner.nextInt();
if (i == 0)
{
        System.out.println("Vous avez saisi une valeur nulle.");
}
else
{
        System.out.println("La valeur que vous avez saisi, " + i +
                          ", n'est pas nulle.");
}
```

Notez la présence de l'opérateur de comparaison ==. **Si vous utilisez = pour comparer deux valeurs, ça ne compilera pas !**

Connecteurs logiques

On formule des conditions davantage élaborées en utilisant des connecteurs `et` et `ou`. La condition `A et B` est vérifiée si les deux conditions `A` et `B` sont vérifiées simultanément. La condition `A ou B` est vérifiée si au moins une des deux conditions `A` et `B` est vérifiée. Le `et` s'écrit && et le `ou` s'écrit ||. Par exemple, voici un programme C qui nous donne le signe de $i \times j$ sans les multiplier.

```
System.out.println("Saisissez deux valeurs numériques : ");
float i = scanner.nextFloat();
float j = scanner.nextFloat();
System.out.print("Le produit de " + i + " par " + j + " est ");
if ((i >= 0 && j >= 0) || (i < 0 && j < 0))
{
        System.out.println("positif.");
}
else
{
        System.out.println("négatif.");
}
```

Accolades superflues

Lorsqu'une seule instruction d'un bloc **if** doit être exécutée, les accolades ne sont plus nécessaires. Il est possible par exemple de reformuler le programme précédent de la sorte :

```
System.out.println("Saisissez deux valeurs numériques : ");
float i = scanner.nextFloat();
float j = scanner.nextFloat();
System.out.print("Le produit de " + i + " et " + j + " est ");
if ((i >= 0 && j >= 0) || (i < 0 && j < 0))
       System.out.println("positif.");
else
```

```
        System.out.println("négatif.");
```

Blocs

Un **bloc** est un ensemble d'instructions délimité par des accolades.

Par exemple le bloc **public static void** `Main(string[] args)`, ou encore le bloc **if**.

Vous aurez remarqué qu'il est possible d'imbriquer les blocs et qu'il convient d'ajouter un niveau d'indentation supplémentaire à chaque fois qu'un nouveau bloc est ouvert.

Une des conséquences de la structure de blocs du langage Java s'observe dans l'exemple suivant :

Achtung !

```
int i =  4;
if (i == 4)
{
        int j = i + 1;
}
System.out.println("j = " + j + ".");
```

Vous aurez remarqué la variable j est déclarée à l'intérieur du bloc **if**, et utilisé à l'extérieur de ce bloc. Ce code ne compilera pas parce que **une variable n'est utilisable qu'à l'intérieur du bloc où elle a été déclarée**. La **portée** (ou encore la **visibilité**) de la variable j se limite donc à ce bloc **if**.

Opérateur ternaire

En plaçant l'instruction suivante à droite d'une affectation,

```
<variable> = (<condition>) ? <valeur1> : <valeur2> ;
```

on place `valeur` dans `variable` si `condition` est vérifié, `autrevaleur` sinon. Par exemple,

```
max = (i>j) ? i : j ;
```

place la plus grande des deux valeurs `i` et `j` dans `max`. Plus généralement on peut utiliser le si ternaire dans n'importe quel calcul, par exemple

```
int i = 4;
int j = 2;
int k = 7;
int l;
System.out.println((i> (l = (j>k) ? j : k)) ? i : l);
```

`j = (j>k)? j : k)` place dans `l` la plus grande des valeurs `j` et `k`, donc `(i > (l = (j > k)? j : k)) ? i : l` est la plus grande des valeurs `i`, `j` et `k`. La plus grande de ces trois valeurs est donc affichée par cette instruction.

1.4.3 Switch

Le **switch** est une instruction permettant sélectionner un cas selon la valeur d'une variable. La syntaxe est la suivante :

```
switch(<nomvariable>)
{
  case <valeur_1> : <instructions_1> ; break ;
  case <valeur_2> : <instructions_2> ; break ;
  /* ... */
  case <valeur_n> : <instructions_n> ; break ;
  default :  <instructionspardefaut>  ; break;
}
```

Si *nomvariable* contient la valeur $valeur_i$, alors les $instructions_i$ sont exécutées. Si aucune des valeurs énumérées ne correspond à celle de *nomvariable*, ce sont les $instructionspardefaut$ qui sont exécutées. Les **break** servent à fermer chaque cas, y compris le dernier ! Si par exemple, nous voulons afficher le nom d'un mois en fonction de son numéro, on écrit :

```
switch(numeroMois)
{
  case 1 : System.out.print("janvier") ; break ;
  case 2 : System.out.print("fevrier") ; break ;
  case 3 : System.out.print("mars") ; break ;
  case 4 : System.out.print("avril") ; break ;
  case 5 : System.out.print("mai") ; break ;
  case 6 : System.out.print("juin") ; break ;
  case 7 : System.out.print("juillet") ; break ;
  case 8 : System.out.print("aout") ; break ;
  case 9 : System.out.print("septembre") ; break ;
  case 10 : System.out.print("octobre") ; break ;
  case 11 : System.out.print("novembre") ; break ;
  case 12 : System.out.print("decembre") ; break ;
  default : System.out.print("Je connais pas ce mois...");break;
}
```

1.4.4 Booléens

Une variable booléenne ne peut prendre que deux valeurs : *vrai* et $faux$.

Le type booléen en Java est **boolean** et une variable de ce type peut prendre soit la valeur **true**, soit la valeur **false**.

Utilisation dans des **if**

Lorsqu'une condition est évaluée, par exemple lors d'un test, cette condition prend à ce moment la valeur *vrai* si le test est vérifié, $faux$ dans le cas contraire. Il est donc possible de placer une variable booléenne dans un if. Observons le test suivant :

```
System.out.println("Saisissez un booléen : ");
boolean b = scanner.nextBoolean();
if (b)
        System.out.println("b is true.");
else
        System.out.println("b is false.");
```

Si *b* contient la valeur **true**, alors le test est réussi, sinon le **else** est exécuté. On retiendra donc qu'il est possible de placer dans un **if** toute expression pouvant prendre les valeurs **true** ou **false**.

Tests et affectations

Un test peut être effectué en dehors d'un **if**, par exemple de la façon suivante :

```
boolean x = (3>2);
```

Un test peut être effectué en dehors d'un **if**, par exemple de la façon suivante : On remarque que (3>2) est une condition. Pour décider quelle valeur doit être affectée à `x`, cette condition est évaluée. Comme dans l'exemple ci-dessus la condition est vérifiée, alors elle prend la valeur **true**, et cette valeur est affectée à `x`.

Connecteurs logiques binaires

Les connecteurs || et && peuvent s'appliquer à des valeurs (ou variables) booléennes. Observons l'exemple suivant :

```
boolean x = (true && false) || (true);
```

Il s'agit de l'affectation à x de l'évaluation de la condition (**true** && **false**)|| (**true**). Comme (**true** && **false**) a pour valeur **false**, la condition **false** || **true** est ensuite évaluée et prend la valeur **true**. Donc la valeur **true** est affectée à x.

Opérateur de négation

Parmi les connecteurs logiques se trouve !, dit opérateur de **négation**. La négation d'une expression est vraie si l'expression est fausse, fausse si l'expression est vraie. Par exemple,

```
boolean x = !(3==2);
```

Comme 3 == 2 est faux, alors sa négation !(3 == 2) est vraie. Donc la valeur **true** est affectée à x.

1.4.5 Les priorités

Complétons notre tableau des priorités en y adjoignant les connecteurs logiques et les opérateurs de comparaison :

noms	opérateurs
opérateurs unaires	cast, −, ~, !, ++, −−
produit somme	∗, /, % +, −
décalage binaire	>>, <<
comparaison	>, <, >=, <=
égalité	==, !=
ET binaire OU Exlusif binaire OU binaire	& ^ |
connecteurs logiques	&&, ||
if ternaire	() ?:
affectations	=, +=, −=, ...

1.5 Boucles

Nous souhaitons créer un programme qui nous affiche tous les nombres de 1 à 5, donc dont l'exécution serait la suivante :

```
1 2 3 4 5
```

Une façon particulièrement vilaine de procéder serait d'ecrire 5 `sysout` successifs, avec la laideur des copier/coller que cela impliquerait. Nous allons étudier un moyen de coder ce type de programme avec un peu plus d'élégance.

1.5.1 Définitions et terminologie

Une **boucle** permet d'exécuter plusieurs fois de suite une même séquence d'instructions.

Cette ensemble d'instructions s'appelle le **corps** de la boucle. Chaque exécution du corps d'une boucle s'appelle une **itération**, ou plus informellement un **passage** dans la boucle. Lorsque l'on s'apprête à exécuter la première itération, on dit que l'on **rentre** dans la boucle, lorsque la dernière itération est terminée, on dit qu'on **sort** de la boucle.

Il existe trois types de boucle :

— **while**
— **do** ... **while**
— **for**

Chacune de ces boucles a ses avantages et ses inconvénients. Nous les passerons en revue ultérieurement.

1.5.2 while

En java, la boucle `tant que` se code de la façon suivante :

```
while(<condition>)
{
  <instructions>
}
```

Les instructions du corps de la boucle sont délimitées par des accolades. La condition est évaluée **avant** chaque passage dans la boucle, à chaque fois qu'elle est vérifiée, on exécute les instructions de la boucle. Un fois que la condition n'est plus vérifiée, l'exécution se poursuit après l'accolade fermante.

Affichons par exemple tous les nombres de 1 à 5 dans l'ordre croissant,

```
        public static void main(String[] args)
        {
                int i = 1;
                while (i <= 5)
                {
                        System.out.print(i + " ");
                        i++;
                }
                System.out.println();
        }
```

Ce programme **initialise** i **à** 1 et tant que la valeur de i n'éxcède pas 5, cette valeur est affichée puis incrémentée. Les instructions se trouvant dans le corps de la boucle sont donc exécutées 5 fois de suite. La variable i s'appelle un **compteur**, on gère la boucle par incrémentations successives de i et on sort de la boucle une fois que i a atteint une certaine valeur. **L'initialisation du compteur est très importante** ! Si vous n'initialisez pas i explicitement, alors cette variable contiendra n'importe quelle valeur et votre programme ne se comportera pas du tout comme prévu. Notez bien qu'il n'y a **pas de point-virgule après le while** !

N'oubliez pas lorsqu'une boucle fonctionne avec un compteur :

— D'initialiser le compteur avant d'entrer dans la boucle
— D'incrémenter le compteur à la fin du corps
— De contrôler la valeur du compteur dans la conditon de boucle

1.5.3 do ... while

Voici la syntaxe de cette boucle :

```
do
{
  <instructions>
}
while(<condition>);
```

Le fonctionnement est analogue à celui de la boucle `tant que` à quelques détails près :
- la condition est évaluée **après** chaque passage dans la boucle.
- On exécute le corps de la boucle **tant que la condition est vérifiée**.
- On passe toujours **au moins une fois** dans une boucle `répéter... jusqu'à`

En Java, la boucle `répéter ... jusqu'à` est en fait une boucle `répéter ... tant que`, c'est-à-dire une boucle `tant que` dans laquelle la condition est évaluée **à la fin**. Une boucle **do** ... **while** est donc exécutée donc **au moins une fois**. Reprennons l'exemple précédent avec une boucle **do** ... **while** :

```
	public static void main(String[] args)
	{
		int i = 1;
		do
		{
			System.out.print(i + " ");
			i++;
		}
		while (i <= 5);
		System.out.println();
	}
```

De la même façon que pour la boucle **while**, le compteur est initialisé avant le premier passage dans la boucle. Un des usages les plus courant de la boucle **do** ... **while** est le contrôle de saisie :

```
	public static void main(String[] args)
	{
		Scanner saisie = new Scanner(System.in);
		int i;
		do
		{
			System.out.print("Saisissez un entier positif ou nul : ");
			i = saisie.nextInt();
			if (i < 0)
				System.out.println("J'ai dit positif ou nul !");
		}
		while (i < 0);
		saisie.close();
		System.out.println("Vous avez saisi " + i + ".");
	}
```

1.5.4 for

Cette boucle est quelque peu délicate. Commençons par donner sa syntaxe :

```
for(<initialisation> ; <condition> ; <pas>)
{
  <instructions>
}
```

L'`<initialisation>` est une instruction exécutée avant le premier passage dans la boucle. La `<condition>` est évaluée **avant** chaque passage dans la boucle, si elle n'est pas vérifiée, on ne passe pas dans la boucle et l'exécution

de la boucle pour est terminée. La <pas> est une instruction exécutée **après** chaque passage dans la boucle. On peut convertir une boucle **for** en boucle **while** en procédant de la sorte :

```
<initialisation>
while(<condition>)
{
  <instructions>
  <pas>
}
```

On re-écrit l'affiche des 5 premiers entiers de la sorte en utilisant le fait que `<initialisation>` = `i = 1`, `<condition>` = `i <= 5` et `<pas>` = `i++`. On obtient :

```
        public static void main(String[] args)
        {
                for (int i = 1; i <= 5 ;i++)
                        System.out.print(i + " ");
                System.out.println();
        }
```

On utilise une boucle **for** lorsque l'on connait en entrant dans la boucle combien d'itérations devront être faites. Par exemple, n'utilisez pas une boucle `pour` pour contrôler une saisie !

1.5.5 Accolades superflues

De la même façon qu'il est possible de supprimer des accolades autour d'une instruction d'un bloc **if**, on peut supprimer les accolades autour du corps d'une boucle si elle ne contient qu'une seule instruction.

1.6 Survol du langage

Ce cours vous introduit au Java dans sa dimension procédurale. Le but de ce cours est de vous familiariser avec les instructions de base avant d'aborder les concepts de la programmation objet.

1.6.1 Structure d'une classe

Pour le moment, nous appellerons **classe** un programme Java. Une classe se présente de la sorte :

```
class NomClasse
{
  /*
        Class contents
  */
}
```

Vous remarquez que de façon analogue au C, une classe est un bloc délimité par des accolades et que les commentaires d'écrivent de la même façon. On ecrit les noms des classes en concaténant les mots composant ce nom et en faisant commencer chacun d'eux par une majuscule. N'oubliez pas de ne mettre qu'une seule classe par fichier et de donner au fichier le même nom que celui de cette classe.

1.6.2 Variables

Nous disposons en Java des mêmes types qu'en C. Tous les types mis à votre disposition par Java sont appelé `types primitifs`.

Booléens

L'un deux nous servira toutefois à écrire des choses que l'on rédige habituellement de façon crade en C : **boolean**. Une variable de type **boolean** peut prendre une des deux valeurs **true** et **false**, et seulement une de ces deux valeurs. On déclare et utilise les variables exactement de la même façon qu'en C.

Chaînes de caractères

Il existe un type chaîne de caractères en C. Nous l'examinerons plus détails ultérieurement. Les deux choses à savoir est que le type chaîne de caractères est `String`, et que l'opérateur de concaténation est $+$.

final

Les variables dont la déclaration de type est précédée du mot-clé **final** sont non-modifiables. Une fois qu'une valeur leur a été affecté, il n'est plus possible de les modifier. On s'en sert pour représenter des constantes. Les règles typographiqus sont les mêmes : toutes les lettres en majuscules et les mots séparés par des _. Par exemple,

```
final int TAILLE = 100;
```

Déclare une constante `TAILLE` de type **int** initialisée à 100.

1.6.3 Entrées-sorties

La saisie de variables est un calvaire inutile en Java, nous nous en passerons pour le moment. Pour afficher un message, quel que soit son type, on utilise le sous-programme `System.out.print`. Par exemple,

```
System.out.print("Hello World\n");
```

Le sous-programme `System.out.println` ajoute automatiquement un retour à la ligne après l'affichage. On aurait donc pu écrire :

```
System.out.println("Hello World");
```

Il est aussi possible d'intercaler valeurs de variables et chaînes de caractères constantes, on sépare les arguments par des +. Par exemple,

```
System.out.println("La valeur de a est " + a +
                   "et celle de b est " + b);
```

1.6.4 Sous-programmes

On définit en Java des sous-programmes de la même façon qu'en C, attention toutefois, les sous-programmes que nous définirons dans ce cours seront déclarés avec le mot-clé **static**. Par exemple, si je veux faire un sous-programme qui retourne le successeur de la valeur passée en argument cela donne

```
class TestJavaProcedural
{

  static int succ(int i)
  {
    return i + 1;
  }

  /*
        Autres sous-programmes
  */
}
```

Si vous oubliez le mot clé **static**, le compilateur vous enverra des insultes. Nous verrons plus tard ce que signifie ce mot et dans quels cas il est possible de s'en passer. Attention : Tous les types primitifs en Java se passent en paramètre par valeur, et cette fois-ci je ne vous mens pas !

1.6.5 Main

Lorsque vous invoquez la machine virtuelle, elle part à la recherche d'un sous-programme particulier appelé `main`. Il est impératif qu'il soit défini comme suit :

```
public static void main(String[] args)
{
  /*
        instructions
  */
}
```

La signification du mot-clé **public** vous sera expliquée ultérieurement. La seule chose dont il faut prendre note est que vous ne devez pas l'oubliez, sinon la machine virtuelle vous enverra des insultes !

1.6.6 Instructions de contrôle de flux

En Java, les instructions **if**, **switch**, **for**, **while** et **do** ... **while** se comportent de la même façon qu'en C. Donc, pas d'explications complémentaires...

1.6.7 Exemple récapitulatif

Nous avons maintenant tout ce qu'il faut pour écrire un petit programme :

```
package procedural;

public class Exemple
{
        /*
         * Retourne le nombre b eleve a la puissance n.
```

```
         */
        static int puissance(int b, int n)
        {
                int res = 1;
                for (int i = 1; i <= n; i++)
                        res *= b;
                return res;
        }

        /*
         * Affiche {2^k | k = 0, ..., 30}.
         */
        public static void main(String[] args)
        {
                for (int k = 0; k <= 30; k++)
                        System.out.println("2^" + k + " = " + puissance(2, k));
        }
}
```

1.6.8 Packages

L'instruction **import** que vous utilisez depuis le début sert à localiser une classe parmi les packages standard de Java. Si par exemple vous importez **import** `java.awt.event.*`, cela signifie que vous allez utiliser les classes se trouvant dans le package qui se trouve dans le package `event` qui se trouve dans le package `awt` qui se trouve dans le package `java` (qui est la bibliothèque standard).

Créer ses propres packages

Il possible pour un programmeur de créer ses propres packages, deux choses sont nécessaires :
- Utilisez la même arborescence dans le système de fichier que celle des packages
- Ajouter au début de chaque source sa position dans l'arborescence.

Par exemple,

```
package procedural;

public class ExemplePackage
{
        public static void presenteToi()
        {
                System.out.println("Je suis collections.ExemplePackage");
        }

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

On exécute et compile ce fichier en se plaçant non pas dans le répertoire où se trouve les source, mais dans le répertoire ou se trouve le package. Par conséquent, l'exécution en ligne de commande se fait avec `javac procedural.ExemplePackage`

Utiliser ses propres packages

Vos packages s'utilisent comme les packages de la bibliothèque standard. Si le `main` se trouve en dehors de votre package, la variable d'environnement `CLASSPATH` sera utilisée pour localiser votre package.

1.7 Tableaux

1.7.1 Déclaration

Un tableau en Java est un objet. Il est nécessaire de le créer par allocation dynamique avec un **new** en précisant ses dimensions. On note

```
T[]
```

le type tableau d'éléments de type `T`. La taille du tableau n'est précisée qu'à l'instanciation. On déclare un tableau `t` d'éléments de type `T` de la façon suivante :

```
T[] t;
```

Par exemple, si l'on souhaite créer un tableau `i` d'éléments de type **int**, on utilise l'instruction :

```
int[] i;
```

1.7.2 Instanciation

Comme un tableau est un objet, il est nécessaire d'instancier pendant l'exécution. On instancie un tableau avec l'instruction

```
  new T[taille]
```

Par exemple, si l'on souhaite déclarer et allouer dynamiquement un tableau de 100 entiers, en utilise

```
int[] i =  new int[100];
```

1.7.3 Accès aux éléments

On accède aux éléments d'un tableau avec la notation à crochets, par exemple,

```
int[] t =  new int[100];
for(int i = 0 ; i < 100 ; i++)
        {
          t[i] = 100 - (i + 1);
          System.out.println(t[i]);
        }
```

On remarque que la variable `i` est déclarée à l'intérieur de la boucle **for**, cette façon de déclarer les variables est très utile en Java. Dans l'exemple,

```
package tableaux;

public class ExempleTableau
{
        public static void main(String[] args)
        {
                final int T = 20;
                int[] t = new int[T];
                for (int i = 0; i < T; i++)
                        t[i] = i;
                for (int i = 0; i < T; i++)
                        System.out.println(t[i]);
        }
}
```

Une tableau est alloué dynamiquement et ses dimensions sont de taille fixée à la compilation.

1.7.4 Longueur d'un tableau

Pour connaître la taille d'un tableau on utilise l'attribut `length`. Par exemple,

```
int[] t = new int[T];
for (int i = 0 ; i < t.length ; i++)
        t[i] = i;
for (int i = 0 ; i < t.length ; i++)
        System.out.println(t[i]);
```

1.7.5 Tableaux à plusieurs dimensions

On crée un tableau à plusieurs dimensions (3 par exemple) en juxtaposant plusieurs crochets. Par exemple,

```
int[][][] t =  new int[2][2][2];
for(int i = 0 ; i < 2 ; i++)
        for(int j = 0 ; i < 2 ; j++)
                for(int k = 0 ; i < 2 ; k++)
                {
                  t[i][j][k] = 100*i + 10*j + k;
                  System.out.println(t[i][j][k]);
                }
```

Ou encore,

```
package tableaux;

public class ExempleCubique
{
        public static void main(String[] args)
        {
                final int T = 3;
                int[][][] u = new int[T][T][T];
                for (int i = 0; i < T; i++)
                        for (int j = 0; j < T; j++)
                                for (int k = 0; k < T; k++)
                                {
                                        u[i][j][k] = 100 * i + 10 * j + k;
                                        System.out.println(u[i][j][k]);
                                }

        }
}
```

1.8 Objets

Dans un langage de programmation, un **type** est
— Un ensemble de **valeurs**
— Des **opérations** sur ces valeurs

En plus des types primitifs, il est possible en Java de créer ses propres types. On appelle type construit un type non primitif, c'est-à-dire composé de types primitifs. Certains types construits sont fournis dans les bibliothèques du langage. Si ceux-là ne vosu satisfont pas, vous avez la possibilité de créer vos propres types.

1.8.1 Création d'un type

Nous souhaitons créer un type `Point` dans R^2. Chaque variable de ce type aura deux attributs, une abscisse et une ordonnée. Le type point se compose donc à partir de deux types flottants. Un type construit s'appelle une **classe**. On le définit comme suit :

```
public class Point
{
        float abscisse;
        float ordonnee;
}
```

Les deux attributs d'un objet de type `Point` s'appelle aussi des **champs**. Une fois définie cette classe, le type `Point` est une type comme les autres, il devient donc possible d'écrire

```
Point p, q;
```

Cette instruction déclare deux variables `p` et `q` de type `Point`, ces variables s'appellent des **objets**. Chacun de ces objets a deux attributs auxquels on accède avec la notation pointée. Par exemple, l'abscisse du point `p` est `p.abscisse` et son ordonnée est `p.ordonnee`. Le fonctionnement est, pour le moment, le même que pour les structures en C.

1.8.2 Les méthodes

Non contents d'avoir défini ainsi un ensemble de valeurs, nous souhaiterions définir un ensemble d'opérations sur ces valeurs. Nous allons pour ce faire nous servir de **méthodes**. Une méthode est un sous-programme propre à chaque objet. C'est-à-dire dont le contexte d'exécution est délimité par un objet. Par exemple,

```
public class Point
{
    float abscisse;
    float ordonnee;

    void presenteToi()
    {
        System.out.println("Je suis un point, mes coordonnées sont ("
                           + abscisse + ", " + ordonnee + ")");
    }
}
```

La méthode `presenteToi` s'invoque à partir d'un objet de type `Point`. La syntaxe est `p.presenteToi()` où `p` est de type `Point`. `p` est alors le contexte de l'exécution de `presenteToi` et les champs auquel accèdera cette méthode seront ceux de l'objet `p`. Si par exemple, on ecrit `q.presenteToi()`, c'est `q` qui servira de contexte à l'exécution de `presenteToi`. Lorsque l'on rédige une méthode, l'objet servant de contexte à l'exécution de la méthode est appelé l'**objet courant**.

