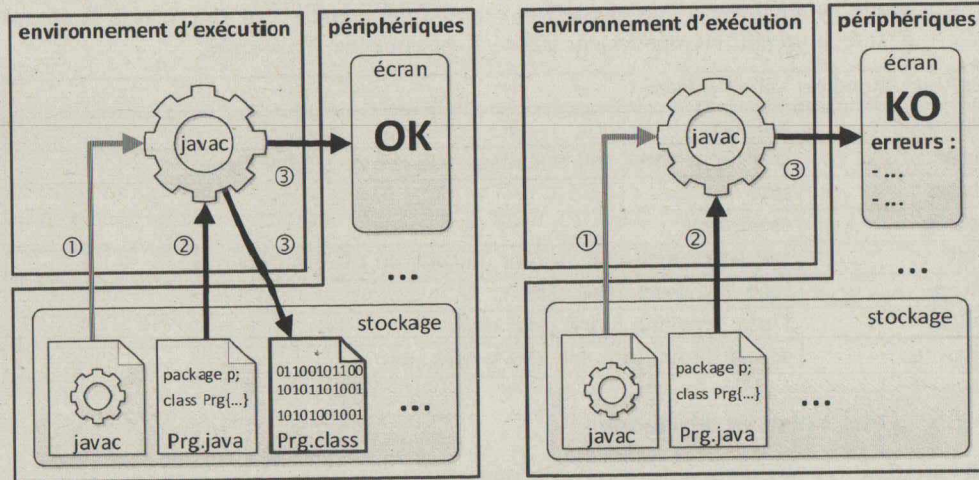


Notes de cours, exercices

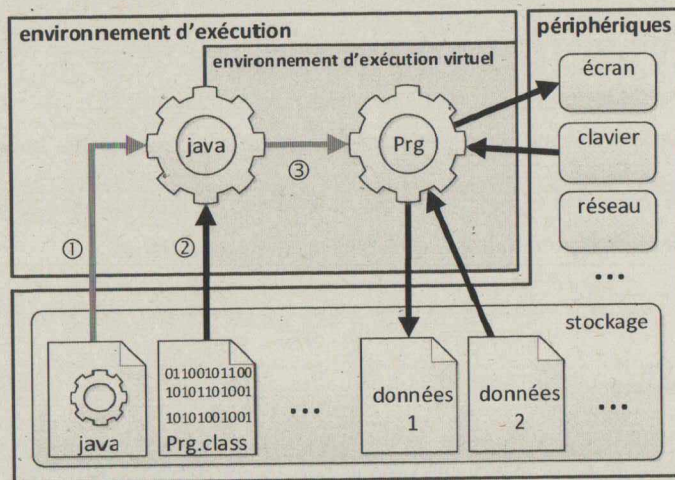
Table des matières

I	Notes de cours	3
1	Fondements du langage Java	4
1.1	Origines et outillage	4
1.2	Constructions élémentaires	6
1.3	Organisation du code et API	14
2	Classes et objets	18
2.1	Principes élémentaires	18
2.2	Classes et instances	20
2.3	Encapsulation et associations	29
3	Héritage et polymorphisme	35
3.1	Principe de substitution	35
3.2	Classes dérivées	37
II	Travaux dirigés	44
1	Fondements impératifs de Java	45
2	Classes, attributs, méthodes et constructeurs	49
3	Classes dérivées et visibilité	52
III	Travaux pratiques	55
1	Manipulation de tableaux	57
2	Classes et modèles d'objets	60
3	Classes, héritage et redéfinitions	64
IV	Java Cheat Sheet	70



1. exécution du compilateur (javac) avec en paramètre le nom du fichier Prg.java
2. chargement et compilation du fichier de code source Prg.java
3. production du bytecode Prg.class (à gauche), ou messages d'erreurs (à droite)

Machine virtuelle Le *bytecode* est chargé en mémoire et exécuté par une *machine virtuelle* (JVM) qui simule le fonctionnement d'une machine réelle et qui dispose de son propre espace mémoire. Illustration :