1.8.3 L'instanciation

Essayons maintenant, impatients et nerveux, de vérifier ce que donnerait l'exécution de

```
public class Point
{
```

```java
    float abscisse;
    float ordonnee;

    void presenteToi()
    {
        System.out.println("Je suis un point, mes coordonnées sont ("
                           + abscisse + ", " + ordonnee + ")");
    }

    public static void main(String[] args)
    {
        Point p;
        p.abscisse = 1;
        p.ordonnee = 2;
        p.presenteToi();
    }
}
```

Pas de chance, ce programme ne compile pas. Et davantage pointilleux que le C, le compilateur de Java s'arrête au moindre petit souci... Le message d'erreur à la compilation est le suivant :

```
Point.java:15: variable p might not have been initialized
        p.abscisse = 1;
        ^
1 error
```

Que peut bien signifier ce charabia ? Cela signifie que le compilateur a de très sérieuses raisons de penser que `p` contient pas d'objet. En Java, toutes les variables de type construit sont des **pointeurs**. Et les pointeurs non initialisés (avec `malloc` en C) sont des pointeurs ayant la valeur **null**. Pour faire une allocation dynamique en Java, on utilise l'instruction **new**. On fait alors ce que l'on appelle une **instanciation**. La syntaxe est donnée dans l'exemple suivant :

```java
p = new Point();
```

Cette instruction crée un objet de type `Point` et place son adresse dans `p`. Ainsi `p` n'est pas un objet, mais juste un pointeur vers un objet de type `Point` qui a été crée par le **new**. Et à partir de ce moment-là, il devient possible d'accéder aux champs de `p`. **new** fonctionne donc de façon similaire à `malloc` et le programme suivant est valide :

```java
package classes;

public class Point
{
        float abscisse;
        float ordonnee;

        void presenteToi()
        {
                System.out.println("Je suis un point, mes coordonnees sont ("
                                + abscisse + ", " + ordonnee + ")");
        }

        public static void main(String[] args)
        {
                Point p;
                p = new Point();
                p.abscisse = 1;
                p.ordonnee = 2;
                p.presenteToi();
        }
}
```

Vous remarquez qu'il n'y a pas de fonction de destruction (`free()` en C). Un programme appelé **Garbage Collector** (ramasse-miette en français) est exécuté dans la machine virtuelle et se charge d'éliminer les objets non référencés.

1.8.4 Les packages

Bon nombre de classes sont prédéfinies en Java, elles sont réparties dans des packages, pour utilisez une classe se trouvant dans un package, on l'importe au début du fichier source. Pour importer un package, on utilise l'instruction **import**.

1.8.5 Le mot-clé this

Dans toute méthode, vous disposez d'une référence vers l'objets servant de contexte à l'exécution de la méthode, cette référence s'appelle **this**.

1.9 Encapsulation

1.9.1 Exemple

Implémentons une file de nombre entiers. Nous avons un exemple dans le fichier suivant :

```
package encapsulation;

public class FilePublic
{
	/*
	 * Elements de la file
	 */

	int[] entiers;

	/*
	 * Indice de la tête de file et du premier emplacement libre dans le
	 * tableau.
	 */

	int first, firstFree;

	/*
	 * Initialise les attributs de la file.
	 */

	public void init(int taille)
	{
		entiers = new int[taille + 1];
		first = firstFree = 0;
	}

	/*
	 * Décale i d'une position vers la droite dans le tableau, revient au debut
	 * si i déborde.
	 */

	public int incrementeIndice(int i)
	{
		i++;
		if (i == entiers.length)
			i = 0;
		return i;
	}

	/*
	 * Retourne vrai si et seulement si la file est pleine.
	 */

	public boolean estPlein()
	{
		return first == incrementeIndice(firstFree);
	}

	/*
	 * Retourne vrai si et seulement si la file est vide.
	 */

	public boolean estVide()
```

```
	{
		return first == firstFree;
	}

	/*
	 * Ajoute l'élément n dans la file.
	 */

	public void enfile(int n)
	{
		if (!estPlein())
		{
			entiers[firstFree] = n;
			firstFree = incrementeIndice(firstFree);
		}
	}

	/*
	 * Supprime la tête de file.
	 */

	public void defile()
	{
		if (!estVide())
			first = incrementeIndice(first);
	}

	/*
	 * Retourne la tête de file.
	 */

	public int premier()
	{
		if (!estVide())
			return entiers[first];
		return 0;
	}
}
```

Un exemple typique d'utilisation de cette file est donné ci-dessous :

```
package encapsulation;

public class TestFilePublic
{
	public static void main(String[] args)
	{
		FilePublic f = new FilePublic();
		f.init(20);
		for (int i = 0; i < 30; i += 2)
		{
			f.enfile(i);
			f.enfile(i + 1);
			System.out.println(f.premier());
			f.defile();
		}
		while (!f.estVide())
		{
			System.out.println(f.premier());
			f.defile();
```

```
                }
        }
}
```

Si vous travaillez en équipe, et que vous êtes l'auteur d'une classe `FilePublic`, vous n'aimeriez pas que vos collègues programmeurs s'en servent n'importe comment ! Vous n'aimeriez pas par exemple qu'ils manipulent le tableau `entiers` ou l'attribut `first` sans passer par les méthodes, par exemple :

```
        FilePublic f = new FilePublic();
        f.init(20);
        f.entiers[4] = 3;
        f.first += 5;
        /* ... arretons la les horreurs ... */
```

Il serait appréciable que nous puissions interdire de telles instructions. C'est-à-dire forcer l'utilisateur de la classe à passer par les méthodes pour manier la file.

1.9.2 Visibilité

La **visibilité** d'un identificateur (attribut, méthode ou classe) est l'ensemble des endroits dans le code ou il est possible de l'utiliser. Si un identificateur et précédé du mot clé **public**, cela signifie qu'il est visible partout. Si un identificateur et précédé du mot clé **private**, cela signifie qu'il n'est visible qu'à l'intérieur de la classe. Seule l'instance à qui cet identificateur appartient pourra l'utiliser. Par exemple :

```
package encapsulation;

public class FilePrivate
{
        private int[] entiers;
        private int first, firstFree;

        public void init(int taille)
        {
                entiers = new int[taille + 1];
                first = firstFree = 0;
        }

        private int incrementeIndice(int i)
        {
                i++;
                if (i == entiers.length)
                        i = 0;
                return i;
        }

        public boolean estPlein()
        {
                return first == firstFree + 1;
        }

        public boolean estVide()
        {
                return first == incrementeIndice(firstFree);
        }

        public void enfile(int n)
        {
                if (!estPlein())
                {
                        entiers[firstFree] = n;
```

```
                    firstFree = incrementeIndice(firstFree);
                }
        }

        public void defile()
        {
                if (!estVide())
                        first = incrementeIndice(first);
        }

        public int premier()
        {
                if (!estVide())
                        return entiers[first];
                return 0;
        }
}
```

On teste cette classe de la même façon :

```
package encapsulation;

public class TestFilePrivate
{
        public static void main(String[] args)
        {
                FilePrivate f = new FilePrivate();
                f.init(20);
                for (int i = 0; i < 30; i += 2)
                {
                        f.enfile(i);
                        f.enfile(i + 1);
                        System.out.println(f.premier());
                        f.defile();
                }
                while (!f.estVide())
                {
                        System.out.println(f.premier());
                        f.defile();
                }
        }
}
```

S'il vous vient l'idée saugrenue d'exécuter les instructions :

```
        FilePrivate f = new FilePrivate();
        f.init(20);
        f.entiers[4] = 3;
        f.first += 5;
        /* ... arretons la les horreurs ... */
```

vous ne passerez pas la compilation. Comme les champs `entiers` et `first` sont **private**, il est impossible de les utiliser avec la notation pointée. Cela signifie qu'ils ne sont accessibles que depuis l'instance de la classe `FilePrivate` qui sert de contexte à leur exécution. De cette façon vous êtes certain que votre classe fonctionnera correctement. En déclarant des champs privés, vous avez caché les divers détails de l'implémentation, cela s'appelle l'**encapsulation**. Celle-ci a pour but de faciliter le travail de tout programmeur qui utilisera cette classe en masquant la complexité de votre code. Les informations à communiquer à l'utilisateur de la classe sont la liste des méthodes publiques. A savoir

```
public class FilePrivate
{
    public void init(int taille){/*...*/}
```

```
    public boolean estPlein(){/*...*/}
    public boolean estVide(){/*...*/}
    public void enfile(int n){/*...*/}
    public void defile(){/*...*/}
    public int premier(){/*...*/}
}
```

On remarque non seulement qu'il plus aisé de comprendre comment utiliser la file en regardant ces quelques méthodes mais surtout que la façon dont a été implémenté la file est totalement masquée.

1.9.3 Constructeur

Supposons que notre utilisateur oublie d'invoquer la méthode `init(int taille)`, que va-t-il se passer ? Votre classe va planter. Comment faire pour être certain que toutes les variables seront initialisées ? Un **constructeur** est un sous-programme appelé automatiquement à la création de tout objet. Il porte le même nom que la classe et n'a pas de valeur de retour. De plus, il est possible de lui passer des paramètres au moment de l'instanciation. Remplaçons `init` par un constructeur :

```
package encapsulation;

public class FileConstructeur
{
        private int[] entiers;
        private int first, firstFree;

        public FileConstructeur(int taille)
        {
                entiers = new int[taille + 1];
                first = firstFree = 0;
        }

        private int incrementeIndice(int i)
        {
                i++;
                if (i == entiers.length)
                        i = 0;
                return i;
        }

        public boolean estPlein()
        {
                return first == firstFree + 1;
        }

        public boolean estVide()
        {
                return first == incrementeIndice(firstFree);
        }

        public void enfile(int n)
        {
                if (!estPlein())
                {
                        entiers[firstFree] = n;
                        firstFree = incrementeIndice(firstFree);
                }
        }

        public void defile()
```

```
	{
		if (!estVide())
			first = incrementeIndice(first);
	}

	public int premier()
	{
		if (!estVide())
			return entiers[first];
		return 0;
	}
}
```

On peut alors l'utiliser sans la méthode `init`,

```
package encapsulation;

public class TestFileConstructeur
{
	public static void main(String[] args)
	{
		FileConstructeur f = new FileConstructeur(20);
		for (int i = 0; i < 30; i += 2)
		{
			f.enfile(i);
			f.enfile(i + 1);
			System.out.println(f.premier());
			f.defile();
		}
		while (!f.estVide())
		{
			System.out.println(f.premier());
			f.defile();
		}
	}
}
```

1.9.4 Accesseurs

Il est de bonne programmation de déclarer tous les attributs en privé, et de permettre leur accès en forçant l'utilisateur à passer par des méthodes. Si un attribut X est privé, on crée deux méthodes $getX$ et $setX$ permettant de manier X. Par exemple,

```
package encapsulation;

public class ExempleAccesseurs
{
	private int foo;

	public int getFoo()
	{
		return foo;
	}

	public void setFoo(int value)
	{
		foo = value;
	}
}
```

On ainsi la possibilité de manier des attributs privés indirectement.

1.9.5 Surcharge

Il est possible de definir dans une même classe plusieurs fonctions portant le même nom. Par exemple,

```java
package encapsulation;

public class FileSurcharge
{
	private int[] entiers;
	private int first, firstFree;

	public FileSurcharge(int taille)
	{
		entiers = new int[taille + 1];
		first = firstFree = 0;
	}

	public FileSurcharge(FileSurcharge other)
	{
		this(other.entiers.length - 1);
		for (int i = 0; i < other.entiers.length; i++)
			entiers[i] = other.entiers[i];
		first = other.first;
		firstFree = other.firstFree;
	}

	private int incrementeIndice(int i)
	{
		i++;
		if (i == entiers.length)
			i = 0;
		return i;
	}

	public boolean estPlein()
	{
		return first == firstFree + 1;
	}

	public boolean estVide()
	{
		return first == incrementeIndice(firstFree);
	}

	public void enfile(int n)
	{
		if (!estPlein())
		{
			entiers[firstFree] = n;
			firstFree = incrementeIndice(firstFree);
		}
	}

	public void defile()
	{
		if (!estVide())
			first = incrementeIndice(first);
	}
```

```
        public int premier()
        {
                if (!estVide())
                        return entiers[first];
                return 0;
        }
}
```

On remarque qu'il y a deux constructeurs. L'un prend la taille du tableau en paramètre et l'autre est un constructeur de copie. Selon le type de paramètre passé au moment de l'instanciation, le constructeur correspondant est exécuté. Dans l'exemple ci-dessous, on crée une file g en applicant le constructeur de copie à la file f.

```
package encapsulation;

public class TestFileSurcharge
{
        public static void main(String[] args)
        {
                FileSurcharge f = new FileSurcharge(20);
                for (int i = 0; i < 30; i += 2)
                {
                        f.enfile(i);
                        f.enfile(i + 1);
                        System.out.println(f.premier());
                        f.defile();
                }
                FileSurcharge g = new FileSurcharge(f);
                while (!f.estVide())
                {
                        System.out.println(f.premier());
                        f.defile();
                }
                while (!g.estVide())
                {
                        System.out.println(g.premier());
                        g.defile();
                }
        }
}
```

Visibilité *package*

En l'absence de mots-clés **public** (tout le monde), **private** (seulement la classe) et **protected** (classe + classes dérivées), une visibilité par défaut est appliquée. Par défaut, la visibilité des éléments est limitée au package.

1.9.6 Collections

Exemple

Considérons l'exemple suivant :

```
package encapsulation;

public class NativeArray
{
        public static void main(String[] args)
        {
                int taille = (47 - 2) / 3 + 1;
```

```
                int[] a = new int[taille];
                for (int value = 2, index = 0; value < 50; value += 3, index++)
                        a[index] = value;
                for (int index = 0; index < taille; index++)
                        System.out.println(a[index]);
        }

}
```

Dans ce programme, les valeurs entières $2, 5, 8, \ldots, 47$ sont placées dans un tableau puis affichées. Vous remarquerez que la principale difficulté est liée à l'utilisation des indices.

Il existe des classes en Java permettant de s'affranchir de ce type d'inconvénient en prenant en charge les problèmes d'indices. Par exemple :

```
package encapsulation;

import java.util.ArrayList;

public class CollectionArray
{
        public static void main(String[] args)
        {
                ArrayList a = new ArrayList();
                for (int value = 2; value < 50; value += 3)
                        a.add(value);
                for (int index = 0; index < a.size(); index++)
                        System.out.println(a.get(index));
        }
}
```

La classe `ArrayList` est un tableau dont la taille s'adapte au nombre de données contenues. Les méthodes suivantes ont été utilisées dans l'exemple ci-avant :

— **public boolean** `add`(... `o`), ajoute l'objet `o` à la fin du tableau.
— **public** ... `get`(**int** `index`) permet de récupérer l'objet stocké dans le tableau à l'indice `index`.
— **public int** `size`() retourne la taille actuelle du tableau, c'est à dire le nombre d'éléments qu'il contient.

Types paramétrés

L'utilisation des collections se paye avec des instructions peu élégantes :

```
package encapsulation;

import java.util.ArrayList;

public class Cast
{
        public static void main(String[] args)
        {
                ArrayList a = new ArrayList();
                a.add("toto");
                String s = (String) a.get(0);
                System.out.println(s);
                a.add(5);
                int i = (int) a.get(1);
                System.out.println(i);
        }
}
```

Il est nécessaire d'effectuer un cast particulièrement laid pour passer la compilation. Les types paramétrés fournissent un moyen élégant de résoudre ce problème.

Utilisation

Lorsque l'on déclare une référence vers un type paramétré, on doit préciser au moment de la déclaration le type que l'on souhaite utiliser. Si par exemple, on souhaite créer un `ArrayList` ne pouvant contenir que des objets de type `String`, la référence devra être du type `ArrayList<String>`.

```
package encapsulation;

import java.util.ArrayList;

public class ArrayListParametre
{
        public static void main(String[] args)
        {
                ArrayList<Integer> a = new ArrayList<Integer>();
                for (int value = 2; value < 50; value += 3)
                        a.add(value);
                for (int index = 0; index < a.size(); index++)
                        System.out.println(a.get(index));
        }
}
```

Il est possible lors d'une instanciation, d'omettre les le type, java prendra automatiquement le type correct.

Parcours

Il existe, de façon analogue à que l'on observe en C++, une syntaxe permettant simplement de parcourir une collection qui est :

```
for (T x : c)
{
  /* ... */
}
```

Dans la spécification ci dessus, `c` est la collection, `T` le type (ou un sur-type) des objets de la collection et `x` une variable (muette) dans laquelle sera placé tour à tour chaque objet de la collection.

```
package encapsulation;

import java.util.ArrayList;

public class ArrayListForEach
{
        public static void main(String[] args)
        {
                ArrayList<Integer> a = new ArrayList<>();
                for (int value = 2; value < 50; value += 3)
                        a.add(value);
                for (int value : a)
                        System.out.println(value);
        }
}
```

1.10 Héritage

1.10.1 Héritage

Le principe

Quand on dispose d'une classe c, on a la possibilité de l'agrandir (ou de l'étendre) en créant une deuxième classe c'. On dit dans ce cas que c' **hérite de** c, ou encore que c est la **classe mère** et c' la **classe fille**. Par exemple, considérons les deux classes suivantes :

```
package heritage;

public class ClasseMere
{
        private final int x;

        public ClasseMere(int x)
        {
                this.x = x;
        }

        public int getX()
        {
                return x;
        }
}
```

```
package heritage;

public class ClasseFille extends ClasseMere
{
        private final int y;

        public ClasseFille(int x, int y)
        {
                super(x);
                this.y = y;
        }

        public int getY()
        {
                return y;
        }
}
```

Le mot-clé extends signifie **hérite de**. Le mot-clé super est le constructeur de la classe mère. Tout ce qui peut être fait avec la classe mère peut aussi être fait avec la classe fille. Notez donc que la classe fille est une **extension** de la classe mère. On peut par exemple utiliser la classe fille de la façon suivante :

```
package heritage;

public class TestClasseFille
{
        public static void main(String[] args)
        {
                ClasseFille o = new ClasseFille(1, 2);
                System.out.println("(" + o.getX() + ", " + o.getY() + ")");
        }
}
```

Notes bien que comme la classe mère possède une méthode `getX`, la classe fille la possède aussi. Et l'attribut `x` de tout instance de la classe mère est aussi un attribut de toute instance de la classe fille.

Héritage simple Vs. héritage multiple

En java, une classe ne peut avoir qu'une seule classe mère. Dans d'autres langages (comme le `C`++) cela est permis, mais pour des raisons de fiabilité, Java l'interdit.

`Object`

En java, toutes les classes héritent implicitement d'une classe `Object`.

1.10.2 Polymorphisme

Considérons l'exemple suivant :

```
package heritage;

public class TestClasseFillePolymorphisme
{
        public static void main(String[] args)
        {
                ClasseMere o = new ClasseFille(1, 2);
                System.out.println("(" + o.getX() + ", "
                                + ((ClasseFille) o).getY()
                                + ")");
        }
}
```

On remarque d'une part que *o* référence un objet de la classe fille de son type. Cela s'appelle le **polymorphisme**. D'autre part, si l'on souhaite effectuer des opérations spécifiques aux objets de la classe fille, il est nécessaire d'effectuer un cast.

1.10.3 Redéfinition de méthodes

La méthode `toString`() appartient initialement à la classe `Object`. Elle est donc héritée par toutes les classes. Vous avez la possibilité de la redéfinir, c'est à dire de remplacer la méthode par défaut par une méthode davantage adaptée à la classe en question.

1.10.4 Interfaces

L'héritage multiple est interdit en Java. Les interfaces sont un moyen de résoudre en partie le problème.
Une **interface** est un ensemble de constantes et de méthodes vides. Il est impossible d'instancier une interface. On utilise une interface en créant une classe qui en hérite et qui contient le corps des méthodes qui y sont déclarées.. Dans le cas où la classe mère est une interface, le mot-clé **implements** prend la place de **extends**.
En voici un exemple d'utilisation :

```
package heritage;

import java.util.ArrayList;

interface Saluer
{
        public void direBonjour();
}

class Bonjour implements Saluer
{
        @Override
```

```java
        public void direBonjour()
        {
                System.out.println("Bonjour");
        }
}

class Hello implements Saluer
{
        @Override
        public void direBonjour()
        {
                System.out.println("Hello");
        }
}

class GutenTag implements Saluer
{
        @Override
        public void direBonjour()
        {
                System.out.println("Guten tag");
        }
}

public class ExempleInterface
{
        public static void main(String[] args)
        {
                Saluer s = new Bonjour();
                s.direBonjour();
                s = new Hello();
                s.direBonjour();
                ArrayList<Saluer> arrayList = new ArrayList<Saluer>();
                arrayList.add(new Bonjour());
                arrayList.add(new Hello());
                arrayList.add(new GutenTag());
                for (Saluer saluer : arrayList)
                        saluer.direBonjour();
        }
}
```

Vous remarquez que la méthode `Saluer` est implémentée par les trois classes `Bonjour`, `Hello` et `GutenTag`, qui contiennent trois façons différentes de programmer la méthode **public void** `saluer()`.
Le polymorphisme permet de mettre dans une référence de type `Saluer` tout objet dont le type hérite de `Saluer`.
On peut voir une interface comme un contrat :

- Toute sous-classe d'une interface se doit d'implémenter toutes les méthodes qui y sont déclarées.
- En échange, il devient possible d'utiliser le polymorphisme, et donc d'utiliser le même code avec des objets de types différents.

La restriction interdisant l'héritage multiple en Java ne s'applique pas aux interfaces. Une classe peut implémenter plusieurs interfaces.

1.10.5 Classes Abstraites

Nous voulons représenter des sommes dans des devises différentes et implémenter automatiquement les conversions. Nous allons créer une classe devise qui contiendra l'attribut somme et nous utiliserons l'héritage pour implémenter les spécificités des diverses devises (Euros, Dollars, Livres, ...).

```java
package heritage;
```

```java
public abstract class Devise
{
	private double somme = 0;

	/*
	 * Nombre de devises pour 1$.
	 */

	public abstract double getCours();

	public abstract String getUnite();

	protected void setSomme(double somme)
	{
		this.somme = somme;
	}

	protected void setSomme(Devise d)
	{
		setSomme(d.getSomme() * this.getCours() / d.getCours());
	}

	public double getSomme()
	{
		return somme;
	}

	@Override
	public String toString()
	{
		return "somme = " + somme + " " + getUnite();
	}

	public static void main(String[] args)
	{
		Devise devise = new Dollars(12);
		System.out.println(devise);
		devise = new Euros(devise);
		System.out.println(devise);
		devise = new Livres(devise);
		System.out.println(devise);
		devise = new Dollars(devise);
		System.out.println(devise);
	}
}

class Livres extends Devise
{
	public Livres(Devise d)
	{
		setSomme(d);
	}

	public Livres(double somme)
	{
		setSomme(somme);
	}

	@Override
```

```java
        public double getCours()
        {
                return 0.76636574;
        }

        @Override
        public String getUnite()
        {
                return "Livres";
        }
}

class Euros extends Devise
{
        public Euros(Devise d)
        {
                setSomme(d);
        }

        public Euros(double somme)
        {
                setSomme(somme);
        }

        @Override
        public double getCours()
        {
                return 0.895744319;
        }

        @Override
        public String getUnite()
        {
                return "Euros";
        }
}

class Dollars extends Devise
{
        public Dollars(Devise d)
        {
                setSomme(d);
        }

        public Dollars(double somme)
        {
                setSomme(somme);
        }

        @Override
        public double getCours()
        {
                return 1.;
        }

        @Override
        public String getUnite()
        {
                return "Dollars";
```

```
        }
}
```

Ces classes permettent de prendre en charge automatiquement les conversions entre devises. La classe `Devise` contient une méthode `setSomme` qui est surchargée et qui peut prendre en paramètre soit une somme exprimée dans la bonne unité, soit une autre devise.
On remarque qu'il est impossible d'implémenter `getCours`() car le cours varie selon la devise. La classe `Euros` hérite de la classe `Devise` et implémente `getCours`(). La classe `Dollars` fonctionne de façon similaire.

1.11 Exceptions

Le mécanisme des exceptions en Java (et dans d'autres langages objets) permet de gérer de façon élégante les erreurs pouvant survenir à l'exécution. Elles présentent trois avantages :

- Obliger les programmeurs à gérer les erreurs.
- Séparer le code de traitement des erreurs du reste du code.
- Rattraper les erreurs en cours d'exécution.