1. exécution de la JVM (java) avec en paramètre le nom du programme à charger (Prg)
2. chargement du bytecode Prg.class
3. exécution dans l'environnement virtuel du programme Prg

1.2 Constructions élémentaires

1.2.1 Premier aperçu du langage

En Java, on code des *classes*. Une classe porte un nom et correspond soit à un *modèle d'objets* (notion détaillée dans le chapitre suivant), soit à un simple module qui rassemble des variables et des fonctions. À quelques nuances près, *il n'y a pas de code en dehors des classes* :

```
class MaPremiereClasse { // classe correspondant à un modèle d'objets
    // ce qu'on peut trouver ici sera dévoilé au chapitre 2...
}

class MonPremierProgramme { // classe correspondant à un simple module de code
    static int x = 18; // variable x, de type entier, qui stocke la valeur 18

    // Voici une fonction à deux paramètres entier a et b, qui renvoie un entier :
    static int add(int a, int b) {
        int res = a + b; // res est une variable locale qui stocke la somme de a et b
        return res;      // instruction renvoyant le contenu de res en résultat.
    }

    // Voici la fonction principale, le point de départ de l'exécution du programme :
    public static void main(String[] args) {
        int s = add(x, 2); // variable locale qui stocke le résultat de l'appel de add
        System.out.println("somme : " + s); // affichage de "somme : 20" sur la console
    }
}
```

1.2.2 Commentaires et annotations

Les commentaires et annotations sont dans le code source mais ne font pas partie du programme.

Ligne Un commentaire sur une ligne commence par `//` et termine à la fin de la ligne.

```
int x = 0; // ceci est un commentaire qui se termine à la fin de la ligne...
```

Bloc Un commentaire sur plusieurs lignes commence par `/*` et termine par `*/`.

```
/* Ce commentaire est long et peut comporter plusieurs lignes... */
int x = /* commentaire au milieu d'une instruction ! */ 0;
```

Javadoc Un commentaire destiné à javadoc commence par `/**` et termine par `*/`.

```
/** Ce commentaire long est destiné à javadoc. Il documente la classe A suivante... */
class A { /* ... */ }
```

Annotation Instruction accolée à un élément du programme (classe, variable, instruction, etc.), typiquement destinée au compilateur. Par exemple, demande de suppression d'avertissements :

```
@SuppressWarnings("unchecked")
class A { /* ... */ }
```


Opérateurs sur les nombres S'appliquent à `byte`, `short`, `int`, `long`, `float`, `double` :

Opérateurs arithmétiques usuels	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code>
Opérateurs bit à bit usuels	<code>~</code> , <code>&</code> , <code> </code> , <code>^</code>
Opérateurs rotation de bits	<code><<</code> , <code>>></code> , <code>>>></code>
Opérateurs de comparaison	<code><</code> , <code><=</code> , <code>></code> , <code>>=</code>
Opérateurs d'incrément et de décrémentation	<code>++</code> , <code>--</code>

Opérateurs sur des objets ou des valeurs primitives Ils s'appliquent indifféremment à des valeurs primitives ou à des *références d'objets* (notion détaillée dans le prochain chapitre) :

Test d'égalité et de différence	<code>==</code> , <code>!=</code>
Coercition de type (<i>cast</i>)	<code>(.)</code>

Une valeur primitive numérique peut être *castée* en un type primitif numérique différent auquel cas la donnée initiale peut être modifiée :

```
int b = 130;
byte o = (byte)b; // o : -126 (correspond à (130+128)%256 - 128)
float f = 3.56;
int i = (int)f; // i : 3 (correspond à la troncature à l'unité de 3.56)
```

Opérateurs spéciaux Opérateurs hors catégories :

Concaténation de chaînes de caractères	<code>+</code>
Affectations	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code>
Instanciation	<code>new</code>
Accès tableau	<code>.</code> <code>[.]</code>