1.11.1 Rattraper une exception

Lorsqu'une séquence d'instructions est susceptible d'occasionner une erreur, on la place dans un bloc **try**. Comme ci-dessous :

```
try
{
  /*
        bloc d'instructions
  */
}
```

On place juste après ce bloc un bloc **catch** qui permet de traiter l'erreur :

```
try
{
  /*
        bloc d'instructions
  */
}
catch(nomErreur e)
{
  /*
        Traitement de l'erreur
  */
}
```

Une **exception** est une **erreur pouvant être rattrapée en cours d'exécution**. Cela signifie que si une erreur survient, une exception est **levée**, le code du **try** est interrompu et le code contenu dans le **catch** est exécuté. Par exemple,

```
try
{
  int i = 0;
  while(true)
  {
    t[i++]++;
  }
}
catch(ArrayIndexOutOfBoundException e)
{
  System.out.println("An exception has been raised.");
}
```

Dans le code précédent l'indice `i` est incrémenté jusqu'à ce que cet indice dépasse la taille du tableau. Dans ce cas l'exception `ArrayIndexOutOfBoundException` est levée et le code correspondant à cette exception dans le **catch** est exécuté.

1.11.2 Méthodes levant des exceptions

Si vous rédigez une méthode susceptible de lever une exception, vous **devez** le déclarer en ajoutant à son entête l'expression **throws** <`listeExceptions`>. Par exemple,

```java
public void bourreTableau() throws ArrayIndexOutOfBoundException
{
  int i = 0;
  while(true)
  {
    t[i++]++;
  }
}
```

Vous remarquez qu'il n'y a pas de **try** ... **catch**. Cela signifie que l'erreur doit être rattrapée dans le sous-programme appelant. Par exemple,

```java
try
{
  bourreTableau();
}
catch(ArrayIndexOutOfBoundException e)
{
  System.out.println("Il fallait s'y attendre...");
}
```

Bref, lorsqu'une instruction est susceptible de lever une exception, vous devez soit la rattraper tout de suite, soit indiquer dans l'entête du sous-programme qu'une exception peut se propager et qu'elle doit donc être traitée dans le sous-programme appelant.

1.11.3 Propagation d'une exception

Par traiter une exception, on entend soit la rattraper tout de suite, soit la **propager**. Par propager, on entend laisser le sous-programme appelant la traiter. Il est possible lorsque l'on invoque un sous-programme levant une exception, de la propager. Par exemple,

```java
public void appelleBourreTableau() throws ArrayIndexOutOfBoundException
{
  bourreTableau();
}
```

On observe que `appelleBourreTableau` se contente de transmettre l'exception au sous-programme appelant. Ce mode de fonctionnement fait qu'une exception non rattrapée va se propager dans la pile d'appels de sous-programmes jusqu'à ce qu'elle soit rattrapée. Et si elle n'est jamais rattrapée, c'est la JVM, qui va le faire.

1.11.4 Définir une exception

Une exception est tout simplement une classe héritant de la classe `Exception`.

```java
class MonException extends Exception
{
    public MonException()
    {
        System.out.println("Exception monException has been raised...");
    }

    public String toString()
    {
      return "You tried to do an illegal assignement !";
    }
}
```

1.11.5 Lever une exception

On lève une exception en utilisant la syntaxe **throw new** <nomexception>(<parametres>). L'instanciation de l'exception se fait donc en même temps que sa levée. Par exemple,

```
try
{
  /*
  ......
  */
  throw new MonException();
  /*
  ......
  */
}
catch(MonException e)
{
  System.out.println(e);
}
```

1.11.6 Rattraper plusieurs exceptions

Il se peut qu'une même instruction (ou suite d'instruction) soit susceptible de lever des exceptions différentes, vous pouvez dans ce cas placer plusieurs **catch** à la suite du même **try**.

```
try
{
  bourreTableau();
}
catch(ArrayIndexOutOfBoundException e)
{
  System.out.println("Il fallait s'y attendre...");
}
catch(Exception e)
{
  System.out.println(e);
}
```

Notez bien que comme toute exception hérite de `Exception`, le deuxième **catch** sera exécuté quelle que soit l'exception levée (sauf si bien entendu le premier **catch** est exécuté).

1.11.7 Finally

Il est quelquefois nécessaire qu'une section de code s'exécute quoi qu'il advienne (pour par exemple fermer un fichier). On place dans ce cas une section **finally**.

```
try
{
  /*
  ......;
  */
}
catch(/* ...*/)
{
  /*
  .....
  */
}
finally
```

```
{
  /*
  .....
  */
}
```

Par exemple,

```
package exceptions;

class MonException extends Exception
{
        @Override
        public String toString()
        {
                return "Fallait pas invoquer cette methode...";
        }
}

public class Finally
{
        public static void main(String[] args)
        {
                try
                {
                        try
                        {
                                throw new MonException();
                        }
                        catch (MonException e)
                        {
                                throw new MonException();
                        }
                        finally
                        {
                                System.out.println("Tu t'afficheras quoi qu'il advienne !");
                        }
                }
                catch (Exception e)
                {
                        System.out.println(e);
                }
        }
}
```

1.11.8 RuntimeException

Je vous ai menti au sujet de `ArrayIndexOutOfBoundException`, il s'agit d'une exception héritant de `RuntimeException` qui lui-même hérite de `Exception`. Un code Java dans lequel une `RuntimeException` n'est pas traitée passera la compilation. Par contre, `MonException` n'est pas une `RuntimeException` et doit être rattrapée !

1.12 Interfaces graphiques

1.12.1 Fenêtres

La classe `JFrame`

Une fenêtre est un objet de type `JFrame`, classe à laquelle on accède avec la commande d'importation **import** `javax.swing.*`. Nous utiliserons les méthodes suivantes :

- **public void** `setTitle(String title)`, pour définir le titre de la fenêtre.
- **public void** `setVisible(`**boolean** `b)`, pour afficher la fenêtre.
- **public void** `pack()`, pour adapter les dimensions de la fenêtre aux composants qu'elle contient.
- **public void** `setDefaultCloseOperation(`**int** `operation)`, pour déterminer le comportement de la fenêtre lors de sa fermeture.

Exemples

Voici un code de création d'une fenêtre vide.

```
package ihm;

import javax.swing.*;

public class PremiereFenetre
{
	public static void main(String[] args)
	{
		JFrame f = new JFrame();
		f.setVisible(true);
		f.setTitle("My first window !");
		f.setSize(200, 200);
		f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}
}
```

1.12.2 Un premier objet graphique

Nous allons ajouter un bouton dans notre fenêtre. Les boutons sont de type `JButton`, le constructeur de cette classe prend en paramètre le libellé du bouton. Un des attributs de la classe JFrame est un objet contenant tous les composants graphiques de la fenêtre, on accède à cet objet par la méthode **public** `Container getContentPane()`. Tout `Container` possède une méthode **public** `Component add(Component comp)` permettant d'ajouter n'importe quel composant graphique (par exemple un `JButton`...). On ajoute donc un Bouton de la façon suivante `getContentPane().add(`**new** `JButton("my First JButton"));`.

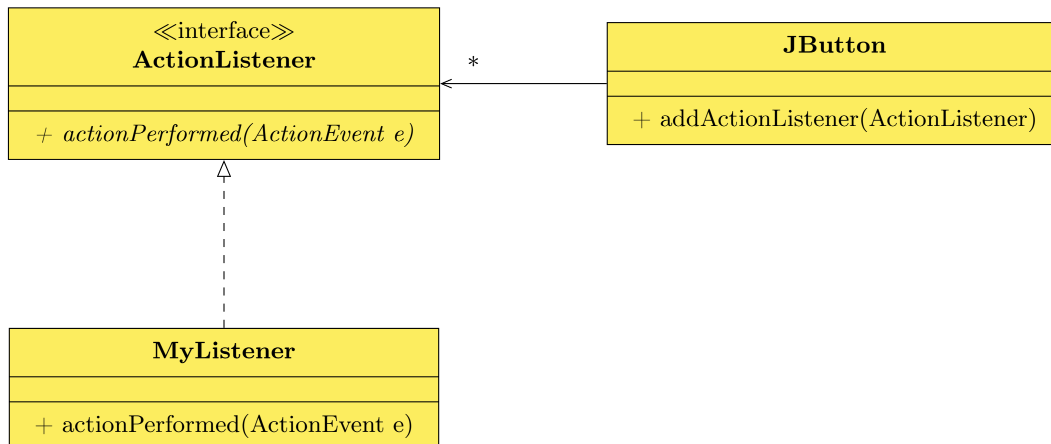
```
package ihm;

import javax.swing.*;
import java.awt.*;

public class PremiersJButtons
{
	public PremiersJButtons()
	{
		JFrame frame = new JFrame();
		frame.setTitle("My second window !");
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.getContentPane().setLayout(new FlowLayout());
		frame.getContentPane().add(new JButton("my First JButton"));
		frame.getContentPane().add(new JButton("my Second JButton"));
		frame.getContentPane().add(new JButton("my Third JButton"));
```

```
                frame.setVisible(true);
                frame.pack();
        }

        public static void main(String[] args)
        {
                new PremiersJButtons();
        }
}
```

L'instruction `getContentPane().setLayout(new FlowLayout());` sert à préciser de quelle façon vous souhaitez que les composants soient disposés dans la fenêtre. Nous reviendrons dessus.

1.12.3 Ecouteurs d'événements

Un écouteur d'événement est un objet associé à un composant graphique. Cet objet doit implémenter l'interface `ActionListener` et de ce fait contenir une méthode **public void** `actionPerformed(ActionEvent e)`. Cette méthode est appelée à chaque fois qu'un événement se produit sur le composant. L'objet `ActionEvent` contient des informations sur l'événement en question.



1.12.4 Premier exemple

Dans ce premier exemple, nous allons définir une classe **public class** `PremierEcouteur` **implements** `ActionListener`. Elle servira donc à la fois à contenir la fenêtre et d'écouteur d'événements. L'objet de type `ActionEvent` passé en paramètre contient une méthode **public** `Object getSource()` retournant l'objet ayant déclenché l'événement.

```
package ihm;

import javax.swing.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.awt.*;

public class PremierEcouteur implements ActionListener
{
        @Override
        public void actionPerformed(ActionEvent e)
        {
                System.out.println("Click on " + ((JButton) e.getSource()).getText());
        }

        public PremierEcouteur()
        {
```

```java
                JFrame frame = new JFrame();
                ArrayList<JButton> jButtons = new ArrayList<>();
                frame.setTitle("My third window !");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.getContentPane().setLayout(new FlowLayout());
                jButtons.add(new JButton("my First JButton"));
                jButtons.add(new JButton("my Second JButton"));
                jButtons.add(new JButton("my Third JButton"));
                for (JButton jButton : jButtons)
                {
                        frame.getContentPane().add(jButton);
                        jButton.addActionListener(this);
                }
                frame.setVisible(true);
                frame.pack();
        }

        public static void main(String[] args)
        {
                new PremierEcouteur();
        }
}
```

1.12.5 Classes anonymes

Les classes anonymes sont des classes que l'on peut créer "à la volée", c'est-à-dire au moment de leur instanciation.. On s'en sert généralement pour implémenter une interface, en l'occurrence `ActionListener`. Voici une autre implémentation faisant intervenir des classes anonymes.

```java
package ihm;

import javax.swing.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.awt.*;

public class EcouteurAnonyme
{
        public EcouteurAnonyme()
        {
                JFrame frame = new JFrame();
                ArrayList<JButton> jButtons = new ArrayList<>();
                frame.setTitle("My fourth window !");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.getContentPane().setLayout(new FlowLayout());
                jButtons.add(new JButton("my First JButton"));
                jButtons.add(new JButton("my Second JButton"));
                jButtons.add(new JButton("my Third JButton"));
                jButtons.add(new JButton("my Fourth JButton"));
                jButtons.add(new JButton("my Fifth JButton"));
                for (JButton button : jButtons)
                        frame.getContentPane().add(button);
                /* Classe dediee */
                jButtons.get(0).addActionListener(new Ecouteur());

                /* Classe anonyme */
                jButtons.get(1).addActionListener(new ActionListener()
                {
                        @Override
```

```
                        public void actionPerformed(ActionEvent e)
                        {
                                System.out.println("click on Second JButton");
                        }
                });

                /* Lambda expression */
                jButtons.get(2).addActionListener(
                                (e) -> {System.out.println("click on Third JButton");}
                );

                /* Classe anonyme dans une fonction */
                jButtons.get(3).addActionListener(getEcouteur());

                /* Lambda expression dans une fonction */
                jButtons.get(4).addActionListener(getEcouteurLambda());
                frame.pack();
                frame.setVisible(true);
        }

        private ActionListener getEcouteur()
        {
                return new ActionListener()
                {
                        @Override
                        public void actionPerformed(ActionEvent e)
                        {
                                System.out.println("click on Fourth JButton");
                        }
                };
        };

        private ActionListener getEcouteurLambda()
        {
                return (e) -> {System.out.println("click on Fifth JButton");};
        };

        public static void main(String[] args)
        {
                new EcouteurAnonyme();
        }
}

class Ecouteur implements ActionListener
{
                @Override
                public void actionPerformed(ActionEvent e)
                {
                        System.out.println("click on First JButton");
                }
}
```

1.12.6 Gestionnaires de mise en forme

Un gestionnaire de mise en forme (Layout manager) est un objet permettant de disposer les composants et les conteneurs dans la fenêtre. Nous avons utilisé un gestionnaire de type `FlowLayout`, qui dispose les composants les uns à coté des autres (ou au dessus des autres si ce n'est pas possible) dans leur ordre d'ajout. Le gestionnaire `GridLayout` représente le conteneur comme une grille

```java
package ihm;

import javax.swing.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.awt.*;

public class TestGridLayout implements ActionListener
{
	ArrayList<JButton> jButtons;

	@Override
	public void actionPerformed(ActionEvent e)
	{
		System.out.println("Click on " + ((JButton) e.getSource()).getText());
	}

	public TestGridLayout()
	{
		JFrame frame = new JFrame();
		jButtons = new ArrayList<JButton>();
		frame.setTitle("One More Window !");
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.getContentPane().setLayout(new GridLayout(2, 2));
		jButtons.add(new JButton("North-West"));
		jButtons.add(new JButton("North-East"));
		jButtons.add(new JButton("South-West"));
		jButtons.add(new JButton("South-East"));
		for (JButton jButton : jButtons)
		{
			frame.getContentPane().add(jButton);
			jButton.addActionListener(this);
		}
		frame.pack();
		frame.setVisible(true);
	}

	public static void main(String[] args)
	{
		new TestGridLayout();
	}
}
```

1.12.7 Un exemple complet : Calcul d'un carré

```java
package ihm;

import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Carre
{
	JTextField operand = new JTextField();
	JLabel result = new JLabel();

	private void afficheCarre()
```

```
        {
                try
                {
                        int k = Integer.parseInt(operand.getText());
                        k *= k;
                        result.setText("" + k);
                }
                catch (NumberFormatException e)
                {
                        result.setText("");
                }
        }

        private KeyListener getKeyListener()
        {
                return new KeyAdapter()
                {
                        @Override
                        public void keyReleased(KeyEvent e)
                        {
                                afficheCarre();
                        }
                };
        }

        private JPanel getMainPanel()
        {
                JPanel panel = new JPanel();
                panel.setLayout(new GridLayout(2, 2));
                panel.add(new JLabel("x = "));
                operand.addKeyListener(getKeyListener());
                panel.add(operand);
                panel.add(new JLabel("x^2 = "));
                panel.add(result);
                return panel;
        }

        public Carre()
        {
                JFrame frame = new JFrame();
                frame.setTitle("Square computer !");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setContentPane(getMainPanel());
                frame.setVisible(true);
                frame.pack();
        }

        public static void main(String[] args)
        {
                new Carre();
        }
}
```

1.13 Tests unitaires

1.13.1 Exemple

Spécification

```java
package testsUnitaires;

public interface Puissance
{
	/**
	 * Retourne x + 1. x quelconque.
	 */

	public int succ(int x);

	/**
	 * Retourne x - 1. x quelconque.
	 */

	public int pred(int x);

	/**
	 * Retourne a + b. a et b quelconques.
	 */

	public int somme(int a, int b);

	/**
	 * Retourne a * b. a et b quelconques.
	 */

	public int produit(int a, int b);

	/**
	 * Retourne base^exp.
	 * base quelconque, exp positif ou nul.
	 */

	public int puissance(int base, int exp);
}
```

L'interface `Puissance` indique quelles fonctionnalités doivent être remplies par toutes ses classes filles.

Implémentation

Voici un exemple d'implémentation :

```java
package testsUnitaires;

public class ImplementationPuissance implements Puissance
{
	public int succ(int x)
	{
		return x + 1;
	}

	public int pred(int x)
	{
		return -succ(-x);
```

```java
	}

	public int somme(int a, int b)
	{
		if (b == 0)
			return a;
		if (b > 0)
			return somme(succ(a), pred(b));
		else
			return somme(pred(a), succ(b));
	}

	public int produit(int a, int b)
	{
		if (b == 0)
			return 0;
		if (b > 0)
			return somme(produit(a, pred(b)), a);
		else
			return somme(produit(a, succ(b)), -a);
	}

	public int puissance(int base, int exp)
	{
		if (exp == 0)
			return 1;
		if ((exp & 1) == 0)
			return puissance(produit(base, base), exp >> 1);
		return produit(puissance(base, pred(exp)), base);
	}
}
```

Comment s'assurer que ce code fonctionne correctement ?

Il existe en Java (et en programmation en général) plusieurs façons d'effectuer des tests.

- Exécuter le programme et vérifier s'il se comporte conformément à ce qui est attendu.
- Tester les fonctions une par une en affichant les valeurs qu'elles retournent.
- Appeler les fonctions en comparant automatiquement les résultats retournés aux résultats attendus.
- Utiliser JUnit.

1.13.2 Test à la bourrin

```java
package testsUnitaires;

public class TestBourrin
{
	public static void main(String[] args)
	{
		Puissance p = new ImplementationPuissance();
		System.out.println(p.puissance(2, 10));
	}
}
```

1.13.3 Test des fonctions

```java
package testsUnitaires;

public class TestFonctions
```

```
{
        public static void main(String[] args)
        {
                Puissance p = new ImplementationPuissance();
                System.out.println("2 = " + p.succ(1));
                System.out.println("5 = " + p.pred(6));
                System.out.println("2 = " + p.somme(2, 0));
                System.out.println("9 = " + p.somme(3, 6));
                System.out.println("10 = " + p.somme(13, -3));
                System.out.println("2 = " + p.produit(2, 1));
                System.out.println("18 = " + p.produit(3, 6));
                System.out.println("-18 = " + p.produit(-3, 6));
                System.out.println("1 = " + p.puissance(4, 0));
                System.out.println("5 = " + p.puissance(5, 1));
                System.out.println("9 = " + p.puissance(3, 2));
                System.out.println("1024 = " + p.puissance(2, 10));
        }
}
```

1.13.4 Test des fonctions automatisé

```
package testsUnitaires;

public class TestFonctionsAutomatique
{
        private boolean ok = true;

        private void teste(String fonction, int obtenu, int attendu)
        {
                if (attendu == obtenu)
                        System.out.println("test OK");
                else
                {
                        System.out.println("test " + fonction + " échoué : " + obtenu
                                        + " != " + attendu);
                        ok = false;
                }
        }

        public boolean teste(Puissance p)
        {
                ok = true;
                try
                {
                        teste("succ", p.succ(1), 2);
                        teste("pred", p.pred(6), 5);
                        teste("somme", p.somme(2, 0), 2);
                        teste("somme", p.somme(3, 6), 9);
                        teste("somme", p.somme(13, -3), 10);
                        teste("produit", p.produit(2, 1), 2);
                        teste("produit", p.produit(3, 6), 18);
                        teste("produit", p.produit(-3, 6), -18);
                        teste("produit", p.produit(3, -6), -18);
                        teste("puissance", p.puissance(4, 0), 1);
                        teste("puissance", p.puissance(5, 1), 5);
                        teste("puissance", p.puissance(3, 2), 9);
                        teste("puissance", p.puissance(2, 10), 1024);
                }
```

```
                catch (Exception e)
                {
                        ok = false;
                        System.out.println(e);
                }
                if (ok)
                        System.out.println("----> Soooo goood");
                else
                        System.out.println("----> Au moins un test a échoué.");
                return ok;
        }

        public static void main(String[] args)
        {
                Puissance p = new ImplementationPuissance();
                TestFonctionsAutomatique t = new TestFonctionsAutomatique();
                t.teste(p);
        }
}
```

1.13.5 Tests unitaires

```
package testsUnitaires;

import static org.junit.Assert.*;
import org.junit.Test;

public class TestsJUnit
{
        private Puissance p = new ImplementationPuissance();

        @Test
        public void testSucc()
        {
                assertEquals("test de la fonction successeur", 2, p.succ(1));
        }

        @Test
        public void testPred()
        {
                assertEquals("pred", 5, p.pred(6));
        }

        @Test
        public void testSomme()
        {
                assertEquals("somme", 2, p.somme(2, 0));
                assertEquals("somme", 9, p.somme(3, 6));
                assertEquals("somme", 10, p.somme(13, -3));
        }

        @Test
        public void testProduit()
        {
                assertEquals("produit", 2, p.produit(2, 1));
                assertEquals("produit", 18, p.produit(3, 6));
                assertEquals("produit", -18, p.produit(-3, 6));
        }
```

```java
        @Test
        public void testPuissance()
        {
                assertEquals("puissance", 1, p.puissance(4, 0));
                assertEquals("puissance", 5, p.puissance(5, 1));
                assertEquals("puissance", 9, p.puissance(3, 2));
                assertEquals("puissance", 1024, p.puissance(2, 10));
        }
}
```

1.13.6 Logs

```java
package testsUnitaires;

import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Logger;
import java.util.logging.SimpleFormatter;

public class PuissanceAvecLog extends ImplementationPuissance
{
        private final static Logger LOGGER = Logger.getLogger(PuissanceAvecLog.class
                        .getName());
        static
        {
                try
                {
                        FileHandler handler = new FileHandler("log/puissance.log");
                        LOGGER.addHandler(handler);
                        handler.setFormatter(new SimpleFormatter());
                }
                catch (IOException e)
                {
                        e.printStackTrace();
                }
        }

        @Override
        public int puissance(int base, int exp)
        {
                String str = "Call to puissance : " + base + "^" + exp + " = ";
                if (exp == 0)
                        str += 1;
                else if ((exp & 1) == 0)
                        str += "" + base * base + "^" + (exp >> 1);
                else
                        str += "" + base + "*(" + base + "^" + (exp - 1) + ")";
                LOGGER.info(str);
                return super.puissance(base, exp);
        }
}
```

1.14 Collections

1.14.1 Types paramétrés

Déclaration

On crée une classe paramétrée en plaçant le type inconnu (dans l'exemple ci-dessous : `T`) entre chevrons à côté du nom de la classe.

```
package collections.exemples;

public class ClasseParametree<T>
{
        private T data;

        public ClasseParametree(T data)
        {
                this.data = data;
        }

        public T get()
        {
                return data;
        }

        public void set(T data)
        {
                this.data = data;
        }

        public static void main(String[] args)
        {
                ClasseParametree<String> conteneurString = new ClasseParametree<>(
                                "toto");
                String chaine = conteneurString.get();
                System.out.println(chaine);

                ClasseParametree<Integer> conteneurInt = new ClasseParametree<>(5);
                int entier = conteneurInt.get();
                System.out.println(entier);
        }
}
```

On peut voir une classe paramétrée comme un *moule à classes*.

Héritage entre classes paramétrées

On est amené naturellement à se demander si `ClasseParametree<String>` hérite de `ClasseParametree<Object>`. Un exercice traite cette question...

Java Vs C++

Le mécanisme des classes paramétrées en Java en bien plus fin que les templates utilisées en C++. Les templates ne sont que de vulgaires rechercher/remplacer, alors qu'en Java, le code compilé est le même que si les types ne sont pas paramétrés. Les types paramétrés de java ne sont donc qu'un moyen pour le compilateur de s'assurer que vous faites les choses proprement en vous obligeant à déclarer les types que vous placez dans les collections.