Les chaînes de caractères sont des *objets* de type `String` et peuvent être désignés par des *littéraux spécifiques* comme `"abc"`. L'opérateur `+` correspond à la concaténation de chaînes *seulement si* au moins une opérande est de type `String`. Si l'autre opérande est une valeur primitive, c'est sa représentation littérale est utilisée en guise de chaîne de caractères :

```
"abc" + "def" // valeur de l'expression : "abcdef"
"2" + 2 // valeur de l'expression : "22"
```

Priorité des opérateurs Les expressions peuvent être *parenthésées*. Les opérateurs ont des *niveaux de priorité* d'évaluation, qui respectent en particulier la priorité des opérateurs arithmétiques. Exemple :

```
int n = 2 + 3 * 5; // ordre d'évaluation des opérateurs : *, +, =
int n = (2 + 3) * 5; // ordre d'évaluation des opérateurs : +, *, =
```

Si l'expression comporte des opérateurs de même priorité, ils sont évalués de gauche à droite :

```
int n = 2 + 3 - 5; // ordre d'évaluation des opérateurs : +, -, =
```

1.2.7 Structures de contrôle

En Java, les instructions s'enchaînent par défaut et implicitement en *séquence*. Les *structures de contrôle* sont les instructions qui modifient cet enchaînement.

Itérations L'itération fondamentale est la boucle *while*. Elle consiste à répéter l'exécution d'un bloc de code tant qu'une condition (appelée *invariant de boucle*) est vraie. Cette condition est (ré)évaluée avant chaque (ré)exécution du bloc. La négation de cet invariant correspond à la condition de sortie. Exemples :

```
while (x < 2) { /* code exécuté tant que x < 2. */ } // condition de sortie : x >= 2
while (true) { /* code perpétuellement exécuté */ } // condition de sortie : false
```

Variante *do ... while* : le corps de la boucle est exécuté une première fois avant la première évaluation de l'invariant. Illustration :

```
int i;
do { // code exécuté au moins une fois
    i = /* lecture d'un entier sensé être positif */ ;
} while (i < 0); // si i est négatif, on recommence la lecture de l'entier...
```

Variante *for* : intègre des clauses (optionnelles, simples ou multiples) d'initialisation et de mise à jour en fin de boucle. Cette boucle est typiquement utilisée dans le cas où le nombre d'itérations est prédéterminé :

```
for (int i = 0; i < 10; i++) { /* code exécuté 10 fois */ }
int x, y;
for (x = 0, y = 10; x < y; x++, y--) { /* code exécuté 5 fois */ }
```

Variante de type *for each* : dédiée au parcours séquentiel en lecture seule des éléments d'une *collection* ou d'un tableau. Ce type de boucle est présenté plus loin avec les tableaux.

Conditionnelles La conditionnelle simple utilise un booléen pour exécuter une certaine portion de code (clause *alors* obligatoire) ou une autre (clause *sinon* optionnelle) :

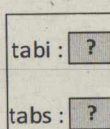
```
if (a < 0) { a = b; } else { a = c; } // variante : if (a < 0) a = b; else a = c;
if (a < 0) { a = b; b = c; } else { } // variante : if (a < 0) { a = b; b = c; }
if (a < 0) { } else { a = b; b = c; } // variante : if (a >= 0) { a = b; b = c; }
```

Le sélecteur (ou *switch*) utilise une valeur énumérée pour sélectionner une portion de code à exécuter. Cette valeur peut être notamment un entier, un caractère ou une chaîne de caractères. Exemple :

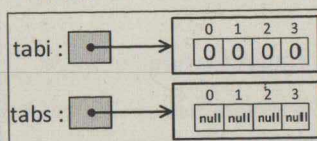
```
int direction;
/* ... */
String directionString;
switch (direction) {
    case 0: directionString = "Nord"; break;
    case 1: directionString = "Est"; break;
    case 2: directionString = "Sud"; break;
    case 3: directionString = "Ouest"; break;
    default: directionString = "???";
}
```



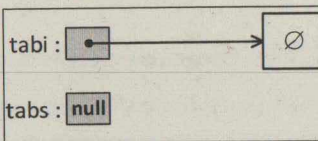
```
int[] tabi;
String[] tabs;
```



```
int[] tabi = new int[4];
String[] tabs = new String[4];
```



```
int[] tabi = new int[0];
String[] tabs = null;
```



Un tableau peut être alloué et initialisé directement grâce aux raccourcis suivants :

```
int[] tabi = {8, -1, 3, 0}; // déclaration + création + initialisation
tabi = new int[]{7, 6, 25}; // création avec new + initialisation
```

Un tableau peut être parcouru grâce à un indice variant de 0 (inclus) à la taille du tableau (exclue) fournie par `length` :

```
String[] tab = new String[4]; // tableau de 4 cases
for (int i = 0; i < tab.length; i++) // for classique : i prend tous les indices de tab
    tab[i] = "abc"; // cases mises à jour : à la fin, elles désignent toutes "abc"
```

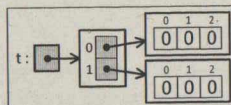
La boucle de type *for each* permet le parcours d'un tableau grâce à une variable qui prend successivement toutes les valeurs de ce tableau (parcours en *lecture seule*) :

```
String[] tab = {"ab", "cd", "ef"}; // tableau de 3 cases initialisées
String r = ""; // r est une chaîne vide
for (String s : tab) // for each : s prend toutes les valeurs de tab
    r += s; // concaténation de toutes les cases de tab dans r
```

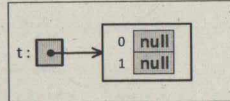
Tableau à deux dimensions Un tableau à 2 dimensions est un tableau à 1 dimension de tableaux à 1 dimension (*tableau de lignes*). Son type est défini par le *type* des cases suivi de deux paires de crochets. Sa création nécessite de connaître *au minimum* le nombre de lignes à allouer. Il est possible de préciser en plus la taille de chaque ligne :

```
int[][] t = new int[2][3];
```

Représentation schématique du tableau



```
int[][] t = new int[2]{};
```



Raccourcis de création :

```
int[][] tabi = { {8, -1, 3, 0}, {1, -1, 8, 5} }; // à la déclaration
tabi = new int[][]{ {8, -1}, {1, -1}, {8, 5} }; // hors déclaration
```

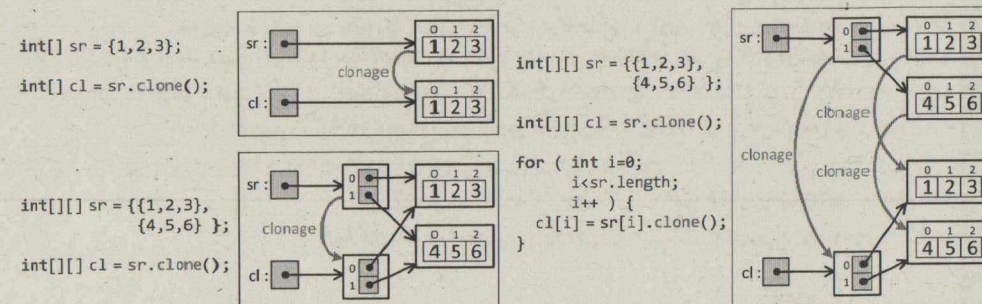
Le parcours d'un tableau à 2 dimensions correspond typiquement à un parcours des cases (par une boucle *for*) de chaque *ligne* (par une boucle *for* englobante) :

```
String r = "";
for (int l = 0; l < tab.length; l++) // indice des lignes
    for (int c = 0; c < tab[l].length; c++) // indice des colonnes (des cases d'une ligne)
        r += tab[l][c];
```

Variante avec une boucle de type *for each* :

```
for (String[] ligne : tab) // valeur des lignes
    for (String valeur : ligne) // valeur des cases
        r += valeur;
```