1.14.2 Paramètres et héritage

Paramètre sur la classe mère

Il est possible depuis une classe non paramétrée d'hériter d'une classe paramétrée en précisant son type :

```
package collections.exemples;

public class StringWrapper implements Comparable<StringWrapper>
{
	private String data;

	public StringWrapper(String data)
	{
		this.data = data;
	}

	public String getData()
	{
		return data;
	}

	public void setData(String data)
	{
		this.data = data;
	}

	@Override
	public int compareTo(StringWrapper autre)
	{
		return data.compareTo(autre.getData());
	}

	public static void main(String[] args)
	{
		StringWrapper e1 = new StringWrapper("toto"),
				e2 = new StringWrapper("tutu");
		System.out.println(e1.compareTo(e2));
	}
}
```

Contraintes sur le paramètre

Il est souvent nécessaire que le type servant de paramètre vérifie certaines propriétés. Si vous souhaitez par exemple que l'objet hérite de `Comparable`, vous avez la possibilité de poser cette contrainte avec le paramètre `<T extends Comparable<T>>`.

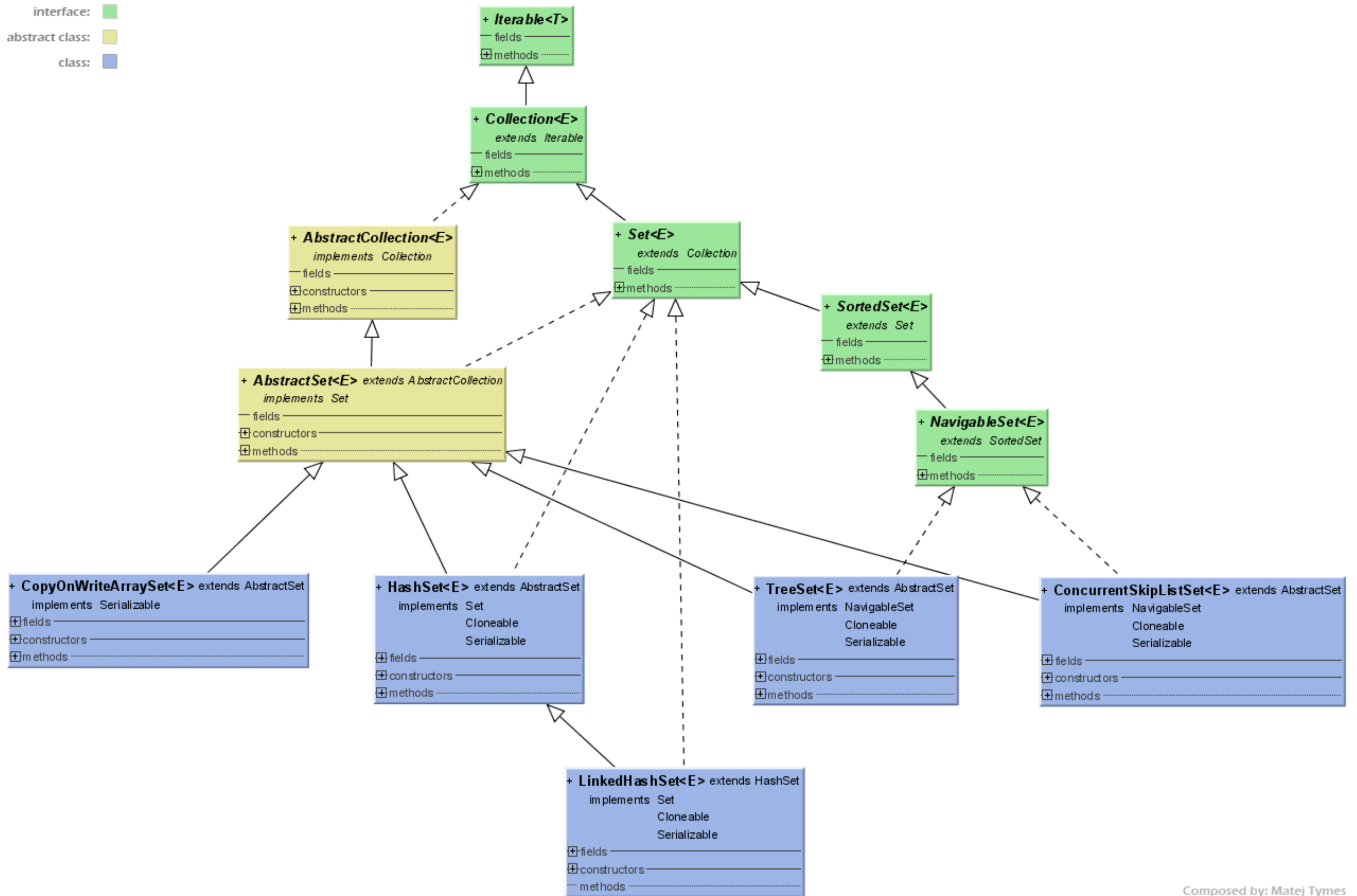
```
package collections.exemples;

public class ComparableWrapper<T extends Comparable<T>>
	implements Comparable<ComparableWrapper<T>>
{
	private T data;

	public ComparableWrapper(T data)
	{
		this.data = data;
	}

	public T getData()
```

```
        {
                return data;
        }

        public void setData(T data)
        {
                this.data = data;
        }

        @Override
        public int compareTo(ComparableWrapper<T> autre)
        {
                return data.compareTo(autre.getData());
        }

        public static void main(String[] args)
        {
                ComparableWrapper<String> e1 = new ComparableWrapper<>("toto"),
                                e2 = new ComparableWrapper<>("tutu");
                System.out.println(e1.compareTo(e2));
                ComparableWrapper<Integer> i1 = new ComparableWrapper<>(4),
                                i2 = new ComparableWrapper<>(3);
                System.out.println(i1.compareTo(i2));
        }
}
```

1.14.3 Collections standard

La distribution officielle de Java est fournie avec plusieurs milliers de classes. Les `Collections` sont des structures de données permettant d'optimiser des opérations comme le tri, la recherche de plus petit élément, etc. Les collections sont donc des regroupements d'objets.

Set<T>



Un ensemble (Set<T>) est un regroupement d'éléments de type T, non ordonnés et sans doublons.

```java
package collections.exemples;

import java.util.HashSet;
import java.util.Set;

public class SetInscriptions
{
	public static void main(String[] args)
	{
		Set<String> inscrits = new HashSet<>();
		inscrits.add("Lucien");
		inscrits.add("Raymond");
		inscrits.add("Huguette");
		System.out.println(inscrits.contains("Gégé"));
		System.out.println(inscrits.contains("Raymond"));
		for (String nom : inscrits)
			System.out.println(nom);
	}
}
```

SortedSet<T>

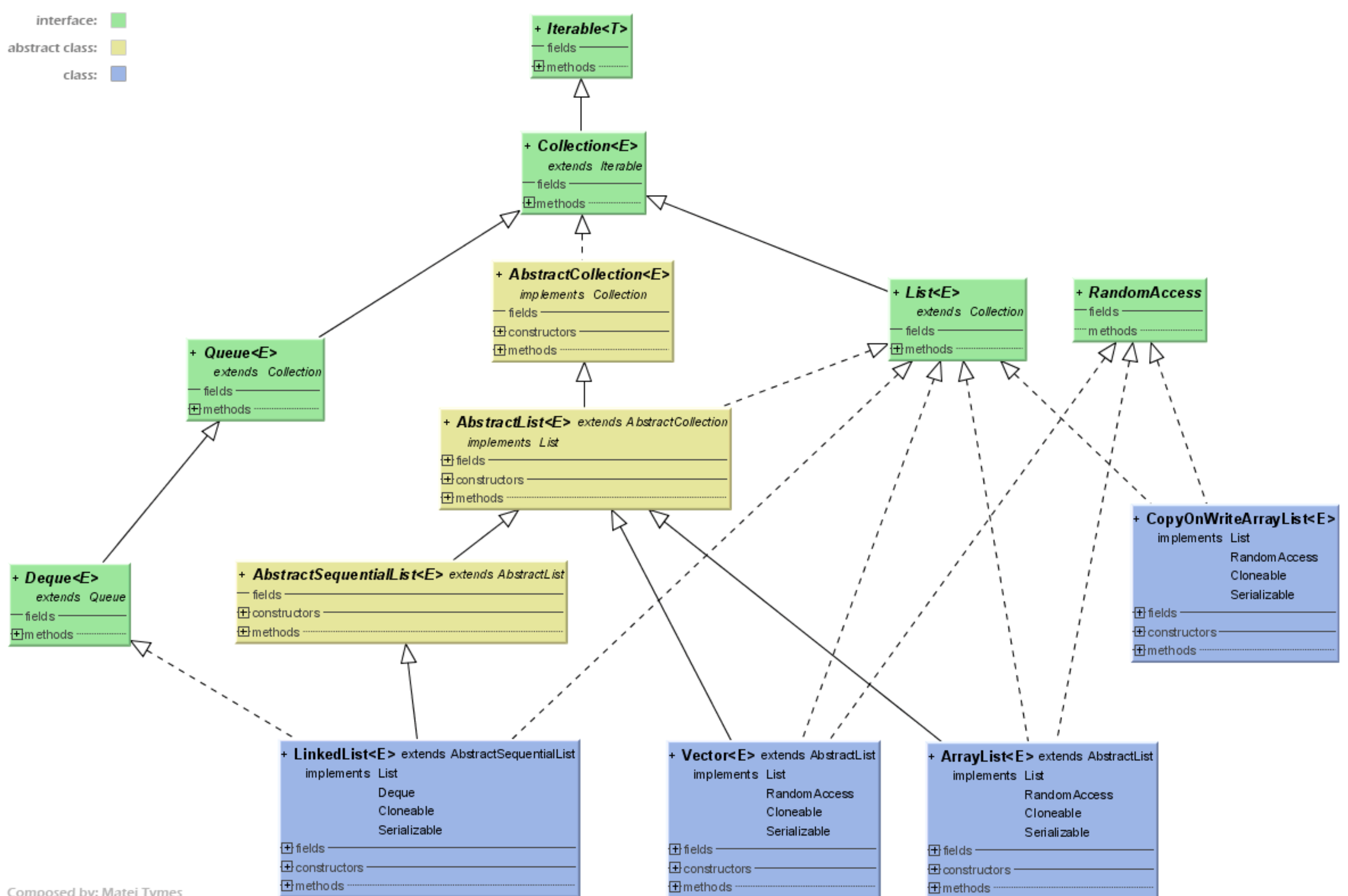
```java
package collections.exemples;

import java.util.SortedSet;
import java.util.TreeSet;

class Coordonnees implements Comparable<Coordonnees>
{
	private int x, y;

	public Coordonnees(int x, int y)
	{
		this.x = x;
		this.y = y;
	}

	@Override
	public String toString()
	{
		return x + " " + y;
	}

	@Override
	public int compareTo(Coordonnees autre)
	{
		if (x == autre.x)
			return y - autre.y;
		return x - autre.x;
	}
}

public class SortedSetCoordonnees
{
	public static void main(String[] args)
	{
		SortedSet<Coordonnees> cases = new TreeSet<>();
		cases.add(new Coordonnees(1, 6));
		cases.add(new Coordonnees(7, 3));
		cases.add(new Coordonnees(-2, 5));
		cases.add(new Coordonnees(1, 5));
		for (Coordonnees c : cases)
			System.out.println(c);
	}
}
```

```java
package collections.exemples;

import java.util.Comparator;
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetComparator
{
	public static void main(String[] args)
	{
		SortedSet<String> dico = new TreeSet<>(new Comparator<String>()
		{
			@Override
```

```
                        public int compare(String str1, String str2)
                        {
                                return str1.length() - str2.length();
                        }
                });
                dico.add("abc");
                dico.add("abcde");
                dico.add("ab");
                dico.add("abcdefg");
                for (String s : dico)
                        System.out.println(s);

        }
}
```

Lists<T>

Une liste (List<T>) est un ensemble d'éléments de type T, disposés dans l'ordre dans lequel ils ont été insérés, et contenant éventuellement des doublons.

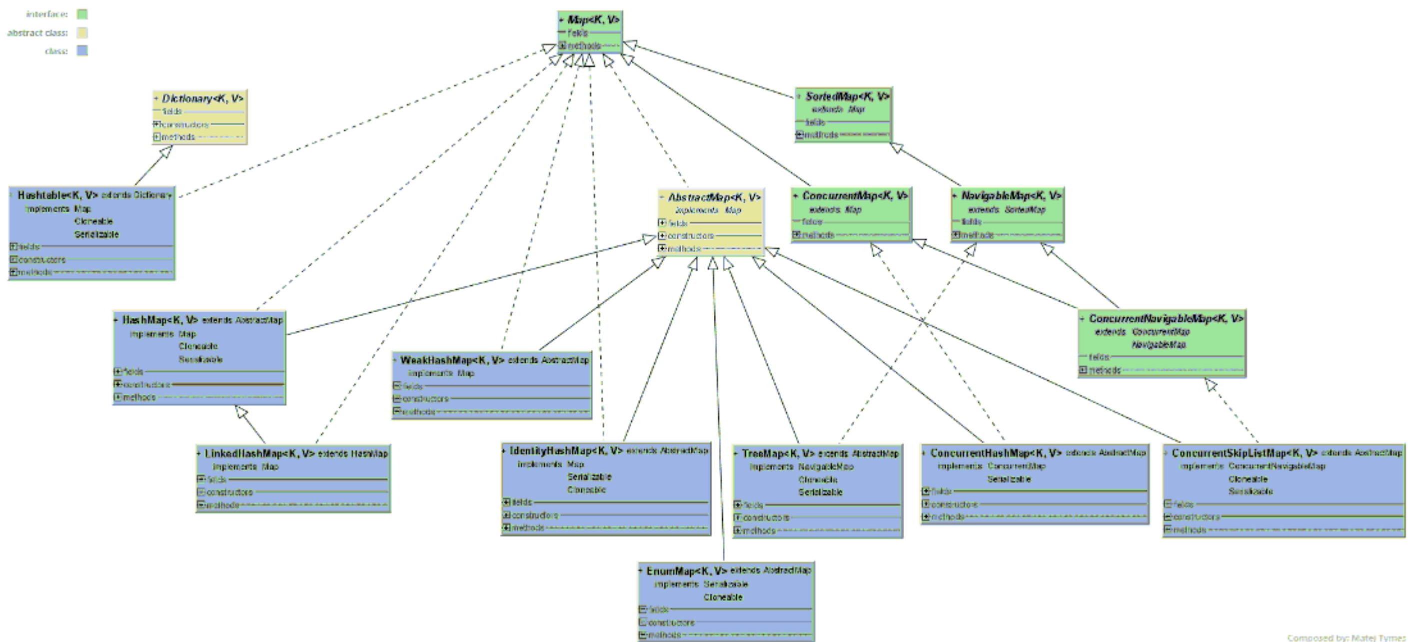



```
package collections.exemples;

import java.util.LinkedList;
```

```java
import java.util.List;

public class ListPalmares
{
        public static void main(String[] args)
        {
                List<String> palmares = new LinkedList<>();
                palmares.add("Ginette");
                palmares.add("Gertrude");
                palmares.add("Maurice");
                for (String nom : palmares)
                        System.out.println(nom);
        }
}
```

Maps<T>

Une application, (Map<K, T>) permet d'associer à des **clés** de type K des objets de type T.



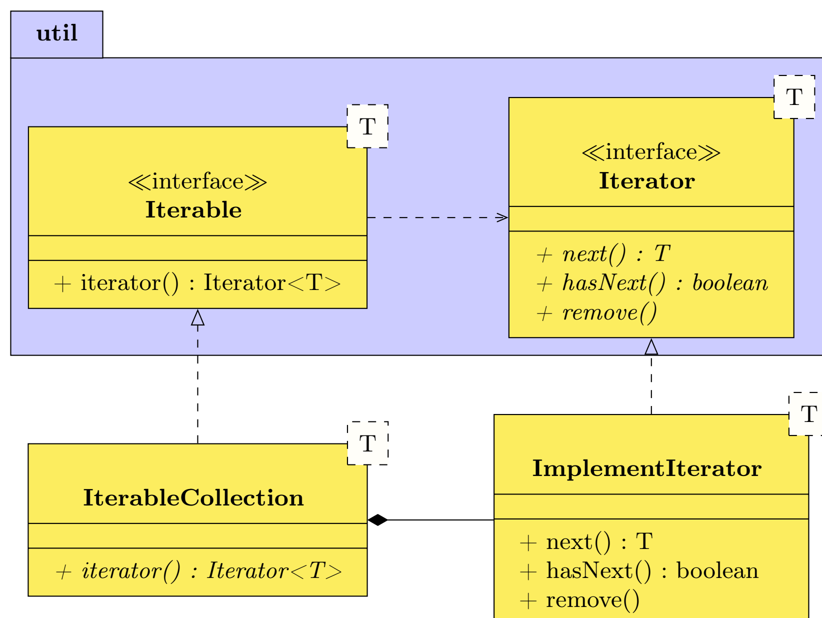

```java
package collections.exemples;

import java.util.HashMap;
import java.util.Map;

public class HashMapSalaires
{
        public static void main(String[] args)
        {
                Map<String, Integer> salaires = new HashMap<>();
                salaires.put("Jojo", 1000);
                salaires.put("Marcel", 2000);
                salaires.put("Ursule", 3000);
                System.out.println("Le salaire de Marcel est " + salaires.get("Marcel")
```

```
                                        + " euros.");
        }
}
```

```
package collections.exemples;

import java.util.Map.Entry;
import java.util.SortedMap;
import java.util.TreeMap;

public class SortedMapSalaires
{
        public static void main(String[] args)
        {
                SortedMap<String, Integer> salaires = new TreeMap<>();
                salaires.put("Dédé", 5000);
                salaires.put("Marcel", 2000);
                salaires.put("Ginette", 3000);
                salaires.put("Lucienne", 1000);
                for (String e : salaires.keySet())
                        System.out.println(e);
                for (int e : salaires.values())
                        System.out.println(e);
                for (Entry<String, Integer> e : salaires.entrySet())
                        System.out.println("Le salaire de " + e.getKey() + " est " + e.
                            getValue());
        }
}
```

`Iterable<T>`

Il n'est possible d'utiliser la boucle **for** simplifiée que si collection parcourue implémente `Iterable<T>`.



Voici un exemple de collection itérable :

```
package collections.exemples;
```

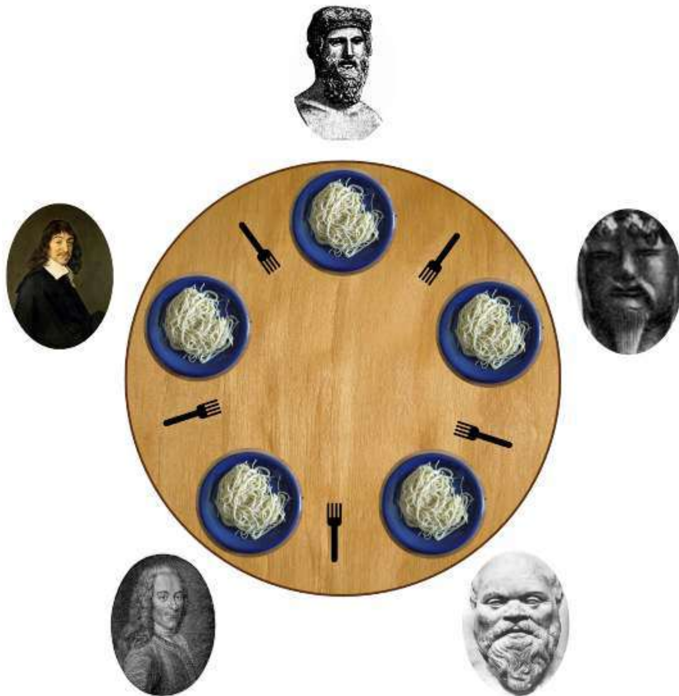
```java
import java.util.Iterator;

public class IterableArray implements Iterable<Integer>
{
        private final int TAILLE;
        private int[] tableau;

        public IterableArray(int taille)
        {
                this.TAILLE = taille;
                tableau = new int[taille];
        }

        public void set(int i, int data)
        {
                tableau[i] = data;
        }

        public int get(int i)
        {
                return tableau[i];
        }

        @Override
        public Iterator<Integer> iterator()
        {
                return new Iterator<Integer>()
                {
                        private int index = 0;

                        @Override
                        public boolean hasNext()
                        {
                                return index < TAILLE;
                        }

                        @Override
                        public Integer next()
                        {
                                int item = get(index);
                                index++;
                                return item;
                        }

                        @Override
                        public void remove()
                        {
                                for (int i = index; i < TAILLE; i++)
                                        set(i, get(i + 1));
                        }
                };
        }

        public static void main(String[] args)
        {
                IterableArray tab = new IterableArray(10);
                for (int i = 0; i < 10; i++)
                        tab.set(i, i + 1);
```

```
                for (int value : tab)
                        System.out.println(value);
        }
}
```

1.15 Threads

1.15.1 Le dîner des philosophes

Les programmes peuvent être décomposés en **processus légers** (eng. **threads**) s'exécutant en parallèle de façon asynchrone. Ils sont susceptibles d'accéder à des ressources communes pour se transmettre des données. Le dîner des philosophes est une illustration des problèmes se posant lorsque l'on manipule des processus.



(Illustration par Benjamin D. Esham / Wikimedia Commons, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=56559`)

Un philosophe, pour manger, va utiliser les deux couverts qui sont à côté de son assiette. De la sorte, ses deux voisins ne peuvent pas manger en même temps que lui. Ce modèle est une transposition de ce qui se produit lorsque des programmes (les philosophes) ont besoin de ressources communes (les couverts). Un philosophe se comportera de la façon suivante une fois face à son assiette :

```
Prendre le couvert gauche
Prendre le couvert droit
Manger
Reposer le couvert droit
Reposer le couvert gauche
```

L'interblocage

Si jamais un des couverts qu'il doit prendre n'est pas disponible, il devra attendre que celui-ci se libère. Dans le cas où le couvert gauche serait disponible mais pas le droit, le philosophe prendra le couvert gauche et le tiendra jusqu'à ce que le droit se libère, empêchant de la sorte un autre philosophe, à sa gauche, de manger.

La pire situation est celle dans laquelle les philosophes arrivent tous en même temps, prennent chacun le couvert se trouvant à leur gauche, et attendent tous que leur couvert droit se libère. Ils resteront tous bloqués sur la première étape de leur algorithme, formant ce que l'on appelle un **interblocage**, (eng. **deadlock**).

La famine

Une solution pourrait être de libérer le couvert gauche si le droit n'est pas disponible. Mais malheureusement cela pourrait conduire à un autre problème s'appelant la **famine**. Dans le cas où des philosophes se relaierait pour toujours

manger à côté de notre philosophe *fair-play*, celui-ci se retrouverait en attente indéfiniment.

1.15.2 Lancement

En java, on définit un thread de deux façons :

— En héritant de la classe `Thread`

— En implémentant l'interface `Runnable`

Bien que la première solution soit généralement plus commode, la deuxième est quelquefois le seul moyen d'éviter l'héritage multiple. Nous détaillerons le premier cas, le deuxième est décrit dans la documentation.

La classe `Thread`

La classe `Thread` dispose entre autres de deux méthodes

— **public void** `start()` qui est la méthode permettant de démarrer l'exécution du thread.

— **public void** `run()` qui est la méthode automatiquement invoquée par start quand le thread est démarré.

```
package threads;

public class BinaireAleatoire extends Thread
{
	private int value;
	private int nbIterations;

	public BinaireAleatoire(int value, int nbIterations)
	{
		this.value = value;
		this.nbIterations = nbIterations;
	}

	@Override
	public void run()
	{
		for (int i = 1; i <= nbIterations; i++)
			System.out.print(value);
	}

	public static void main(String[] args)
	{
		Thread un = new BinaireAleatoire(1, 30);
		Thread zero = new BinaireAleatoire(0, 30);
		un.start();
		zero.start();
	}
}
```

L'interface `Runnable`

Le constructeur de la classe `Thread` est surchargé pour prendre un paramètre une instance `Runnable`. `Runnable` est une interface contenant une méthode **public void** `run()`, celle-ci sera invoquée par le thread au moment de son lancement.

1.15.3 Synchronisation

Le modèle producteur/consommateur

Le modèle producteur/consommateur se construit à l'aide de deux programmes :

— Le producteur transmet des données en les faisant transiter par une mémoire tampon.

— Le consommateur traite les données produites en les récupérant dans la mémoire tampon.