Clonage de tableaux Les tableaux sont des objets *clonables*. Ils disposent d'une fonction appelée `clone` qui fournit une *copie superficielle*. Un tableau *t* et son clone *c* partagent donc les mêmes références d'objets. C'est en particulier le cas d'un tableau à deux dimensions. Il peut donc être utile voire nécessaire dans ce cas de cloner également les lignes. Illustration :



Fonctions de manipulation Les classes utilitaires `System` et `Arrays` fournissent des fonctions dédiées à la manipulation des tableaux. Exemples :

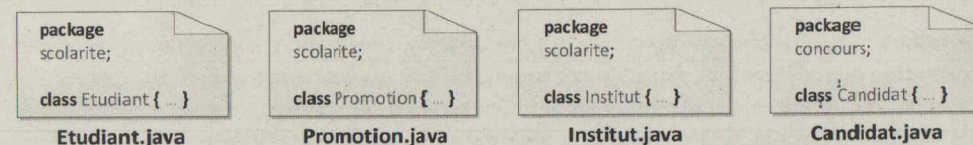
```
/* Dans la classe System */
static void arraycopy(Object s, int i, Object d, int j, int l) // s et d sont des tableaux !
// cette fonction copie l cases de s à partir de l'indice i, dans d à partir de l'indice j
```

```
/* Dans la classe Arrays */
static int[] copyOf(int[] t, int newLength) // copie de t avec plus ou moins de cases
static int[] copyOfRange(int[] t, int i, int j) // extrait de t de i inclus à j exclu
static boolean equals(int[] a1, int[] a2) // test d'égalité de deux tableaux à 1 dimension
static boolean deepEquals(Object[] a1, Object[] a2) // comme equals, avec 2 dimensions
static String toString(int[] a) // représentation littérale (textuelle) de a
static String deepToString(Object[] a) // comme toString, avec 2 dimensions
```

1.3 Organisation du code et API

1.3.1 Organisation du code source

Code source Un fichier de code source porte l'extension `.java` et comporte généralement une directive `package`, des directives `import` et des *déclarations* de classes (ou autres éléments assimilés non détaillés ici). En théorie, plusieurs classes peuvent être déclarées dans un même fichier source dès lors qu'elles appartiennent au même package. En pratique, il est d'usage de définir chacune d'elles dans un fichier dédié portant son nom :




```
Boolean choix = new Boolean(true); // objet Boolean créé à partir de la valeur true
Boolean bo = Boolean.valueOf(true); // objet Boolean créé à partir de la valeur true
boolean b1 = bo.booleanValue(); // boolean obtenu à partir d'un objet Boolean
boolean b2 = Boolean.parseBoolean("true"); // boolean obtenu à partir de sa représentation
```

Les types numériques sont représentés par des classes qui *héritent toutes* de la classe `Number`. Elles disposent donc toutes des fonctionnalités de conversion suivantes :

```
byte   byteValue() // conversion vers byte
double doubleValue() // conversion vers double
float  floatValue() // conversion vers float
int     intValue()   // conversion vers int
long    longValue()  // conversion vers long
short   shortValue() // conversion vers short
```

Affichage Pour afficher un message sur la sortie standard, il suffit d'invoquer les méthodes `System.out.print` ou `System.out.println` avec en paramètre ce qui doit être affiché, et qui peut concrètement être une valeur primitive, une chaîne de caractères ou un objet (grâce à sa représentation littérale obtenue implicitement par `toString`). La version `println` rajoute un retour à la ligne. Illustration :

```
int[] tab = {5, -2, 8};
System.out.println("Tableau de_" + tab.length + "_éléments_" + Arrays.toString(tab));
```

À l'exécution, le message suivant s'affiche sur la sortie standard :

```
Tableau de 3 éléments : [5, -2, 8]
```