Lorsque la mémoire tampon est pleine, le producteur doit se mettre *en sommeil*, et lorsque la mémoire tampon est vide, c'est au consommateur de se mettre en sommeil. Lorsque le producteur place une donnée dans une mémoire tampon vide, il réveille le consommateur, et lorsque le consommateur libère de la place dans une mémoire tampon pleine, il réveille le producteur. Le comportement du producteur est décrit par l'algorithme suivant :

```
Si la mémoire tampon est pleine alors
  (*)
  Se mettre en sommeil
Sinon
  Placer une donnée dans la mémoire tampon
  Si la mémoire tampon était vide alors
    Réveiller le consommateur
  Fin si
Fin si
```

Et celui du consommateur est le suivant :

```
Si la mémoire tampon est vide alors
  Se mettre en sommeil
Sinon
  Récupérer une donnée dans la mémoire tampon
  Si la mémoire tampon était pleine alors
    Réveiller le producteur
  Fin si
Fin si
```

Le problème des réveils perdus

La commutation entre les processus peut avoir lieu à n'importe quel moment. Si par exemple, le producteur est interrompu à l'endroit indiqué l'étoile (*), le signal de réveil risque d'être envoyé par le consommateur avant que le producteur ne s'endorme. Le signal de réveil étant perdu, le producteur ne se réveillera pas. Le consommateur pendant se temps va vider la mémoire tampon pour s'endormir à son tour. A la fin, chacun des deux processus sera en sommeil et attendra que l'autre le réveille.

Section critique

Une **section critique** est un bloc d'instructions qu'il est impossible d'interrompre. Une section critique se construit avec le mot-clé **synchronized**.

Méthodes synchronisées

Une méthode synchronisée verrouille un objet pendant son exécution, et met en attente les autres threads tentant d'accéder à l'objet. On synchronise une méthode en plaçant le mot clé **synchronized** dans sa définition.

Instructions synchronisées

On synchronise des instructions en les plaçant dans un bloc

```
synchronized(o)
{
  /* ... */
}
```

Où `o` est l'objet ne pouvant être accédé par deux threads simultanément.

1.15.4 Mise en Attente

Un thread peut décider de se mettre en attente s'il a besoin pour s'exécuter de données qui ne sont pas encore disponibles. On gère cela avec les instructions suivantes :

- **public void** `wait()`**throws** `InterruptedException` met le thread en attente.
- **public void** `notify()` réveille un thread en attente.
- **public void** `notifyAll()` réveille tous les threads en attente.

On place en général ces instructions dans une section critique. Un `wait()` libère le verrou pour autoriser d'autres threads à accéder à la ressource. Un `notify()` choisit un des objets placés en attente sur la même ressource, lui rend le verrou, et relance son exécution. Par exemple,

```java
package threads;

public class Counter
{
	private int value = 0;
	private int upperBound;
	private int lowerBound;

	public Counter(int lowerBound, int upperBound)
	{
		this.upperBound = upperBound;
		this.lowerBound = lowerBound;
		value = (upperBound + lowerBound) / 2;
	}

	public synchronized void increaseCounter() throws InterruptedException
	{
		while (value == upperBound)
			wait();
		value++;
		System.out.println("+ 1 = " + value);
		if (value == lowerBound + 1)
			notify();
	}

	public synchronized void decreaseCounter() throws InterruptedException
	{
		while (value == lowerBound)
			wait();
		value--;
		System.out.println("- 1 = " + value);
		if (value == upperBound - 1)
			notify();
	}

	public static void main(String[] args)
	{
		Counter c = new Counter(0, 100);
		Thread p = new Plus(c);
		Thread m = new Moins(c);
		p.start();
		m.start();
	}
}

class Plus extends Thread
{
	private Counter c;
```

```java
        Plus(Counter c)
        {
                this.c = c;
        }

        @Override
        public void run()
        {
                while (true)
                        try{c.increaseCounter();}
                        catch (InterruptedException e){}

        }
}

class Moins extends Thread
{
        private Counter c;

        Moins(Counter c)
        {
                this.c = c;
        }

        @Override
        public void run()
        {
                while (true)
                        try{c.decreaseCounter();}
                        catch (InterruptedException e){}
        }
}
```

Ce programme affiche aléatoirement les valeurs prises par un compteur incrémenté et décrémenté alternativement par deux threads. Si l'on tente de décrémenter la valeur minimale, le thread de décrémentation s'endort pour laisser la main au thread d'incrémentation. Si le thread d'incrémentation est parti de la valeur minimale, il réveille le thread de décrémentation qui peut reprendre son exécution. Et vice-versa.

1.16 Persistance

La persistance est la possibilité (offerte par tous les langages de programmation) de conserver en mémoire des données entre deux exécutions. Si par exemple vous utilisez des variables lors d'une exécution d'un programme, lors de sa clôture les valeurs de ces variables seront perdues. Plusieurs solutions permettent d'éviter ce problème. Une solution consiste à stocker dans un fichier les données que vous souhaitez rendre persistantes. Une autre solution consiste à utiliser une base de données.

1.16.1 Fichiers

Tout logiciel de programmation permet d'écrire des données dans un fichier. Par exemple :

```
package persistance;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class Fichier
{
	public static void main(String[] args)
	{
		BufferedReader br = null;
		String fileName = "src/persistance/Fichier.java";
		try
		{
			FileInputStream fis = new FileInputStream(fileName);
			InputStreamReader isr = new InputStreamReader(fis);
			br = new BufferedReader(isr);
			String line;
			while ((line = br.readLine()) != null)
				System.out.println(line);
		}
		catch (IOException e)
		{
			System.out.println("Impossible d'ouvrir le fichier " + fileName
					+ ".");
		}
		finally
		{
			try
			{
				if (br != null)
					br.close();
			}
			catch (IOException e)
			{
				e.printStackTrace();
			}
		}
	}
}
```

Le programme ci-dessus affiche son propre code source. On remarquera que `FileInputStream` est un objet permettant de lire le contenu d'un fichier (Il existe une classe `File`, mais elle représente un fichier dans le sens un chemin dans l'arborescence). Dans le cas où il faudrait rendre persistantes plusieurs données hétérogènes, cette méthode pourrait s'avérer fastidieuse. Il existe une méthode permettant de remédier à ce type de problème.

1.16.2 Serialization

La serialization est un mécanisme permettant de stocker des objets dans des fichiers sans se soucier du format qui sera utilisé. Observons le code suivant :

```
package persistance;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Wrapper implements Serializable
{
        private static final long serialVersionUID = 1L;

        private int value;

        public Wrapper(int value)
        {
                this.value = value;
        }

        public int getValue()
        {
                return value;
        }

        public void setValue(int value)
        {
                this.value = value;
        }

        @Override
        public String toString()
        {
                return "" + value;
        }

        public static Wrapper read(String fileName) throws IOException,
            ClassNotFoundException
        {
                ObjectInputStream ois = null;
                try
                {
                        FileInputStream fis = new FileInputStream(fileName);
                        ois= new ObjectInputStream(fis);
                        return (Wrapper) (ois.readObject());
                }
                finally
                {
                        if (ois != null)
                                ois.close();
                }
        }

        public void write(String fileName) throws IOException
        {
```

```
		ObjectOutputStream oos = null;
		try
		{
			FileOutputStream fos = new FileOutputStream(fileName);
			oos = new ObjectOutputStream(fos);
			oos.writeObject(this);
		}
		finally
		{
			if (oos != null)
				oos.close();
		}
	}
}

public class Serialization
{
	public static void main(String[] args)
	{
		String fileName = "serialization.srz";
		Wrapper w = new Wrapper(5);
		System.out.println(w);
		try
		{
			w.write(fileName);
			w.setValue(4);
			Wrapper wBis = Wrapper.read(fileName);
			System.out.println(wBis);// 4 ou 5 ?
		}
		catch (IOException e)
		{
			System.out.println("Impossible d'ouvrir le fichier " + fileName + ".
			   ");
		}
		catch (ClassNotFoundException e)
		{
			System.out.println("Le fichier " + fileName + " est corrompu.");
		}

	}
}
```

Dans le cas où l'objet serialisé référence d'autres objets, ceux-ci seront aussi sérialisés. Ce procédé est donc très puissant, vous pouvez en vous y prenant bien stocker tous les objets utilisés pour un projet en quelques lignes de code.

1.16.3 JDBC

Dans le cas où un application est multi-utilisateurs, la sérialisation est malheureusement insuffisante. La base de données est un moyen de résoudre ce problème. Par exemple :

```
package persistance;

import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBC
{
```

```java
	public static void main(String[] args)
	{
		Connection c = null;
		try
		{
			Class.forName("com.mysql.jdbc.Driver");
			String url = "jdbc:mysql://localhost/test", user = "", password = ""
				;
			c = DriverManager.getConnection(url, user, password);
			String req = "select * from test";
			Statement s = c.createStatement();
			ResultSet rs = s.executeQuery(req);
			while (rs.next())
			{
				System.out.println(rs.getInt(1) + " : " + rs.getString(2));
			}
		}
		catch (ClassNotFoundException e)
		{
			System.out.println("Pilote JDBC non installé.");
		}
		catch (SQLException e)
		{
			System.out.println(e);
		}
		finally
		{
			try
			{
				if (c != null)
					c.close();
			}
			catch (SQLException e)
			{
				System.out.println("Impossible de fermer la connection.");
			}
		}
	}
}
```

1.16.4 L'attaque par injection

Tout dialogue avec un logiciel de stockage de données peut comporter des failles de sécurité. Considérons par exemple le programme suivant :

```java
package persistance.injection;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import commandLineMenus.rendering.examples.util.*;

public class Gruyere
{
	protected Connection c = null;
	protected String login, password;
```

```java
public Gruyere()
{
        try
        {
                Class.forName("com.mysql.jdbc.Driver");
                String url = "jdbc:mysql://localhost/test", user = "", password = ""
                    ;
                c = DriverManager.getConnection(url, user, password);
        }
        catch (ClassNotFoundException e)
        {
                System.out.println("Pilote JDBC non installé.");
        }
        catch (SQLException e)
        {
                System.out.println(e);
        }
}

public void close()
{
        try
        {
                if (c != null)
                        c.close();
        }
        catch (SQLException e)
        {
                System.out.println("Impossible de fermer la connection.");
        }

}

private void saisitIdentifiants()
{
        login = InOut.getString("login : ");
        password = InOut.getString("password : ");
}

public boolean connect()
{
        saisitIdentifiants();
        boolean connexionAcceptee = false;
        try
        {
                ResultSet rs = executeConnect();
                connexionAcceptee = rs.next();
        }
        catch (SQLException e)
        {
                System.out.println(e);
        }
        if (connexionAcceptee)
                System.out.println("Connexion acceptée");
        else
                System.out.println("Accés refusé");
        return connexionAcceptee;
}
```

```
	protected ResultSet executeConnect() throws SQLException
	{
		String req = "select * from utilisateur where login = '" + login
			+ "' and password = '" + password + "'";
		Statement s = c.createStatement();
		return s.executeQuery(req);
	}

	public static void main(String[] args)
	{
		Gruyere gruyere = new Gruyere();
		gruyere.connect();
		gruyere.close();
	}
}
```

On effectue une attaque par injection en saisissant par exemple le login `' OR 1=1#`. Le soin vous est laissé de comprendre ce qu'il se passe. La requête préparée ci-dessous, se charge elle-même d'échapper les caractères spéciaux pour éviter l'injection de scripts.

```
package persistance.injection;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class RequetePreparee extends Gruyere
{
	protected ResultSet executeConnect() throws SQLException
	{
		String req = "select * from utilisateur where login = ? and password = ?";
		PreparedStatement s = c.prepareStatement(req);
		s.setString(1, login);
		s.setString(2, password);
		return s.executeQuery();
	}

	public static void main(String[] args)
	{
		Gruyere requetePreparee = new RequetePreparee();
		requetePreparee.connect();
		requetePreparee.close();
	}
}
```

1.17 Hibernate

1.17.1 Introduction

Dès que l'on souhaite manier des données persistantes dans un contexte autorisant les connexions simultanées, l'utilisation d'une base de données est incontournable.

Les Data Access Objects

Un **DAO** (Data Access Objects) est un objet rendu persistant par une base de données, ou permettant de rendre un objet persistant.

La programmation des **DAO** (Data Access Objects) à l'aide de JDBC oblige le programmeur à mettre au point un code long et répétitif. Ce qui, en plus de nécessiter un code pénible (et difficile) à rédiger, est une source d'erreurs non négligeable.

Les ORM

Un package d'**ORM** (Object/Relationnal Mapping) permet de sous-traiter la persistance des données. Il devient alors possible d'enregistrer ou de lire des objets dans une base de données en quelques instructions.

L'utilisation de ce type de bibliothèque occasionne un gain considérable de temps tout en garantissant un programme bien plus fiable. **Hibernate** est la bibliothèque qui sera présentée dans ce cours, mais les concepts (et même la syntaxe des annotations) que nous passerons en revue sont facilement transposable à un autre outil, voire à un autre langage.

La norme JPA

Hibernate utilise des **annotations**, c'est à dire des indications qui se placent dans les fichiers sources. Les annotations ne sont pas exécutées par le compilateur, mais sont utilisées par d'autres programmes (documenteur, ORM, etc.). Par exemple :

```
@Column(name = "nom")
```

L'annotation ci-dessus est lue par Hibernate et lui indique comment représenter le champ `nom` dans une base de données relationnelle. Les annotations permettant de gérer la persistance des données répondent à la norme **JPA** (Java Persistence Annotations) et sont donc les mêmes d'un ORM à l'autre.

Avant de présenter les nombreuse facettes d'hibernate, je vous propose de commencer par un exemple.

1.17.2 Un premier exemple

Cet exemple, un des plus épurés qu'il est possible de faire, permet de rendre persistant dans une table appelée `personne` des objets contenant les trois champs num, nom, et prénom. Il ne sera pas nécessaire de créer la base de données, Hibernate se chargera tout seul de créer les tables. Suivez scrupuleusement les indications ci-dessous pour exécuter le programme, vous aurez ensuite droit aux explications.

Bibliothèque

Téléchargez les .jars de hibernate 4 (le lien de téléchargement est dans la version html) et décompressez-les.

Créer une bibliothèque `hibernate4` à l'aide d'eclipse, vous y placerez les fichiers du répertoire `/lib/required`.

Base de données

Créez, sous mysql, la base de données que l'on utilisera avec les instructions suivantes :

```
create database courshibernate;
```

Attention à bien respectez les noms et la casse, sinon le programme ne fonctionnera pas. L'instruction suivante permet de créer un compte :

```
grant all privileges on courshibernate.* to 'hibernate'@'localhost' identified by 'hibernate';
```

Prenez note du fait que le nom de la base de données est `courshibernate`, celle de l'utilisateur est `hibernate`, et le password est `hibernate`. Faites attention au fait que si les identifiants ne correspondent pas à ceux que j'ai utilisés, le programme ne fonctionnera pas.

Le fichier java

Le fichier suivant, dont je vous déconseille de changer le nom, est à placer dans `src/hibernate/PremierExemple/`. Il s'agit du code source contenant la classe à mapper ainsi que les appels à Hibernate.

```java
package hibernate.premierExemple;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

@Entity
@Table(name = "personne")
class Personne
{
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	@Column(name = "num")
	private int num;

	@Column(name = "nom")
	private String nom;

	@Column(name = "prenom")
	private String prenom;

	public Personne(String prenom, String nom)
	{
		this.nom = nom;
		this.prenom = prenom;
	}
}

public class PremierExemple
{
	private static Session getSession() throws HibernateException
	{
		Configuration configuration = new Configuration()
				.configure("hibernate/premierExemple/PremierExemple.cfg.xml"
				    );
		ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
				.applySettings(configuration.getProperties()).build();
		SessionFactory sessionFactory = configuration
				.buildSessionFactory(serviceRegistry);
		return sessionFactory.openSession();
```

```java
	}

	public static void main(String[] args)
	{
		try
		{
			Session s = getSession();
			Personne joffrey = new Personne("Joffrey", "Baratheon");
			Transaction t = s.beginTransaction();
			s.persist(joffrey);
			t.commit();
			s.close();
		}
		catch (HibernateException ex)
		{
			throw new RuntimeException("Probleme de configuration : "
					+ ex.getMessage(), ex);
		}
	}
}
```

Le fichier de configuration

Le fichier suivant est à placer dans `src/`. Il contient les identifiants de connexion à la base de données ainsi que les noms des classes qui seront mappées.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory >

	<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
	<property name="hibernate.connection.url">
	    jdbc:mysql://localhost/hibernate
	</property>
	<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
	<property name="hibernate.connection.username">hibernate</property>
	<property name="hibernate.connection.password">hibernate</property>
	<property name="hibernate.show_sql">true</property>

	<property name="hbm2ddl.auto">create</property>

	<mapping class="hibernate.premierExemple.Personne" />

</session-factory>
</hibernate-configuration>
```

Lancement sous Eclipse

Une fois tous les fichiers placés, il convient d'ajouter la librairie `Hibernate` précédemment crée au `Build path` pour que le fichier source puisse compiler. Lorsque vous lancerez l'exécution, un log (en rouge) sera généré et se terminera par l'affichage d'une requête. Si une exception est levée, c'est qu'un fichier a été mal placé ou mal orthographié.

Vous pourrez constater que le programme a fonctionné en consultant la base de données : la table personne a été crée et une personne a été insérée.

1.17.3 Un gestionnaire de contacts en quelques lignes

Le fichier métier

```
package hibernate.gestionContacts;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
class Contact
{
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int num;

	private String nom;

	private String prenom;

	Contact()
	{
	}

	public Contact(String prenom, String nom)
	{
		this.nom = nom;
		this.prenom = prenom;
	}

	int getNum()
	{
		return num;
	}

	void setNum(int num)
	{
		this.num = num;
	}

	public String getNom()
	{
		return nom;
	}

	public void setNom(String nom)
	{
		this.nom = nom;
	}

	public String getPrenom()
	{
		return prenom;
	}

	public void setPrenom(String prenom)
	{
		this.prenom = prenom;
```

```
	}

	@Override
	public String toString()
	{
		return prenom + " " + nom;
	}
}
```

Le point d'entrée

```
package hibernate.gestionContacts;

import java.util.List;

public class GestionContacts
{
	private static GestionContacts gestionContacts = null;

	public List<Contact> getContacts()
	{
		return Passerelle.refreshList();
	}

	public static GestionContacts getGestionContacts()
	{
		if (gestionContacts == null)
			gestionContacts = new GestionContacts();
		return gestionContacts;
	}

	private GestionContacts(){}

	public void sauvegarder(Contact contact)
	{
		Passerelle.save(contact);
	}

	public void supprimer(Contact contact)
	{
		Passerelle.delete(contact);
	}
}
```

La passerelle

```
package hibernate.gestionContacts;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
```

```java
abstract class Passerelle
{
	private static Session session = null;

	static
	{
		SessionFactory sessionFactory = null;
		try
		{
			Configuration configuration = new Configuration()
					.configure("hibernate/gestionContacts/
					   GestionContacts.cfg.xml");
			ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder
			    ()
					.applySettings(configuration.getProperties()).build
					   ();
			sessionFactory = configuration.buildSessionFactory(serviceRegistry);
			session = sessionFactory.openSession();
		}
		catch (HibernateException ex)
		{
			throw new RuntimeException("Probleme de configuration : "
					+ ex.getMessage(), ex);
		}
	}

	static void delete(Contact personne)
	{
		Transaction tx = session.beginTransaction();
		session.delete(personne);
		tx.commit();
	}

	static void save(Contact personne)
	{
		Transaction tx = session.beginTransaction();
		session.save(personne);
		tx.commit();
	}

	@SuppressWarnings("unchecked")
	static java.util.List<Contact> refreshList()
	{
		Query query = session.createQuery("from Contact");
		return query.list();
	}
}
```

Le fichier de configuration

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory >
```

```
                <!-- local connection properties -->
            <property name="hibernate.dialect">
                org.hibernate.dialect.MySQLDialect</property>
                <property name="hibernate.connection.url">
                        jdbc:mysql://localhost/hibernate</property>
                <property name="hibernate.connection.driver_class">
                        com.mysql.jdbc.Driver</property>
                <property name="hibernate.connection.username">
                        hibernate</property>
                <property name="hibernate.connection.password">
                        hibernate</property>

    <property name="hbm2ddl.auto">update</property>

        <!-- property name="hibernate.show_sql">true</property -->
        <property name="hibernate.use_outer_join">true</property>
        <property name="jta.UserTransaction">
                java:comp/UserTransaction</property>
         <mapping class="hibernate.gestionContacts.Contact" />

    </session-factory>
</hibernate-configuration>
```

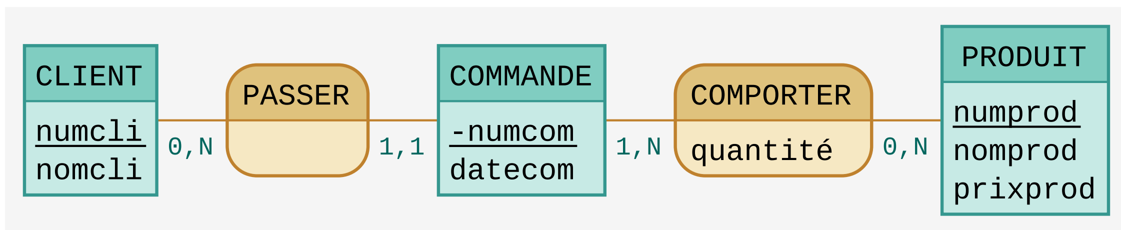
Les entrées/sorties

```
package hibernate.gestionContacts;

import commandLineMenus.List;
import commandLineMenus.Menu;
import commandLineMenus.Option;
import commandLineMenus.rendering.examples.util.InOut;

public class GestionContactsLigneCommande
{
        GestionContacts gestionContacts;

        public GestionContactsLigneCommande(GestionContacts gestionContacts)
        {
                this.gestionContacts = gestionContacts;
                menuPrincipal().start();
        }

        private Option getAfficher()
        {
                return new Option("Afficher", "l",
                                () ->
                                {
                                        for (Contact contact : gestionContacts.getContacts()
                                            )
                                                System.out.println(contact);
                                }
                        );
        }

        private Option getAjouter()
        {
                return new Option("Ajouter", "a",
                                () ->
```

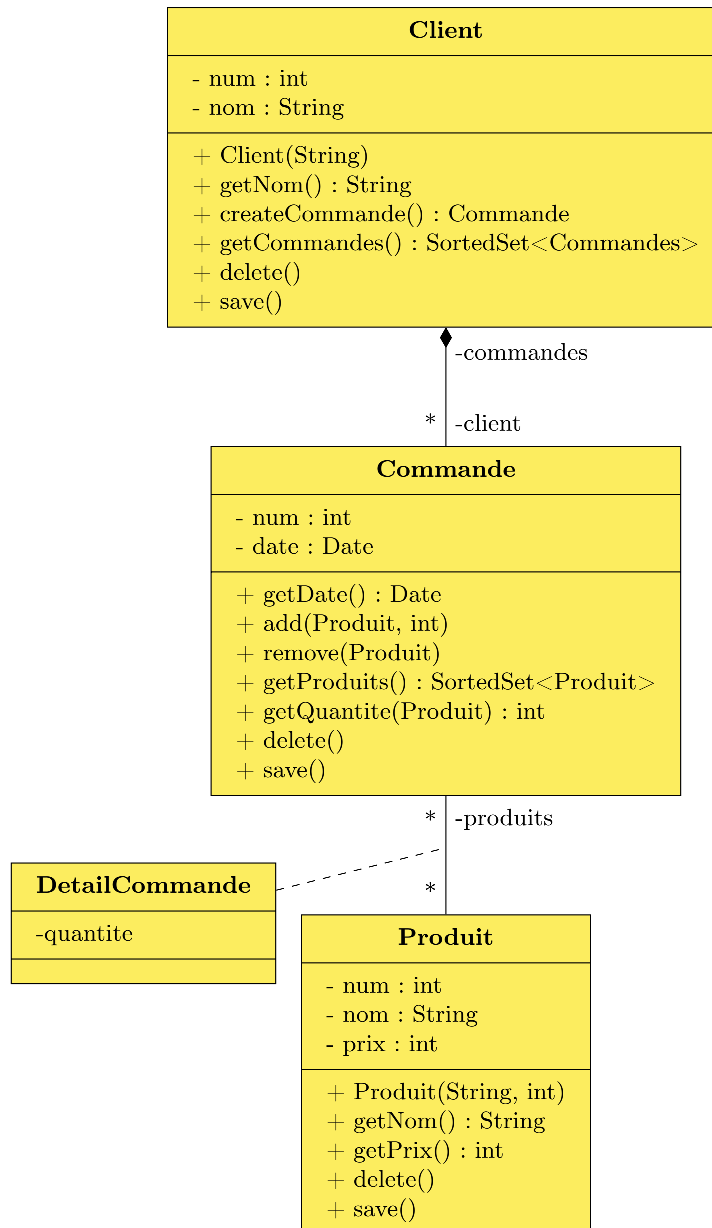
```
                                {
                                        gestionContacts.sauvegarder(new Contact(
                                                        InOut.getString("Prénom : "),
                                                        InOut.getString("Nom : ")));
                                }
                        );
        }

        private Option getSupprimer()
        {
                return new List<>("Supprimer", "s",
                                () -> gestionContacts.getContacts(),
                                (indice, contact) ->
                                        {
                                                gestionContacts.supprimer(contact);
                                        }
                                );
        }

        private Option getModifier()
        {
                return new List<>("Modifier", "m",
                                () -> gestionContacts.getContacts(),
                                (indice, contact) ->
                                        {
                                                contact.setPrenom(InOut.getString("Prénom :
                                                    "));
                                                contact.setNom(InOut.getString("Nom : "));
                                                gestionContacts.sauvegarder(contact);
                                        }
                                );
        }

        private Menu menuPrincipal()
        {
                Menu menu = new Menu("Gestionnaire de contacts");
                menu.add(getAfficher());
                menu.add(getAjouter());
                menu.add(getSupprimer());
                menu.add(getModifier());
                menu.addQuit("q");
                return menu;
        }

        public static void main(String[] args)
        {
                new GestionContactsLigneCommande(GestionContacts.getGestionContacts());
        }
}
```

1.17.4 Un exemple de relations entre les classes

L'exemple suivant utilise la base de données suivante :



Les classes permettant l'exploitation de cette base sont représentées ci-dessous :



La couche métier

```
package hibernate.relations;

import java.util.Collections;
```

```java
import java.util.TreeSet;
import java.util.SortedSet;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

import org.hibernate.annotations.CascadeType;
import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.SortNatural;

@Entity
public class Client
{
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int num;

	private String nom;

	@OneToMany(mappedBy = "client")
	@Cascade(value = { CascadeType.ALL })
	@SortNatural
	private SortedSet<Commande> commandes = new TreeSet<>();

	@SuppressWarnings("unused")
	private Client()
	{
	}

	public Client(String nom)
	{
		this.nom = nom;
	}

	public String getNom()
	{
		return nom;
	}

	int getNum()
	{
		return num;
	}

	public void delete()
	{
		Passerelle.delete(this);
	}

	public void save()
	{
		Passerelle.save(this);
	}

	@Override
	public String toString()
```

```
	{
		return nom + "(" + getCommandes().size() + " commande(s))";
	}

	public Commande createCommande()
	{
		Commande commande = new Commande(this);
		commandes.add(commande);
		return commande;
	}

	void remove(Commande commande)
	{
		commandes.remove(commande);
	}

	public SortedSet<Commande> getCommandes()
	{
		return Collections.unmodifiableSortedSet(commandes);
	}
}
```

```
package hibernate.relations;

import java.util.Date;
import java.util.SortedMap;
import java.util.SortedSet;
import java.util.TreeMap;
import java.util.TreeSet;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.MapKey;
import javax.persistence.OneToMany;

import org.hibernate.annotations.CascadeType;
import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.SortNatural;

@Entity
public class Commande implements Comparable<Commande>
{
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int num;

	private Date date;

	@ManyToOne
	@Cascade(value = { CascadeType.SAVE_UPDATE})
	private Client client;

	@OneToMany(mappedBy = "commande")
	@Cascade(value = { CascadeType.ALL })
	@SortNatural
	@MapKey(name = "produit")
```

```java
	private SortedMap<Produit, DetailCommande> detailsCommandes = new TreeMap<>();

	@SuppressWarnings("unused")
	private Commande()
	{
	}

	Commande(Client client)
	{
		this.date = new Date();
		this.client = client;
	}

	int getNum()
	{
		return num;
	}

	public Client getClient()
	{
		return client;
	}

	public Date getDate()
	{
		return date;
	}

	public void delete()
	{
		client.remove(this);
		Passerelle.delete(this);
	}

	public void save()
	{
		Passerelle.save(this);
	}

	public void add(Produit produit, int quantite)
	{
		detailsCommandes.put(produit, new DetailCommande(this, produit,
				quantite));
	}

	public void remove(Produit produit)
	{
		detailsCommandes.remove(produit);
	}

	public SortedSet<Produit> getProduits()
	{
		return new TreeSet<Produit>(detailsCommandes.keySet());
	}

	SortedSet<DetailCommande> getDetailsCommande()
	{
		return new TreeSet<DetailCommande>(detailsCommandes.values());
	}
```

```java
	public int getNbProduits()
	{
		return detailsCommandes.size();
	}

	public int getQuantite(Produit produit)
	{
		return detailsCommandes.get(produit).getQuantite();
	}

	@Override
	public String toString()
	{
		String s = client.getNom() + "::" + num + " ";
		for (DetailCommande detailCommande : detailsCommandes.values())
			s += detailCommande + " -> ";
		s += "\n";
		return s;
	}

	@Override
	public int compareTo(Commande autre)
	{
		return getDate().compareTo(autre.getDate());
	}
}
```

```java
package hibernate.relations;

import java.util.HashSet;
import java.util.Set;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;

@Entity
public class Produit implements Comparable<Produit>
{
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int num;

	private String nom;

	private double prix;

	@OneToMany(mappedBy = "produit", orphanRemoval=true)
	@Cascade(value = { CascadeType.ALL })
	private Set<DetailCommande> detailsCommandes = new HashSet<>();

	void add(DetailCommande detailCommande)
	{
```

```java
		detailsCommandes.add(detailCommande);
	}

	void remove(DetailCommande detailCommande)
	{
		detailsCommandes.remove(detailCommande);
	}

	public int getNbCommandes()
	{
		return detailsCommandes.size();
	}

	@SuppressWarnings("unused")
	private Produit()
	{
	}

	public Produit(String nom, double prix)
	{
		this.nom = nom;
		this.prix = prix;
	}

	int getNum()
	{
		return num;
	}

	public String getNom()
	{
		return nom;
	}

	public double getPrix()
	{
		return prix;
	}

	public void delete()
	{
		for (DetailCommande detailCommande : detailsCommandes)
			detailCommande.delete();
		Passerelle.delete(this);
	}

	public void save()
	{
		Passerelle.save(this);
	}

	@Override
	public String toString()
	{
		return nom + "(" + prix + " euros)";
	}

	@Override
	public int compareTo(Produit autre)
```

```java
	{
		return getNom().compareTo(autre.getNom());
	}
}
```

```java
package hibernate.relations;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;

@Entity
class DetailCommande implements Comparable<DetailCommande>
{
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int num;

	@ManyToOne
	@Cascade(value = { CascadeType.ALL })
	private Commande commande;

	@ManyToOne
	@Cascade(value = { CascadeType.ALL })
	private Produit produit;

	private int quantite;

	@SuppressWarnings("unused")
	private DetailCommande()
	{
	}

	DetailCommande(Commande commande, Produit produit, int quantite)
	{
		this.commande = commande;
		this.produit = produit;
		produit.add(this);
		this.quantite = quantite;
	}

	int getQuantite()
	{
		return quantite;
	}

	Commande getCommande()
	{
		return commande;
	}

	Produit getProduit()
	{
		return produit;
```

```
	}

	@Override
	public String toString()
	{
		return "" + quantite + " * " + produit.getNom();
	}

	void delete()
	{
		if (commande != null)
		{
			Commande commande = this.commande;
			this.commande = null;
			commande.remove(this.getProduit());
		}
		if (produit != null)
		{
			Produit produit = this.produit;
			this.produit = null;
			produit.remove(this);
		}
		Passerelle.delete(this);
	}

	@Override
	public int compareTo(DetailCommande autre)
	{
		return this.getProduit().compareTo(autre.getProduit());
	}
}
```

La passerelle

```
package hibernate.relations;

import java.util.ArrayList;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

class Passerelle
{
	private static Session session = null;
	private static SessionFactory sessionFactory = null;
	private static final String CONF_FILE = "hibernate/relations/relations.cfg.xml";
	private static Transaction transaction = null;

	static void initHibernate()
	{
		try
```

```
        {
                Configuration configuration = new Configuration()
                                .configure(CONF_FILE);
                ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder
                    ()
                                .applySettings(configuration.getProperties()).build
                                    ();
                sessionFactory = configuration.buildSessionFactory(serviceRegistry);
        }
        catch (HibernateException ex)
        {
                throw new RuntimeException("Probleme de configuration : "
                                + ex.getMessage(), ex);
        }
}

public static void open()
{
        if (sessionFactory == null)
                initHibernate();
        if (!isOpened())
                session = sessionFactory.openSession();
}

public static boolean isOpened()
{
        return session != null && session.isOpen();
}

public static void close()
{
        if (isOpened())
                session.close();
}

static void delete(Object o)
{
        transaction = session.beginTransaction();
        session.delete(o);
        transaction.commit();
        transaction = null;
        session.flush();
}

static void save(Object o)
{
        Transaction tx = session.beginTransaction();
        session.save(o);
        tx.commit();
        session.flush();
}

@SuppressWarnings("unchecked")
public static <T> List<T> getData(String className)
{
        Query query = session.createQuery("from " + className);
        return new ArrayList<T>((List<T>) query.list());
}
```

```
        @SuppressWarnings("unchecked")
        public static <T> T getData(String className, int id)
        {
                Query query = session.createQuery("from " + className + " where num = "
                                + id);
                return (T) (query.list().get(0));
        }
}
```

Le fichier de configuration

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory >

                <!-- local connection properties -->
            <property name="hibernate.dialect">
                org.hibernate.dialect.MySQLDialect</property>
                <property name="hibernate.connection.url">
                        jdbc:mysql://localhost/hibernate_relations</property>
                <property name="hibernate.connection.driver_class">
                        com.mysql.jdbc.Driver</property>
                <property name="hibernate.connection.username">
                        hibernate</property>
                <property name="hibernate.connection.password">
                        hibernate</property>

    <!-- Comportement pour la conservation des tables -->
    <property name="hbm2ddl.auto">create</property>

        <property name="hibernate.show_sql">false</property>
        <property name="jta.UserTransaction">
                java:comp/UserTransaction</property>
        <mapping class="hibernate.relations.Client" />
        <mapping class="hibernate.relations.Commande" />
        <mapping class="hibernate.relations.Produit" />
        <mapping class="hibernate.relations.DetailCommande" />

    </session-factory>
</hibernate-configuration>
```

Le fichier de test

Une configuration d'hibernate est laborieuse et difficile à tester. L'usage de tests unitaires permet un gain de temps et de fiabilité non négligeable (Téléchargeable seulement depuis la version web du cours).

Chapitre 2

Exercices

2.1 Variables

2.1.1 Saisie et affichage

Exercice 1 Saisie d'une chaîne

Écrire un programme demandant à l'utilisateur de saisir son nom, puis affichant le nom saisi.

Exercice 2 Saisie d'un entier

Écrire un programme demandant à l'utilisateur de saisir une valeur numérique entière puis affichant cette valeur.

Exercice 3 Permutation de 2 variables

Saisir deux variables et les permuter avant de les afficher.

Exercice 4 Permutation de 4 valeurs

Écrire un programme demandant à l'utilisateur de saisir 4 valeurs A, B, C, D et qui permute les variables de la façon suivante :

noms des variables	A	B	C	D
valeurs avant la permutation	1	2	3	4
valeurs après la permutation	3	4	1	2

Exercice 5 Permutation de 5 valeurs

On considère la permutation qui modifie cinq valeurs de la façon suivante :

noms des variables	A	B	C	D	E
valeurs avant la permutation	1	2	3	4	5
valeurs après la permutation	4	3	5	1	2

Écrire un programme demandant à l'utilisateur de saisir 5 valeurs que vous placerez dans des variables appelées A, B, C, D et E. Vous les permuterez ensuite de la façon décrite ci-dessus.

Exercice 6 Permutation ultime

Même exercice avec :

noms des variables	A	B	C	D	E	F
valeurs avant la permutation	1	2	3	4	5	6
valeurs après la permutation	3	4	5	1	6	2

2.1.2 Entiers

Exercice 7 Opération sur les entiers

Saisir deux variables entières, afficher leur somme et leur quotient.

2.1.3 Flottants

Exercice 8 Saisie et affichage

Saisir une variable de type `float`, afficher sa valeur.

Exercice 9 Moyenne arithmétique

Saisir 3 valeurs, afficher leur moyenne.

Exercice 10 Surface du rectangle

Demander à l'utilisateur de saisir les longueurs et largeurs d'un rectangle, afficher sa surface.

2.1.4 Caractères

Exercice 11 Prise en main

Affectez le caractère `'a'` à une variable de type `char`, affichez ce caractère ainsi que son code `UNICODE`.

Exercice 12 Casse

Écrivez un programme qui saisit un caractère miniscule et qui l'affiche en majuscule.

2.2 Opérateurs

2.2.1 Conversions

Exercice 1 Successeur

Ecrivez un programme qui saisit un caractère et qui affiche son successeur dans la table des codes `UNICODE`.

Exercice 2 Codes `UNICODE`

Quels sont les codes `UNICODE` des caractères `'0'`, `'1'`, ..., `'9'`

Exercice 3 Moyennes arithmétique et géométrique

Demander à l'utilisateur de saisir deux valeurs a et b et type **float**. Afficher ensuite la différence entre la moyenne arithmétique $\frac{(a+b)}{2}$ et la moyenne géométrique $\sqrt{ab}$. Vous utiliserez l'instruction `Math.Sqrt(f)` qui donne la racine carrée du **double** `f`.

Exercice 4 Cartons et camions

Nous souhaitons ranger des cartons pesant chacun k kilos dans un camion pouvant transporter M kilos de marchandises. Ecrire un programme C demandant à l'utilisateur de saisir M et k, que vous représenterez avec des nombres flottants, et affichant le nombre (entier) de cartons qu'il est possible de placer dans le camion.

2.2.2 Opérations sur les bits (difficiles)

Exercice 5 Codage d'adresses IP

Une adresse `IP` est constituée de 4 valeurs de 0 à 255 séparées par des points, par exemple `192.168.0.1`, chacun de ces nombres peut se coder sur 1 octet. Comme certaines variables de type numérique occupent 4 octets en mémoire, il est possible de s'en servir pour stocker une adresse `IP` entière. Ecrivez un programme qui saisit dans 4 variables chaque valeur constituant une adresse `IP`. Vous créerez ensuite une variable numérique de octets dans laquelle vous juxtaposerez ces quatre valeurs. Ensuite vous mettrez en oeuvre le processus inverse : vous extrairez de cet entier les 4 nombres de l'adresse `IP` et les afficherez en les séparant par des points. Vous devriez ainsi retrouver les quatre nombres saisis par l'utilisateur...

Exercice 6 Permutation circulaire des bits

Effectuez une permutation circulaire vers la droite des bits d'une variable numérique entière, faites de même vers la gauche.

Exercice 7 Permutation de 2 octets

Permutez les deux octets d'une variable numérique à 2 octets saisie par l'utilisateur.

Exercice 8 Inversion de l'ordre de 4 octets

Inversez l'ordre des octets d'une variable numérique de 4 octets saisie par l'utilisateur. Utilisez le code du programme sur les adresses `IP` pour tester votre programme.

2.2.3 Morceaux choisis (difficiles)

Exercice 9 Pièces de monnaie

Nous disposons d'un nombre illimité de pièces de 0.5, 0.2, 0.1, 0.05, 0.02 et 0.01 euros. Nous souhaitons, étant donné une somme S, savoir avec quelles pièces la payer de sorte que le nombre de pièces utilisée soit minimal. Par exemple, la somme de 0.96 euros se paie avec une pièce de 0.5 euros, deux pièces de 0.2 euros, une pièce de 0.05 euros et une pièce de 0.01 euros.

1. Le fait que la solution donnée pour l'exemple est minimal est justifié par une idée plutôt intuitive. Expliquez ce principe sans excès de formalisme.
2. Ecrire un programme demandant à l'utilisateur de saisir une valeur comprise entre 0 et 0.99. Ensuite, affichez le détail des pièces à utiliser pour constituer la somme saisie avec un nombre minimal de pièces.

Exercice 10 Modification du dernier bit

Modifiez le dernier bit d'une variable numérique `a` saisie par l'utilisateur.

Exercice 11 Associativité de l'addition flottante

L'ensemble des flottants n'est pas associatif, cela signifie qu'il existe trois flottants a, b et c, tels que $(a + b) + c \neq a + (b + c)$. Trouvez de tels flottants et vérifiez-le dans un programme.

Exercice 12 Permutation sans variable temporaire

Permutez deux variables `a` et `b` sans utiliser de variable temporaire.

2.3 Conditions

2.3.1 Prise en main

Exercice 1 Majorité

Saisir l'âge de l'utilisateur et lui dire s'il est majeur ou s'il est mineur.

Exercice 2 Valeur Absolue

Saisir une valeur, afficher sa valeur absolue.

Exercice 3 Admissions

Saisir une note, afficher "ajourné" si la note est inférieure à 8, "rattrapage" entre 8 et 10, "admis" dessus de 10.

Exercice 4 Assurances

Une compagnie d'assurance effectue des remboursements sur lesquels est ponctionnée une franchise correspondant à 10% du montant à rembourser. Cependant, cette franchise ne doit pas excéder 4000 euros. Demander à l'utilisateur de saisir le montant des dommages, afficher ensuite le montant qui sera remboursé ainsi que la franchise.

Exercice 5 Valeurs distinctes parmi 2

Afficher sur deux valeurs saisies le nombre de valeurs distinctes.

Exercice 6 Plus petite valeur parmi 3

Afficher sur trois valeurs saisies la plus petite.

Exercice 7 Recherche de doublons

Écrire un algorithme qui demande à l'utilisateur de saisir trois valeurs et qui lui dit s'il s'y trouve un doublon.

Exercice 8 Valeurs distinctes parmi 3

Afficher sur trois valeurs saisies le nombre de valeurs distinctes.

Exercice 9 $ax + b = 0$

Saisir les coefficients a et b et afficher la solution de l'équation $ax + b = 0$.

Exercice 10 $ax^2 + bx + c = 0$

Saisir les coefficients a, b et c, afficher la solution de l'équation $ax^2 + bx + c = 0$.

2.3.2 Switch

Exercice 11 Calculatrice

Ecrire un programme demandant à l'utilisateur de saisir
— deux valeurs a et b, de type *int* ;
— un opérateur *op* de type *char*, vérifiez qu'il s'agit del'une des valeurs suivantes : $+, -, *, /$.
Puis affichez le résultat de l'opération a *op* b.

2.4 Boucles

2.4.1 Compréhension

Exercice 1

Qu'affichent les instructions suivantes ?

```
int a = 1, b = 0, n = 5;
while (a <= n)
        b += a++;
System.out.println(a + ", " + b);
```

Exercice 2

Qu'affichent les instructions suivantes ?

```
int a = 0, b = 0, c = 0, d = 0, m = 3, n = 4;
for (; a < m; a++)
{
        d = 0;
        for (b = 0; b < n; b++)
                d += b;
        c += d;
}
System.out.println(a + ", " + b + ", " + c + ", " + d + ".");
```

Exercice 3

Qu'affichent les instructions suivantes ?

```
int a, b, c, d;
a = 1;
b = 2;
c = a / b;
d = (a == b) ? 3 : 4;
System.out.println(c + ", " + d + ".");
a = ++b;
b %= 3;
System.out.println(a + ", " + b + ".");
b = 1;
for (a = 0; a <= 10; a++)
        c = ++b;
System.out.println(a + ", " + b + ", " + c + ", " + d + ".");
```

2.4.2 Utilisation de toutes les boucles

Les exercices suivants seront rédigés avec les trois types de boucle : tant que, répéter jusqu'à et pour.

Exercice 4 Compte à rebours

Écrire un programme demandant à l'utilisateur de saisir une valeur numérique positive n et affichant toutes les valeurs $n, n-1, \ldots, 2, 1, 0$.

Exercice 5 Factorielle

Ecrire un programme calculant la factorielle (factorielle $n = n! = 1 \times 2 \times \ldots \times n$ et $0! = 1$) d'un nombre saisi par l'utilisateur.

2.4.3 Choix de la boucle la plus appropriée

Pour les exercices suivants, vous choisirez la boucle la plus simple et la plus lisible.

Exercice 6 Table de multiplications

Écrire un programme affichant la table de multiplication d'un nombre saisi par l'utilisateur.

Exercice 7 Tables de multiplications

Écrire un programme affichant les tables de multiplications des nombres de 1 à 10 dans un tableau à deux entrées.

Exercice 8 Puissance

Écrire un programme demandant à l'utilisateur de saisir deux valeurs numériques b et n (vérifier que n est positif) et affichant la valeur b^n.

Exercice 9 Joli carré

Écrire un programme demandant la saisir d'une valeur n et affichant le carré suivant ($n = 5$ dans l'exemple) :

```
n = 5
X  X  X  X  X

X  X  X  X  X

X  X  X  X  X

X  X  X  X  X

X  X  X  X  X
```

2.4.4 Morceaux choisis

Exercice 10 Approximation de 2 par une série

1

On approche le nombre 2 à l'aide de la série $\sum_{i=0}^{+\infty} \frac{1}{2^i}$. Effectuer cette approximation en calculant un grand nombre de termes de cette série. L'approximation est-elle de bonne qualité ?

Exercice 11 Approximation de e par une série

Mêmes questions qu'à l'exercice précédent en e à l'aide de la série $\sum_{i=0}^{+\infty} \frac{1}{i!}$.

Exercice 12 Approximation de e^x par une série

Calculer une approximation de e^x à l'aide de la série $e^x = \sum_{i=0}^{+\infty} \frac{x^i}{i!}$.

Exercice 13 Conversion d'entiers en binaire

Écrire un programme qui affiche un `int` en binaire.

Exercice 14 Conversion de décimales en binaire

Écrire un programme qui affiche les décimales d'un `double` en binaire.

Exercice 15 Inversion de l'ordre des bits

Écrire un programme qui saisit une valeur de type `unsigned short` et qui inverse l'ordre des bits. Vous testerez ce programme en utilisant le précédent.

Exercice 16 Racine carrée par dichotomie

Écrire un algorithme demandant à l'utilisateur de saisir deux valeurs numériques x et p et affichant $\sqrt{x}$ avec une précision p. On utilisera une méthode par dichotomie : à la k-ème itération, on cherche x dans l'intervalle $[min, sup]$, on calcule le milieu m de cet intervalle (à vous de trouver comment le calculer). Si cet intervalle est suffisament petit (à vous de trouver quel critère utiliser), afficher m. Sinon, vérifiez si $\sqrt{x}$ se trouve dans $[inf, m]$ ou dans $[m, sup]$, et modifiez les variables inf et sup en conséquence. Par exemple, calculons la racine carrée de 10 avec une précision 0.5,

- Commençons par la chercher dans $[0, 10]$, on a $m = 5$, comme $5^2 > 10$, alors $5 > \sqrt{10}$, donc $\sqrt{10}$ se trouve dans l'intervalle $[0, 5]$.
- On recommence, $m = 2.5$, comme $\frac{5}{2}^2 = \frac{25}{4} < 10$, alors $\frac{5}{2} < \sqrt{10}$, on poursuit la recherche dans $[\frac{5}{2}, 5]$
- On a $m = 3.75$, comme $3.75^2 > 10$, alors $3.75 > \sqrt{10}$ et $\sqrt{10} \in [2.5, 3.75]$
- On a $m = 3.125$, comme $3.125^2 < 10$, alors $3.125 < \sqrt{10}$ et $\sqrt{10} \in [3.125, 3.75]$
- Comme l'étendue de l'intervalle $[3.125, 3.75]$ est inférieure 2×0.5, alors $m = 3.4375$ est une approximattion à 0.5 près de $\sqrt{10}$.

2.4.5 Extension de la calculatrice

Une calculatrice de poche prend de façon alternée la saisie d'un opérateur et d'une opérande. Si l'utilisateur saisit 3, + et 2, cette calculatrice affiche 5, l'utilisateur a ensuite la possibilité de se servir de 5 comme d'une opérande gauche dans un calcul ultérieur. Si l'utilisateur saisit par la suite $*$ et 4, la calculatrice affiche 20. La saisie de la touche = met fin au calcul et affiche un résultat final.

Exercice 17 Calculatrice de poche

Implémentez le comportement décrit ci-dessus.

Exercice 18 Puissance

Ajoutez l'opérateur $ qui calcule a^b, vous vous restreindrez à des valeurs de b entières et positives.

Exercice 19 Opérations unaires

Ajoutez les opérations unaires racine carrée et factorielle.

2.5 Tableaux

2.5.1 Exercices de compréhension

Qu'affichent les programmes suivants ?

Exercice 1