Constantes et fonctions mathématiques La classe `Math` est une classe utilitaire, i.e. qui ne représente pas un modèle d'objets, et qui rassemble des constantes et des fonctions utiles :

```
static double E; // approximation de la constante d'Euler
static double Pi; // approximation de Pi
static double abs(int a) // valeur absolue
static double sqrt(double a) // racine carrée
static double cbrt(double a) // racine cubique
static double cos(double a) // cosinus
static double log(double a) // logarithme népérien (ln)
static double log10(double a) // logarithme décimal
static double max(double a, double b) // plus grand élément
static double min(double a, double b) // plus petit élément
static double pow(double a, double b) // puissance
static double random() // valeur aléatoire dans [0 - 1[
```

La fonction `random` peut être utilisée pour tirer au hasard une valeur entière dans un intervalle de valeurs possibles. Il suffit de multiplier son résultat par le nombre de valeurs de l'intervalle, de rajouter la valeur de la borne inférieure de l'intervalle et de caster le résultat en entier. Illustration avec un nombre aléatoire compris entre 5 et 20 :

```
double r = Math.random(); // r : [0 - 1[
r = r * (20 - 5 + 1); // r : [0 - 16[
r = r + 5; // r : [5 - 21[
int n = (int)r; // n : [5, 6, ... , 20]
```

Chapitre 2

Classes et objets

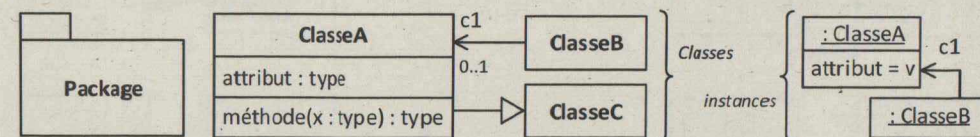
Extraits de la partie III du manuel, page 155

2.1	Principes élémentaires	18
2.2	Classes et instances	20
2.3	Encapsulation et associations	29

2.1 Principes élémentaires

2.1.1 Préambule : langage UML

UML est un langage graphique de modélisation qui permet en particulier de représenter un programme, i.e. de décrire de manière abstraite ses éléments et sa structure. La figure suivante illustre les concepts d'UML utilisés pour ce cours : *objet (instance)*, *classe*, *attribut*, *méthode*, *association*, *héritage* et *package* :



2.1.2 Objets

Définition Un objet est une entité qui rassemble des données, et avec laquelle des messages peuvent être échangés. Un objet est créé, existe, puis disparaît au cours de l'exécution d'un programme. Au cours de son existence, un objet interagit avec d'autres objets en échangeant des messages.

Identité, encapsulation de données et état Un objet a une identité unique qui le distingue de tous les autres objets. Il porte en lui des données. Il est possible de consulter ou de mettre à jour ces données seulement si l'objet a été conçu pour répondre à ces requêtes. Ce principe d'enfouissement des données s'appelle *encapsulation*. À un instant donné de la vie d'un objet, l'ensemble des données qu'il porte constitue son *état*.

Instanciation Pour créer un objet à partir d'une classe, il faut utiliser l'opérateur `new` comme dans l'exemple suivant :

```
new Couleur() // objet créé à partir de la classe Couleur
```

Cette opération est l'*instanciation*. On appelle *instance* l'objet ainsi obtenu. Cette instance a pour type la classe dont elle est issue. Elle peut donc être utilisée partout où un objet de ce type-là est attendu. Illustration :

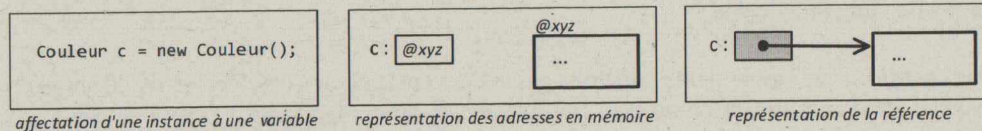
```
Couleur c = new Couleur(); // affectation d'une instance à une variable
Couleur[] tab = {c, new Couleur()}; // tableau contenant 2 couleurs
```

```
static Couleur melange(Couleur[] couleurs) {
    Couleur resultat = new Couleur(); // affectation d'une instance à une variable
    /* ... */
    return resultat;
}
```