```
package tableaux;

public class ExerciceUn
{
        public static void main(String[] args)
        {
                char[] c = new char[4];
                c[0] = 'a';
                c[3] = 'J';
                c[2] = 'k';
                c[1] = 'R';
                for(int k = 0 ; k < 4 ; k++)
                        System.out.println(c[k]);
                for(int k = 0 ; k < 4 ; k++)
                        c[k]++;
                for(int k = 0 ; k < 4 ; k++)
                       System.out.println(c[k]);
        }
}
```

Exercice 2

```
package tableaux;

public class ExerciceDeux
{
        public static void main(String[] args)
        {
                int[] k;
                k = new int[10];
                k[0] = 1;
                for(int i = 1 ; i < 10 ; i++)
                        k[i] = 0;
                for(int j = 1 ; j <= 3 ; j++)
                        for(int i = 1 ; i < 10 ; i++)
                                k[i] += k[i - 1];
                for(int i = 1 ; i < 10 ; i++)
                        System.out.println(k[i]);
        }
}
```

Exercice 3

```
package tableaux;

public class ExerciceTrois
{
        public static void main(String[] args)
```

```
        {
                int [] k;
                k = new int [10];
                k[0] = 1;
                k[1] = 1;
                for(int i = 2 ; i < 10 ; i++)
                        k[i] = 0;
                for(int j = 1 ; j <= 3 ; j++)
                        for(int i = 1 ; i < 10 ; i++)
                                k[i] += k[i - 1];
                for(int i = 0 ; i < 10 ; i++)
                        System.out.println(k[i]);
        }
}
```

2.5.2 Prise en main

Exercice 4 Initialisation et affichage

Écrire un programme plaçant dans un tableau `int[] t;` les valeurs $1, 2, \dots, 10$, puis affichant ce tableau. Vous initialiserez le tableau à la déclaration.

Exercice 5 Initialisation avec une boucle

Même exercice en initialisant le tableau avec une boucle.

Exercice 6 Somme

Affichez la somme des n éléments du tableau t.

Exercice 7 Recherche

Demandez à l'utilisateur de saisir un *int* et dites-lui si ce nombre se trouve dans t.

2.5.3 Indices

Exercice 8 Permutation circulaire

Placez dans un deuxième tableau la permutation circulaire vers la droite des éléments de t.

Exercice 9 Permutation circulaire sans deuxième tableau

Même exercice mais sans utiliser de deuxième tableau.

Exercice 10 Miroir

Inversez l'ordre des éléments de t (sans utiliser de deuxième tableau).

2.5.4 Matrices

Exercice 11 Opérations sur les matrices

Programmer les fonctions suivantes :

1. `public static int [][] somme(int [][] a, int [][] b)` retourne la somme des matrices a et b.
2. `public static void echange (int[][] m, int i1, int j1, int i2, int j2)` permute dans m les éléments d'indices (i_1, j_1) avec (i_2, j_2).
3. `public static int [][] retournertTransposee(int[][] m)` retourne la matrice transposée de m.

4. **public static void** `transpose`(**int**[][] `m`) transpose la matrice m.
5. **public static void** `echangeLignes` (**int**[][] `m`, **int** `i1`, **int** `i2`) échange les lignes d'indices i_1 et i_2 de la matrice m.
6. **public static int** [][] `produit`(**int** [][] `a`, **int** [][] `b`) retourne le produit des matrices a et b.
7. **public static int** `plusGrandeColonne`(**int**[][] `m`) retourne la colonne de m dont la somme des valeurs est la plus grande..

Exercice 12 Triangle de Pascal

Écrire une procédure retournant un triangle de Pascal de taille n.

2.6 Les sous-programmes

2.6.1 Initiation

Exercice 1 Compte à rebours

Écrire un sous-programme prenant en paramètre un nombre n et affichant les valeurs n, $n-1$, ..., 1, 0.

2.6.2 Géométrie

Exercice 2 Jolies figures

Écrire un programme demandant à l'utilisateur de saisir une valeur numérique positive n et affichant un carré, une croix et un losange, tous de coté n. Par exemple, si $n = 10$, l'exécution donne

```
Saisissez la taille des figures
10
* * * * * * * * * *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
*                 *
* * * * * * * * * *
                  *
                *   *
              *       *
            *           *
          *               *
        *                   *
      *                       *
    *                           *
  *                               *
*                                   *
  *                               *
    *                           *
      *                       *
        *                   *
          *               *
            *           *
              *       *
                *   *
                  *
*                                   *
  *                               *
    *                           *
      *                       *
        *                   *
          *               *
            *           *
              *       *
                *   *
                  *
```

```
        * *
       *   *
      *     *
     *       *
    *         *
   *           *
  *             *
 *               *
*                 *
```

Vous définirez des sous-programmes de quelques lignes et au plus deux niveaux d'imbrication. Vous ferez attention à ne jamais écrire deux fois les mêmes instructions. Pour ce faire, complétez le code source suivant.

```java
package procedural;

import java.util.Scanner;

public class GeometrieVide
{

        /*
         * Affiche le caractère c
         */

        public static void afficheCaractere(char c)
        {
        }

        /*
         * Affiche n fois le caractère c, ne revient pas à la ligne
         * après le dernier caractère.
         */

        public static void ligneSansReturn(int n, char c)
        {
        }

        /*
         * Affiche n fois le caractère c, revient à la
         *      ligne après le dernier caractère.
         */

        public static void ligneAvecReturn(int n, char c)
        {
        }

        /*
         * Affiche n espaces.
         */

        public static void espaces(int n)
        {
        }

        /*
         * Affiche le caractère c à la colonne i, ne revient
         * pas à la ligne après.
         */
```

```java
public static void unCaractereSansReturn(int i, char c)
{
}

/*
 *  Affiche le caractère c à la colonne i,
 *  revient à la ligne après.
 */

public static void unCaractereAvecReturn(int i, char c)
{
}

/* Affiche le caractère c aux colonnes i et j,
 * revient à la ligne après.
 */

public static void deuxCaracteres(int i, int j, char c)
{
}

/*
 *  Affiche un carré de côté n.
 */

public static void carre(int n)
{
}

/* Affiche un chapeau dont la pointe -
 * non affichée - est sur la colonne centre,
 * avec les caractères c.
 */

public static void chapeau(int centre, char c)
{
}

/*
 * Affiche un chapeau à l'envers avec des caractères c,
 * la pointe - non affichée - est à la colonne centre.
 */

public static void chapeauInverse(int centre, char c)
{
}

/*
 * Affiche un losange de côté n.
 */

public static void losange(int n)
{
}

/*
 *  Affiche une croix de côté n.
 */
```

```
	public static void croix(int n)
	{
	}

	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		int taille;
		System.out.println("Saisissez la taille des figures : ");
		taille = scanner.nextInt();
		scanner.close();
		carre(taille);
		losange(taille);
		croix(taille);
	}
}
```

2.6.3 Arithmétique

Exercice 3 chiffres et nombres

Rappelons que `a % b` est le reste de la division entière de a par b.

1. Écrire la fonction `static int unites(int n)` retournant le chiffre des unités du nombre n.
2. Écrire la fonction `static int dizaines(int n)` retournant le chiffre des dizaines du nombre n.
3. Écrire la fonction `static int extrait(int n, int p)` retourant le p-ème chiffre de représentation décimale de n en partant des unités.
4. Écrire la fonction `static int nbChiffres(int n)` retournant le nombre de chiffres que comporte la représentation décimale de n.
5. Écrire la fonction `static int sommeChiffres(int n)` retournant la somme des chiffres de n.

Exercice 4 Nombres amis

Soient a et b deux entiers strictement positifs. a est un diviseur strict de b si a divise b et $a \neq b$. Par exemple, 3 est un diviseur strict de 6. Mais 6 n'est pas un diviseur strict 6. a et b sont des nombres amis si la somme des diviseurs stricts de a est b et si la somme des diviseurs de b est a. Le plus petit couple de nombres amis connu est 220 et 284.

1. Écrire une fonction `static int sommeDiviseursStricts(int n)`, elle doit renvoyer la somme des diviseurs stricts de n.
2. Écrire une fonction `static boolean sontAmis(int a, int b)`, elle doit renvoyer $true$ si a et b sont amis, $false$ sinon.

Exercice 5 Nombres parfaits

Un nombre parfait est un nombre égal à la somme de ses diviseurs stricts. Par exemple, 6 a pour diviseurs stricts 1, 2 et 3, comme $1 + 2 + 3 = 6$, alors 6 est parfait.

1. Est-ce que 18 est parfait ?
2. Est-ce que 28 est parfait ?
3. Que dire d'un nombre ami avec lui-même ?
4. Écrire la fonction `static boolean estParfait(int n)`, elle doit retourner $true$ si n est un nombre parfait, $false$ sinon.

Exercice 6 Nombres de Kaprekar

Un nombre n est un nombre de Kaprekar en base 10, si la représentation décimale de n^2 peut être séparée en une partie gauche u et une partie droite v tel que $u + v = n$. $45^2 = 2025$, comme $20 + 25 = 45$, 45 est aussi un nombre de Kaprekar. $4879^2 = 23804641$, comme $238 + 04641 = 4879$ (le 0 de 046641 est inutile, je l'ai juste placé pour éviter toute confusion), alors 4879 est encore un nombre de Kaprekar.

1. Est-ce que 9 est un nombre de Kaprekar ?
2. Écrire la fonction **static int** `sommeParties(`**int** `n`, **int** `p)` qui découpe n est deux nombres dont le deuxième comporte p chiffres, et aui retourne leur somme. Par exemple,

$$sommeParties(12540, 2) = 125 + 40 = 165$$

3. Écrire la fonction **static boolean** `estKaprekar(`**int** `n)`

2.6.4 Tableaux

Exercice 7 Miroir

Créer un ensemble de sous-programmes permettant d'inverser l'ordre des éléments d'un tableau (sans utiliser de deuxième tableau).

2.6.5 Pour le sport

Exercice 8 Conversion chiffres/lettres

Écrire un sous-programme prenant un nombre en paramètre et l'affichant en toutes lettres. Rappelons que 20 et 100 s'accordent en nombre s'ils ne sont pas suivis d'un autre nombre (ex. : quatre-vingts, quatre-vingt-un). Mille est invariable (ex. : dix-mille) mais pas million (ex. : deux millions) et milliard (ex. : deux milliards). Depuis 1990, tous les mots sont séparés de traits d'union (ex. : quatre-vingt-quatre), sauf autour des mots mille, millions et milliard (ex. : deux mille deux-cent-quarante-quatre, deux millions quatre-cent-mille-deux-cents). (source : `http://www.leconjugueur.com/frlesnombres.php`).

2.7 Objets

2.7.1 Création d'une classe

Exercice 1 La classe `Rationnel`

Créez une classe `Rationnel` contenant un numérateur et un dénominateur tous deux de type entier. Instanciez deux rationnels a et b et initialisez-les aux valeurs respectives $\frac{1}{2}$ et $\frac{4}{3}$. Affichez ensuite les valeurs de ces champs.

2.7.2 Méthodes

Exercice 2 Opérations sur les Rationnels

Ajoutez à la classe `Rationnel` les méthodes suivantes :

1. **public** `String toString()`, retourne une représentation du rationnel courant sous forme de chaîne de caractères.
2. **public** `Rationnel copy()`, retourne une copie du rationnel courant.
3. **public** `Rationnel opposite()`, retourne l'opposé du rationnel courant.
4. **public** `Rationnel reduce()`, met le rationnel sous forme de fraction irréductible.
5. **public boolean** `isPositive()`, retourne **true** si et seulement si le rationnel courant est strictement positif.
6. **public** `Rationnel add(Rationnel other)`, retourne la somme du rationnel courant et du rationnel `other`.
7. **public** `Rationnel multiply(Rationnel other)`, retourne le produit du rationnel courant avec le rationnel `others`.
8. **public** `Rationnel divide(Rationnel other)`, retourne le quotient du rationnel courant avec le rationnel `others`.
9. **public int** `compareTo(Rationnel other)`, retourne 0 si le rationnel courant est égal au rationnel `other`, -1 si le rationnel courant est inférieur à `other`, 1 dans le cas contraire.

2.8 Encapsulation

2.8.1 Prise en main

Exercice 1 Rationnels propres

Reprenez la classe `Rationnel`. Vous encapsulerez le numérateur, le dénominateur, et vous utiliserez un constructeur pour initialiser les champs privés. Vous prendrez ensuite le soin de modifier les corps des méthodes de sorte que la classe compile et fonctionne correctement.

2.8.2 Implémentation d'une pile

Exercice 2 Implémentation d'une pile avec un tableau

Complétez le code ci-dessous :

```java
package encapsulation;

public class Pile
{
        /*
         * Tableau contenant les elements de la pile.
         */

        private int[] tab;

        /*
         * Taille de la pile
         */

        private final int taille;

        /*
         * Indice du premier element non occupe dans le tableau.
         */

        private int firstFree;

        /*
         * Constructeur
         */

        Pile(int taille)
        {
                this.taille = 0;
        }

        /*
         * Constructeur de copie
         */

        Pile(Pile other)
        {
                this(other.taille);
        }

        /*
         * Retourne vrai si et seulement si la pile est vide
         */
```

```java
public boolean estVide()
{
        return true;
}

/*
 * Retourne vrai si et seulement si la pile est pleine.
 */

public boolean estPleine()
{
        return true;
}

/*
 * Retourne l'element se trouvant au sommet de la pile, -1 si la pile est
 * vide.
 */

public int sommet()
{
        return 0;
}

/*
 * Supprime l'element se trouvant au sommet de la pile, ne fait rien si la
 * pile est vide.
 */

public void depile()
{
}

/*
 * Ajoute data en haut de la pile, ne fait rien si la pile est pleine.
 */

public void empile(int data)
{
}

/*
 * Retourne une representation de la pile au format chaine de caracteres.
 */

public String toString()
{
        return null;
}

/*
 * Teste le fonctionnement de la pile.
 */

public static void main(String[] args)
{
        Pile p = new Pile(30);
        int i = 0;
        while (!p.estPleine())
```

```java
                p.empile(i++);
            System.out.println(p);
            while (!p.estVide())
            {
                System.out.println(p.sommet());
                p.depile();
            }
        }
}
```

2.8.3 Collections

Exercice 3 Parcours

Instanciez un objet de type `ArrayList<>` en utilisant le type de votre choix. Placez-y des valeurs quelconques, et parcourez-le en affichant ses valeurs.

Exercice 4 Boucle for

Même exercice en utilisant la boucle **for** simplifiée.

Exercice 5 Miroir

Écrire une procédure inversant l'ordre des éléments d'un `ArrayList<Integer>`. *Ne pas utiliser la méthode reverse().*

Exercice 6 Tri

Écrire une procédure triant un `ArrayList<Integer>`.

Exercice 7 Clients et factures

Implémentez les classes suivantes :

fillclassHTMLFCED5F fill class=fillclass [x=0,y=0]Client - nom : String
+ Client(String)
+ toString() : String
+ createFacture(int montant) : Facture
+ getFactures() : ArrayList<Facture>
~ add(Facture facture)
~ remove(Facture facture) [x=0,y=-6]Facture - montant : int
- date : LocalDate ~ Facture(Client client, int montant)
+ getDate() : LocalDate
+ getMontant() : int
+ getClient() : Client
+ delete()
+ toString() : String
[arg1=-factures,arg2=-client,mult2=*] ClientFacture

2.8.4 Refactoring de la pile avec des ArrayList

Exercice 8 Pile

Re-téléchargez le fichier `Pile.java`, et sans modifier les méthodes publiques, implémentez la pile en utilisant des (`ArrayList`).

2.8.5 Refactoring de la pile avec des listes chaînées

Exercice 9 Listes chaînées

Complétez le code ci-dessous :

```
package encapsulation;

public class ListeInt
{
	/*
	 * Donnee stockee dans le maillon
	 */

	private int data;

	/*
	 * Pointeur vers le maillon suivant
	 */

	private ListeInt next;

	/*
	 * Constructeur initialisant la donnee et le pointeur vers l'element
	 * suivant.
	 */

	ListeInt(int data, ListeInt next)
	{
	}

	/*
	 * Constructeur initialisant la donnee et mettant le suivant a null.
	 */

	ListeInt(int data)
	{
	}

	/*
	 * Constructeur recopiant tous les maillons de other.
	 */

	ListeInt(ListeInt other)
	{
	}

	/*
	 * Retourne la donnee.
	 */

	public int getData()
	{
		return 0;
	}

	/*
	 * Modifie la donnee
	 */

	public void setData(int data)
```

```
	{
	}

	/*
	 * Retourne le maillon suivant.
	 */

	public ListeInt getNext()
	{
		return null;
	}

	/*
	 * Modifie le maillon suivant
	 */

	public void setNext(ListeInt next)
	{
	}

	/*
	 * Retourne une représentation sous forme de chaine de la liste.
	 */

	public String toString()
	{
		return null;
	}

	/*
	 * Teste le fonctionnement de la liste.
	 */

	public static void main(String[] args)
	{
		ListeInt l = new ListeInt(20);
		int i = 19;
		while (i >= 0)
			l = new ListeInt(i--, l);
		System.out.println(l);

	}

}
```

Exercice 10 Pile

Re-téléchargez le fichier `Pile.java`, et sans modifier les méthodes publiques, implémentez la pile en utilisant des listes chaînées (`ListeInt`).

2.9 Héritage

2.9.1 Héritage

Exercice 1 Point

Définir une classe `Point` permettant de représenter un point dans R^2. Vous utiliserez proprement les concepts d'encapsulation.

Exercice 2 Cercle

Définir une classe `Cercle` héritant de de la classe précédente. Vous prendrez soin de ne pas recoder des choses déjà codées...

2.9.2 Polymorphisme

Exercice 3 Pile

Adaptez la pile ci-dessous afin qu'elle puisse contenir des objets de n'importe quel type. Vous testerez la pile avec des données de types différents.

```
package encapsulation.corriges;

public class Pile
{
        /*
         * Tableau contenant les elements de la pile.
         */

        private int[] tab;

        /*******************************************/

        /*
         * Taille de la pile
         */

        private final int taille;

        /*******************************************/

        /*
         * Indice du premier element non occupe dans le tableau.
         */

        private int firstFree;

        /*******************************************/

        /*
         * Constructeur
         */

        Pile(int taille)
        {
                this.taille = taille;
                tab = new int[taille];
                firstFree = 0;
        }
```

```
/*******************************************/

/*
 * Constructeur de copie.
 */

Pile(Pile other)
{
        this(other.taille);
        firstFree = other.firstFree;
        for (int i = 0; i < firstFree; i++)
                tab[i] = other.tab[i];
}

/*******************************************/

/*
 * Retourne vrai si et seulement si la pile est vide
 */

public boolean estVide()
{
        return firstFree == 0;
}

/*******************************************/

/*
 * Retourne vrai si et seulement si la pile est pleine.
 */

public boolean estPleine()
{
        return firstFree == taille;
}

/*******************************************/

/*
 * Retourne l'element se trouvant au sommet de la pile, -1 si la pile est
 * vide.
 */

public int sommet()
{
        if (estVide())
                return -1;
        return tab[firstFree - 1];
}

/*******************************************/

/*
 * Supprime l'element se trouvant au sommet de la pile, ne fait rien si la
 * pile est vide.
 */

public void depile()
{
```

```
                if (!estVide())
                        firstFree--;
        }

        /*******************************************/

        /*
         * Ajoute data en haut de la pile, ne fait rien si la pile est pleine.
         */

        public void empile(int data)
        {
                if (!estPleine())
                        tab[firstFree++] = data;
        }

        /*******************************************/

        /*
         * Retourne une representation de la pile au format chaine de caracteres.
         */

        public String toString()
        {
                String res = "[";
                for (int i = 0; i < firstFree; i++)
                        res += " " + tab[i];
                return res + " ]";
        }

        /*******************************************/

        /*
         * Teste le fonctionnement de la pile.
         */

        public static void main(String[] args)
        {
                Pile p = new Pile(30);
                int i = 0;
                while (!p.estPleine())
                        p.empile(i++);
                System.out.println(p);
                while (!p.estVide())
                {
                        System.out.println(p.sommet());
                        p.depile();
                }
        }
}
```

Exercice 4 Redéfinition de méthode

Reprenez les classes `Point` et `Cercle`, codez une méthode `toString()` pour les deux classes (mère et fille). Vous prendrez soin d'éviter les redondances dans votre code.

2.9.3 Interfaces

Exercice 5 Animaux

Complétez ce code source, sachant qu'un chien dit "Ouaf!", un chat "Miaou!" et une vache "Meuh!".

```
package heritage;

import java.util.ArrayList;
@SuppressWarnings("unused")

interface Animal
{
        // Setter pour le champ nom
        public void setNom(String nom);

        // Getter pour le champ nom
        public String getNom();

        // Retourne le cri de l'animal
        public String cri();
}

//TODO Décommentez le code ci-dessous pour le compléter

public class ExempleAnimaux
{
//      public static void main(String[] args)
//      {
//              ArrayList<Animal> animaux = new ArrayList<>();
//              animaux.add(new Chat("Ronron"));
//              animaux.add(new Chien("Médor"));
//              animaux.add(new Vache("Huguette"));
//              for (Animal animal : animaux)
//                      System.out.println(animal.cri());
//      }
}

//class Chat implements Animal
//{
//}
//
//class Chien implements Animal
//{
//}
//
//class Vache implements Animal
//{
//}
```

N'hésitez pas à en ajouter d'autres!

Exercice 6 Tab

Reprenez la classe `TableauInt` et adaptez-là de sorte qu'on puisse y placer tout objet dont le type implémente l'interface `Comparable` ci-dessous.

```
package heritage;

import java.util.ArrayList;
import java.util.Random;
```

```java
public class TableauInt
{
	private ArrayList<Integer> t;

	public TableauInt()
	{
		t = new ArrayList<>();
	}

	public TableauInt(TableauInt other)
	{
		t = new ArrayList<>();
		for (int i = 0 ; i < other.taille() ; i++)
			t.add(other.get(i));
	}

	public int taille()
	{
		return t.size();
	}

	public TableauInt copie()
	{
		return new TableauInt(this);
	}

	public String toString()
	{
		String res = "[";
		if (taille() >= 1)
			res += t.get(0);
		for (int i = 1 ; i < taille() ; i++)
			res += ", " + t.get(i);
		res += "]";
		return res;
	}

	public int get(int index)
	{
		return t.get(index);
	}

	public void set(int index, int value)
	{
		int n = taille();
		if (index < n)
			t.set(index, value);
		if (index == n)
			t.add(value);
		if (index < n)
			System.out.println("Achtung !");
	}

	public void echange(int i, int j)
	{
		int temp = t.get(i);
		t.set(i, t.get(j));
		t.set(j, temp);
```

```
	}

	public void triSelection()
	{
		for (int i = 0 ; i < taille() - 1 ; i++)
		{
			int indiceMin = i;
			for (int j = i + 1 ; j < taille() ; j++)
				if (t.get(indiceMin) > t.get(j))
					indiceMin = j;
			echange(i, indiceMin);
		}
	}

	public static void main(String[] args)
	{
		int n = 10 ;
		Random r = new Random();
		TableauInt tab = new TableauInt();
		for (int i = 0; i < n; i++)
			tab.set(i, r.nextInt() % 100);
		System.out.println(tab);
		tab.triSelection();
		System.out.println(tab);
	}
}
```

```
package heritage;

public interface Comparable
{
	/*
	 * Retourne un nombre négatif si l'objet courant est plus petit que other,
	 * 0 s'ils sont égaux, et un nombre positif l'objet courant est plus grand
	 * que other.
	 */

	public int compareTo(Comparable other);
}
```

Testez `TableauInt` avec `Heure` **implements** `Comparable` :

```
package heritage;

import java.util.Random;

public class Heure
{
	private int heures, minutes;

	public Heure(int heures, int minutes)
	{
		this.heures = intAbs(heures) % 24;
		this.minutes = intAbs(minutes) % 60;
	}

	public Heure(Random r)
	{
		this(r.nextInt(), r.nextInt());
	}
```

```java
	private int intAbs(int x)
	{
		if (x > 0)
			return x;
		else
			return -x;
	}

	public int getHeures()
	{
		return heures;
	}

	public int getMinutes()
	{
		return minutes;
	}

	public void setHeures(int heures)
	{
		this.heures = heures;
	}

	public void setMinutes(int minutes)
	{
		this.minutes = minutes;
	}

	@Override
	public String toString()
	{
		return heures + ":" + minutes;
	}

	public int enMinutes()
	{
		return 60 * heures + minutes;
	}
}
```

2.9.4 Classes abstraites

Exercice 7 Animaux

Reprenez le tp sur les animaux, et faites remonter les méthodes `setNom` et `getNom` dans la classe mère.

Exercice 8 Classes abstraites et interfaces

Modifier les classes `Devise` et `Animaux` pour qu'elles implémentent l'interface `Comparable`.

Exercice 9 Tri

Tester `ComparableTab` avec des objets de type `Devise` et `Animaux`.

2.9.5 Un dernier casse-tête

Exercice 10 Arbres syntaxiques

Complétez le code source suivant. Il représente un arbre syntaxique.