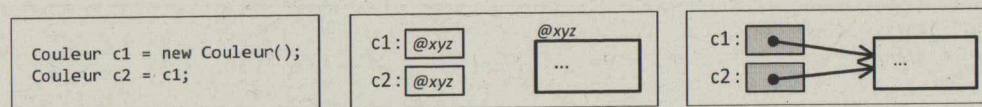
```
Couleur m = melange(tab); // affectation de l'instance résultant de l'appel à une variable
```

2.2.2 Référence d'objet

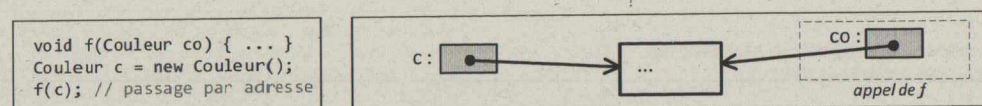
Adresse mémoire Un objet est manipulé par *référence*. Ainsi, une variable typée par une classe n'est pas supposée contenir une instance de cette classe, mais l'*adresse mémoire* d'une instance de cette classe. Lorsque l'opérateur `new` crée une instance, il commence par allouer l'espace mémoire nécessaire au stockage de l'objet créé. C'est l'adresse de cet espace que `new` renvoie, et qui peut ainsi se retrouver stocké dans une variable grâce à l'affectation. Illustration :



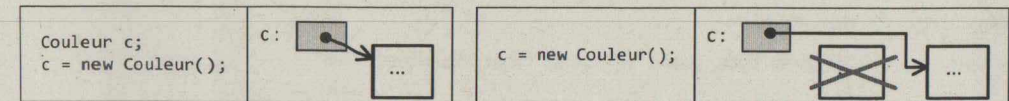
Partage Lorsque plusieurs variables stockent la même adresse d'objet, cet objet est virtuellement *partagé* par toutes ces variables :



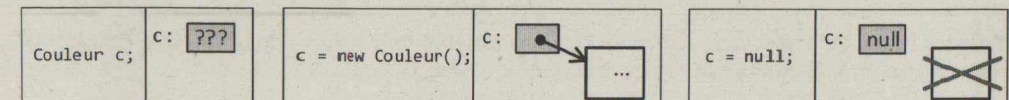
Passage de paramètre par adresse Ce principe de partage s'applique aux appels de méthodes qui comportent des paramètres typés par des classes. Le passage de paramètres dans ce cas est un *passage de paramètre par adresse*. Illustration :



Déréférencement Lorsqu'un objet n'est plus référencé dans aucune variable du programme, le *garbage collector* le libère. Cette libération est automatique et n'a donc pas à être programmée explicitement (comme en C par exemple) :



Littéral null Il existe une valeur d'adresse spéciale désignée par `null` qui ne référence aucun objet. Cette valeur peut être utilisée partout où un objet est attendu. Illustration :



2.2.3 Attribut

Déclaration Un attribut représente une partie des données qu'une instance de la classe stockera. On le déclare à la manière d'une simple variable dans le corps de la classe :

```
class Couleur {
    int r; // attribut r, représente une valeur entière
}
```

État L'ensemble des attributs déclarés constitue la structure interne des instances qui seront créées à partir de la classe. Cette structure portera l'état de l'objet au cours de sa vie. Elle doit être cohérente et doit permettre de répondre à la question suivante : *comment le concept qui m'intéresse est-il représenté par les objets de cette classe ?*

Dans l'exemple suivant, la classe représente le concept de *couleur*. Les attributs déclarés permettent de répondre à la question : *une couleur est représentée par la combinaison des trois composantes rouge, verte et bleue de la lumière; chaque composante est un entier allant de 0 (couleur absente) à 255 (intensité de couleur maximale) :*

```
class Couleur {
    int r; // composante rouge de la couleur
    int v; // composante verte de la couleur
    int b; // composante bleue de la couleur
}
```