```
package heritage;

/**
 * Fonction d'une variable.
 */

public abstract class Function
{
	/**
	 * Retourne l'image de x.
	 */
	public abstract double evaluate(double x);

	/**
	 * Retourne la d&eacute;riv&eacute;e.
	 */
	public abstract Function derivate();

	/**
	 * Retourne la fonction simplifi&eacute;e.
	 */
	public abstract Function simplify();

	/**
	 * Ssi la fonction ne contient pas de variable.
	 */
	public abstract boolean isConstant();

	/**
	 * Ssi la fonction est une feuille valant 0.
	 */
	public abstract boolean isZero();

	/**
	 * Ssi la fonction est une feuille valant 1.
	 */
	public abstract boolean isOne();

	/**
	 * Retourne l'integrale entre a et b (a < b), calcul&eacute;e avec la
	 * m&eacute;thode des trap&egrave;zes en effectuant nbSubdivisions
	 * subdivisions de l'intervalle &agrave; int&eacute;grer.
	 */
	public double integrate(double a, double b, int nbSubdivisions)
	{
		return 0;
	}

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

/**
```

```java
 * f(x) = x.
 */

class Variable extends Function
{
        Variable()
        {
        }

        @Override
        public boolean isZero()
        {
                // TODO à compléter
                return true;
        }

        @Override
        public boolean isOne()
        {
                // TODO à compléter
                return true;
        }

        @Override
        public boolean isConstant()
        {
                // TODO à compléter
                return true;
        }

        @Override
        public Function derivate()
        {
                return null;
        }

        @Override
        public double evaluate(double x)
        {
                return 0;
        }

        @Override
        public Function simplify()
        {
                return null;
        }

        @Override
        public String toString()
        {
                return "";
        }
}

/**
 * Fonction s'exprimant comme une op&eacute;ration binaire entre deux autres
 * fonctions.
 */
```

```java
abstract class BinaryOperator extends Function
{
        protected Function leftSon;
        protected Function rightSon;

        BinaryOperator(Function leftSon, Function rightSon)
        {
        }

        /**
         * Retourne l'op&eacute;rateur binaire sous forme de caract&egrave;re ('+'
         * pour une addition '-' pour une soustraction, etc).
         */
        public abstract char toChar();

        @Override
        public String toString()
        {
                return "(" + leftSon + " " + toChar() + " " + rightSon + ")";
        }

        @Override
        public boolean isZero()
        {
                // TODO à compléter !
                return true;
        }

        @Override
        public boolean isOne()
        {
                // TODO à compléter !
                return true;
        }

        @Override
        public boolean isConstant()
        {
                // TODO à compléter !
                return true;
        }

        /**
         * Remplace les sous-arbres par leurs versions simplifi&eacute;es, retourne
         * une feuille si l'arbre est constant.
         */
        protected Function simplifySubTrees()
        {
                return null;
        }

}

/**
 * f(x) = c, o&ugrave; c est une constante r&eacute;elle.
 */

class Constant extends Function
```

```java
{
        private double value;

        Constant(double value)
        {
        }

        @Override
        public boolean isZero()
        {
                // TODO à compléter
                return true;
        }

        @Override
        public boolean isOne()
        {
                // TODO à compléter
                return true;
        }

        @Override
        public boolean isConstant()
        {
                // TODO à compléter
                return true;
        }

        @Override
        public Function derivate()
        {
                // TODO à compléter
                return null;
        }

        @Override
        public double evaluate(double x)
        {
                // TODO à compléter
                return 0;
        }

        @Override
        public Function simplify()
        {
                // TODO à compléter
                return null;
        }

        @Override
        public String toString()
        {
                // TODO à compléter
                return "";
        }
}

/**
 * f(x) = g(x) - h(x), o&ugrave; g et h sont les sous-arbres gauche et droit.
```

```java
 */

class Minus extends BinaryOperator
{
        Minus(Function leftSon, Function rightSon)
        {
                super(leftSon, rightSon);
        }

        @Override
        public char toChar()
        {
                // TODO à compléter
                return '?';
        }

        @Override
        public double evaluate(double x)
        {
                // TODO à compléter
                return 0;
        }

        @Override
        public Function derivate()
        {
                // TODO à compléter
                return null;
        }

        @Override
        public Function simplify()
        {
                // TODO à compléter
                return null;
        }
}

/**
 * f(x) = g(x) + h(x), o&ugrave; g et h sont les sous-arbres gauche et droit.
 */

class Plus extends BinaryOperator
{
        Plus(Function leftSon, Function rightSon)
        {
                super(leftSon, rightSon);
        }

        @Override
        public char toChar()
        {
                // TODO à compléter
                return '?';
        }

        @Override
        public double evaluate(double x)
        {
```

```
		// TODO à compléter
		return 0;
	}

	@Override
	public Function derivate()
	{
		// TODO à compléter
		return null;
	}

	@Override
	public Function simplify()
	{
		// TODO à compléter
		return null;
	}
}

/**
 * f(x) = g(x) * h(x), o&ugrave; g et h sont les sous-arbres gauche et droit.
 */

class Times extends BinaryOperator
{
	Times(Function leftSon, Function rightSon)
	{
		super(leftSon, rightSon);
	}

	@Override
	public char toChar()
	{
		// TODO à compléter
		return '?';
	}

	@Override
	public double evaluate(double x)
	{
		// TODO à compléter
		return 0;
	}

	@Override
	public Function derivate()
	{
		// TODO à compléter
		return null;
	}

	@Override
	public Function simplify()
	{
		// TODO à compléter
		return null;
	}
}
```

```java
/**
 * f(x) = g(x) / h(x), o&ugrave; g et h sont les sous-arbres gauche et droit.
 */

class Div extends BinaryOperator
{
        Div(Function leftSon, Function rightSon)
        {
                super(leftSon, rightSon);
        }

        @Override
        public char toChar()
        {
                // TODO à compléter
                return '?';
        }

        @Override
        public double evaluate(double x)
        {
                // TODO à compléter
                return 0;
        }

        @Override
        public Function derivate()
        {
                // TODO à compléter
                return null;
        }

        @Override
        public Function simplify()
        {
                // TODO à compléter
                return null;
        }
}
```

2.10 Exceptions

Exercice 1 Âge de l'utilisateur

Calculez l'âge de l'utilisateur en fonction de son année de naissance. Vous ferez le calcul dans une fonction qui déclecheraune exception si l'année de naissance n'est pas dans le passé.

Exercice 2 L'exception inutile

Créer une classe `ClasseInutile` contenant une seule méthode `nePasInvoquer()` contenant une seule instruction, qui lève l'exception `ExceptionInutile`. Lorsqu'elle est levée, l'`ExceptionInutile` affiche "Je vous avais dit de ne pas invoquer cette fonction !". Vous testerez la méthode `nePasInvoquer()` dans le `main` de `ClasseInutile`.

Exercice 3 Rationnel

Reprennez le code de la classe Rationnel, modifiez-le de sorte que si l'on tente de mettre un 0 au dénominateur, une exception soit levée.

```
package encapsulation.corriges;

public class Rationnel
{
	private int num, den;

	/*------------------------------------------------------*/

	public Rationnel(int num, int den)
	{
		this.num = num;
		this.den = den;
	}

	/*------------------------------------------------------*/

	public int getNum()
	{
		return num;
	}

	/*------------------------------------------------------*/

	public int getDen()
	{
		return den;
	}

	/*------------------------------------------------------*/

	public void setNum(int num)
	{
		this.num = num;
	}

	/*------------------------------------------------------*/

	public void setDen(int den)
	{
		if (den != 0)
			this.den = den;
		else
```

```java
                System.out.println("Division by Zero !!!");
}

/*----------------------------------------------------------*/

public String toString()
{
        return getNum() + "/" + getDen();
}

/*----------------------------------------------------------*/

public Rationnel copy()
{
        Rationnel r = new Rationnel(getNum(), getDen());
        return r;
}

/*----------------------------------------------------------*/

public Rationnel opposite()
{
        Rationnel r = copy();
        r.setNum(-r.getNum());
        return r;
}

/*----------------------------------------------------------*/

private static int pgcd(int a, int b)
{
        if (b == 0)
                return a;
        return pgcd(b, a % b);
}

public void reduce()
{
        int p = pgcd(getNum(), getDen());
        setNum(getNum() / p);
        setDen(getDen() / p);
}

/*----------------------------------------------------------*/

public boolean isPositive()
{
        return num > 0 && den > 0 || num < 0 && den < 0;
}

/*----------------------------------------------------------*/

public Rationnel add(Rationnel other)
{
        Rationnel res = new Rationnel(getNum() * other.getDen() + getDen()
                * other.getNum(), getDen() * other.getDen());
        other.reduce();
        return res;
}
```

```java
/*-----------------------------------------------------------*/

public void addBis(Rationnel other)
{
        num = num * other.getDen() + den * other.getNum();
        den = den * other.getDen();
        reduce();
}

/*-----------------------------------------------------------*/

public Rationnel multiply(Rationnel other)
{
        Rationnel res = new Rationnel(getNum() * other.getNum(), getDen()
                        * other.getDen());
        other.reduce();
        return res;
}

/*-----------------------------------------------------------*/

public Rationnel divide(Rationnel other)
{
        Rationnel res = new Rationnel(getNum() * other.getDen(), getDen()
                        * other.getNum());
        other.reduce();
        return res;
}

/*-----------------------------------------------------------*/

public int compareTo(Rationnel other)
{
        Rationnel sub = add(other.opposite());
        if (sub.isPositive())
                return 1;
        if (sub.opposite().isPositive())
                return -1;
        return 0;
}

/*-----------------------------------------------------------*/

public static void main(String[] args)
{
        Rationnel a, b;
        a = new Rationnel(1, 2);
        b = new Rationnel(3, 4);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("compareTo(" + a + ", " + b + ") = "
                        + a.compareTo(b));
        System.out.println(a.copy());
        System.out.println(a.opposite());
        System.out.println(a.add(b));
        System.out.println(a.multiply(b));
        System.out.println(a.divide(b));
}
```

```
}
```

Exercice 4 Pile

Reprennez le code de la pile, modifiez-le de sorte que si l'on tente de dépiler une pile vide, une exception soit levée.

2.11 Interfaces graphiques

2.11.1 Prise en main

Exercice 1 Gestionnaire de disque dur

Créer une fenêtre "Gestionnaire de disque dur". Ajouter un bouton ayant pour intitulé "Formater le disque dur".

Exercice 2 Ecouteur d'événement

Ajoutez un écouteur d'événements à la fenêtre de l'exercice précédent. La pression sur le bouton affichera dans la console le message "formatage en cours". Vous utiliserez les trois méthodes suivantes :

1. La même que dans le premier exemple du cours : implémentation de l'écouteur d'événement dans la même classe que la `JFrame`.
2. Avec une classe anonyme.
3. Avec une classe non anonyme.

Exercice 3 Gestionnaires de mises en forme

Modifier le programme de l'exercice précédent en utilisant un `GridLayout`. Vous afficherez le message "formatage en cours" non pas sur la console mais dans une zone de texte.

2.11.2 Maintenant débrouillez-vous

Exercice 4 Convertisseur

Coder un convertisseur euro/dollar.

Exercice 5 Calculatrice

Coder une calculatrice.

2.12 Tests unitaires

2.12.1 Prise en main

Exercice 1 Test

Mettez en place une batterie de tests unitaires pour les interfaces suivantes :

```
package testsUnitaires.tp;

/**
 *
 * Spécifie un point dans le plan (on remarquera qu'un point est aussi un
 * vecteur).
 *
 */

public interface InterfacePoint
{
        public int getOrd();

        public int getAbs();

        public void setOrd(int ord);

        public void setAbs(int abs);

        /**
         * Retourne la somme des deux points.
         */

        public InterfacePoint add(InterfacePoint p);

        public boolean equals(InterfacePoint p);
}
```

```
package testsUnitaires.tp;

/**
 * Représente un object mobile dans le plan.
 */

public interface InterfaceCursor
{
        /**
         * Retourne la position de l'objet.
         */
        public InterfacePoint getPosition();

        /**
         * Retourne la direction vers laquelle l'objet est tourné.
         */
        public InterfacePoint getDirection();

        /**
         * Restaure la position à 0 et la direction à (1, 0)
         */

        /**
         * Détermine la position de l'objet.
```

```java
         */
        public void setPosition(InterfacePoint position);

        /**
         * Détermine la direction vers laquelle l'objet est tourné.
         */
        public void setDirection(InterfacePoint direction);

        /**
         * Restaure la position à 0 et la direction à (1, 0)
         */
        public void reset();

        /**
         * Fait avancer le mobile d'un pas dans la direction dans laquelle il est
         * tourné.
         */
        public void stepStraigth();

        /**
         * Fait pivoter l'objet d'un quart de tour vers la gauche.
         */
        public void turnLeft();

        /**
         * Fait pivoter l'objet d'un quart de tour vers la droite.
         */
        public void turnRight();
}
```

Exercice 2 Implémentation

Implémentez ces deux interfaces.

2.12.2 Pour aller plus loin

Exercice 3 Echanges de tests

Échangez vos classes de test avec celles d'un autre étudiant, voire avec celles du corrigé. Est-ce que ces tests fonctionnent ?

Exercice 4 Echanges de points

Échangez vos implémentations de la classe `InterfacePoint`. Est-ce que les tests fonctionnent encore ? Est-ce que vos tests fonctionnent avec les implémentations du corrigé ?

2.13 Collections

2.13.1 Types paramétrés

Exercice 1 Paire

Créez une classe Paire<T> contenant deux objets de même type T et deux getters.

Exercice 2 Héritage

Vérifiez expérimentalement si ClasseParametree<String> hérite de ClasseParametree<Object>. Trouvez une explication...

Exercice 3 Pile

Reprenez la classe Pile et adaptez-le pour que la pile, implémentée avec des listes chaînées devienne un type paramétré.
Télécharger la pile

Exercice 4 Paire Ordonnée

Créez une classe PaireOrdonnee<T **extends** Comparable<T>> **extends** Paire<T> contenant deux méthodes getPetit() et getGrand() retournant respectivement le plus petit des deux objets et le plus grand.

Exercice 5 Paire Ordonnée Comparable

Créez une classe PaireOrdonneeComparable<T **extends** Comparable<T>> **extends** PaireOrdonnee<T> **implements** Comparable <PaireOrdonnee<T>>. Vous comparerez deux paires ordonnées en comparant leurs plus grandes valeurs.

2.13.2 Collections

Exercice 6 Pile Iterable<T>

Modifiez la Pile pour qu'elle implémente Iterable<T>. Adaptez la fonction toString() en conséquence.

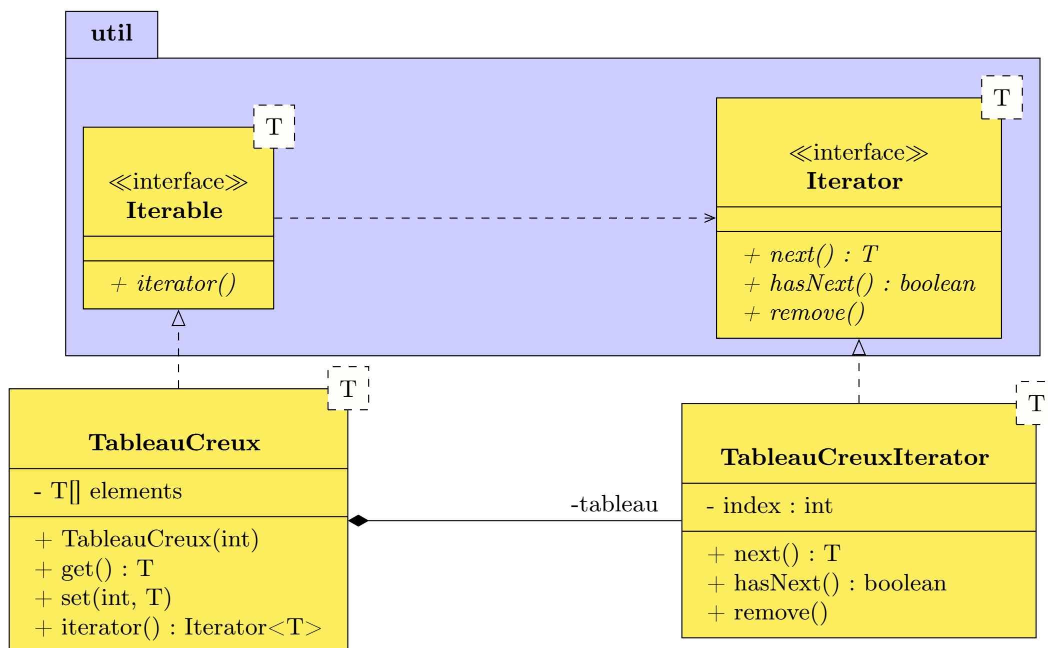
Exercice 7 Parcours

Créez un TreeSet<T>, placez-y des PaireOrdonneeComparable<Integer>, et parcourez-le en affichant ses valeurs.

2.13.3 Morceaux choisis

Exercice 8 Tableaux creux

Implémentez l'interface Tableau creux, l'itérateur implémenté ne devra passer que par les éléments non nuls du tableau. Votre itérateur retournera tous les éléments non **null** du tableau.



Dans une classe paramétrée en java, on ne peut pas instancier un tableau T[], vous êtes obligé de remplacer le tableau par une collection.

```
package collections.interfaces;

public interface TableauCreux<T> extends Iterable<T>
{
        public T get(int i);

        public void set(int i, T item);
}
```

Exercice 9 `Map`

Implémentez l'interface de l'exercice précédent avec une `Map`.

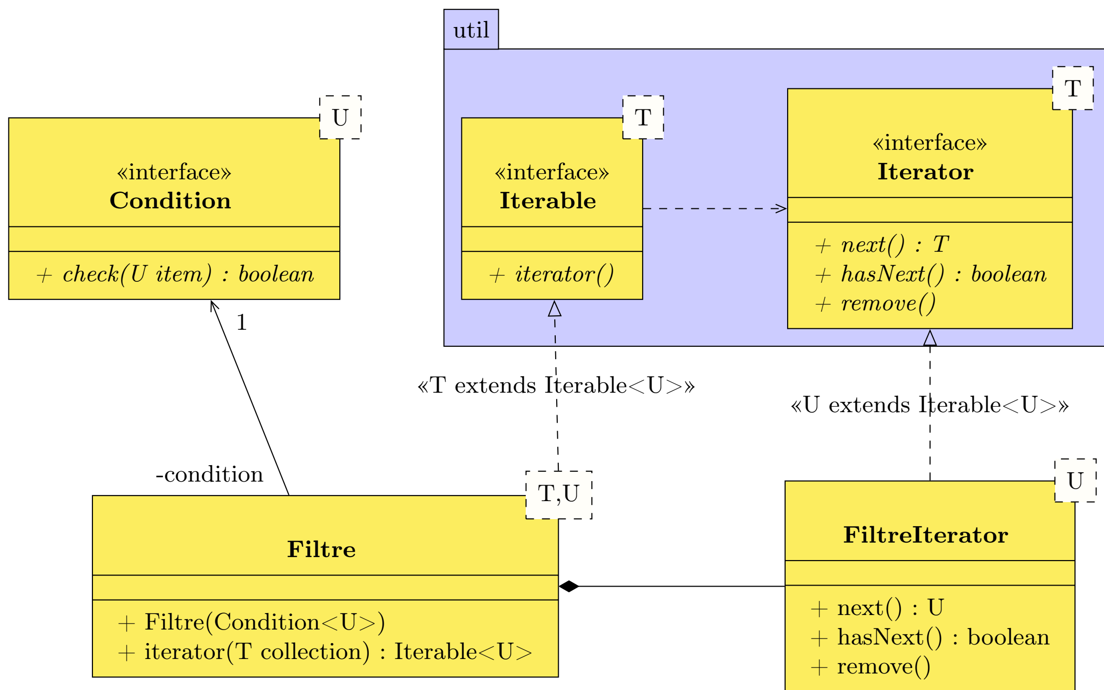
Exercice 10 Matrice

Créez une classe itérable permettant de représenter une matrice.

```
package collections.interfaces;

public interface Matrice<T> extends Iterable<T>
{
        public T get(int i, int j);

        public void set(int i, int j, T value);
}
```

Exercice 11 Filtre

Créez une interface `Condition<U>` contenant une méthode **boolean** `check(U item)`. Créez une classe filtre `Filtre<T, U>` contenant un `Condition<U>` et permettant de filtrer une sous-classe `T` de `Iterable<U>`. Créez une méthode `filtre(Collection<T> condition)` retournant un itérable contenant tous les éléments de la `collection` qui vérifient `check`.


util
U
«interface»
Condition
+ check(U item) : boolean
T
«interface»
Iterable
+ iterator()
T
«interface»
Iterator
+ next() : T
+ hasNext() : boolean
+ remove()
1
«T extends Iterable<U>»
«U extends Iterable<U>»
-condition
T,U
Filtre
+ Filtre(Condition<U>)
+ iterator(T collection) : Iterable<U>
U
FiltreIterator
+ next() : U
+ hasNext() : boolean
+ remove()

2.14 Threads

2.14.1 Prise en main

Exercice 1 La méthode `start()`

Remplacez les appels à `start`() par `run`(), que remarquez-vous ?

Exercice 2 `Runnable`

Reprenez le code de `BinaireAleatoire` et adaptez-le pour utiliser l'interface `Runnable`.

2.14.2 Synchronisation

Exercice 3 Méthode synchronisée

Créez un compteur commun aux deux threads `un` et `zero`, vous y placerez une fonction **synchronized boolean** `stepCounter`() qui retournera vrai tant que le compteur n'aura pas atteint sa limite, et qui retournera tout le temps faux ensuite. Le but étant d'afficher 30000 chiffres, peu importe que ce soit des 1 ou des 0.

Exercice 4 Instructions synchronisées

Reprennez l'exercice précédent en synchronisant les appels à `stepCounter`() dans la méthode `run`().

2.14.3 Débrouillez-vous

Exercice 5 Producteur consommateur

Implémentez une file permettant des accès synchronisés. Créez une classe écrivant des valeurs aléatoires dans la file et une classe lisant ces valeurs et les affichant au fur et à mesure. Lancez plusiseurs instances de la classe de lecture et plusieurs instances de la classe d'écriture sur la même file.

Exercice 6 Diner des philosophes

Implémentez le problème du diner des philosophes. Faites en sorte (dans la mesure du possible) qu'il n'y ait ni famine ni interblocage.

2.15 Persistance

2.15.1 Remember my name

Écrire un programme qui, lors de sa première exécution, demande à l'utilisateur son nom. Le programme devra par la suite se souvenir de ce nom et afficher un message d'accueil personnalisé à chaque exécution. Vous devrez mettre en place les méthodes de persistance demandées dans les exercices suivants.

Exercice 1 Fichier

Écrivez le nom dans un fichier pour le rendre persistant.

Exercice 2 Serialization

Utilisez la serialization.

Exercice 3 JDBC

Utilisez une base de données.

Exercice 4 Requête préparée

Utilisez une requête préparée.

2.15.2 Hachage

Reprenez l'exemple du cours sur les requêtes préparées et faites passer les mots de passe dans une fonction de hachage.

2.15.3 Morceaux choisis

Exercice 5 Copie

Écrivez un programme permettant de recopier un fichier.

Exercice 6 Contacts

Écrire un programme (en ligne de commande) permettant de gérer une liste d'adresses électroniques.

Exercice 7 Contacts JDBC

Idem en version multi-utilisateurs.

2.16 Hibernate

2.16.1 Prise en main

Exercice 1 Remember my name

Reprendre le tp sur la persistence et utilisez Hibernate pour stocker le nom de l'utilisateur dans une base de données. Le programme doit demander à l'utilisateur son nom lors de sa première connexion, et l'afficher à toute autre connexion.

2.16.2 Contacts

Reprendre le gestionnaire de contacts du tp précédent et mettre en place hibernate pour rendre les données persistantes.