Accès aux attributs On accède aux attributs d'un objet en utilisant l'opérateur *point* (`.`) :

```
Couleur c = new Couleur();
// mise à jour des attributs de c :
c.r = 61;
c.v = 194;
c.b = 93
```


Appel de constructeur dans un constructeur Il est possible dans un constructeur d'invoquer un *autre* constructeur de la classe en le désignant par `this` suivi d'une paire de parenthèses éventuellement vide. Cette invocation ne peut apparaître *qu'en première ligne* d'un constructeur :

```
class Couleur {
    int r, v, b; // déclaration combinée des 3 attributs

    Couleur (int r, int v, int b) { // 1er constructeur : 3 paramètres
        this.r = r;
        this.v = v;
        this.b = b;
    }

    Couleur () { // 2ème constructeur : pas de paramètre
        this(255, 255, 255); // appel du premier constructeur (en première ligne : ok)
    }
}
```

2.2.5 Méthode

Déclaration Une méthode représente un message qu'une instance de la classe saura recevoir. On la déclare à la manière d'une fonction dans le corps de la classe :

```
class Couleur {
    // attributs et constructeurs...
    String codeHTML() {
        /* corps de la méthode codeHTML */
        ...
    }
}
```

Invocation On invoque une méthode sur un objet en utilisant l'opérateur *point* (`.`) :

```
Couleur c = new Couleur(61, 194, 93);
String s = c.codeHTML(); // appel de codeHTML sur c
```

Cet appel de méthode correspond bien à la notion de message envoyé à un objet :

- le message porte un nom : `codeHTML`
- il n'est pas accompagné de données (pas de paramètre dans cet exemple)
- il est envoyé à l'objet référencé par `c`, qui peut le recevoir puisque sa classe comporte effectivement une méthode `codeHTML` sans paramètre
- l'objet référencé par `c` *réagit* en exécutant le corps de la méthode
- il est prévu qu'il renvoie une réponse (résultat de type `String`) : c'est bien ce qu'il fait !
- le résultat est récupéré et stocké dans la variable `s`

À un niveau plus abstrait, on voit bien le dialogue qui s'opère ici entre l'objet référencé par `c` et l'objet inconnu qui s'adresse à lui :

- *Bonjour couleur c, pourrais-tu me donner ta valeur en HTML s'il te plait ?*
- *Bien sûr, je peux répondre à ta requête, c'est : "3DC25D"*

Comportement L'ensemble des méthodes déclarées constitue l'interface des instances qui seront créées à partir de la classe. Cette interface est un mode d'emploi pour ces objets. Elle indique comment interagir avec eux. Cette interface et les réactions correspondantes (*i.e.* le corps des méthodes) détermine le comportement des objets.

Dans l'exemple de la classe `Couleur`, les deux méthodes suivantes indiquent la manière d'interagir avec des couleurs. On peut leur demander leur code HTML (méthode `codeHTML`), et on peut leur demander de passer au rouge (méthode `enRouge`) :

```
class Couleur { // attributs et constructeurs...
    String codeHTML() { ... }
    void enRouge () { ... }
}
```

Receveur Le corps d'une méthode spécifie la réaction d'une instance à la réception du message correspondant. Cette instance s'appelle *receveur* et est désignée par `this`. Illustration :

```
class Couleur { // attributs et constructeurs...
    String codeHTML() {
        // String.format("%02X", n) : représentation hexadécimale de n sur 2 chiffres
        String rHex = String.format("%02X", this.r);
        String vHex = String.format("%02X", this.v);
        String bHex = String.format("%02X", this.b);
        return (rHex+vHex+bHex).toUpperCase(); // mise en majuscules des lettres
    }

    void enRouge() {
        this.r = 255;
        this.v = 0;
        this.b = 0;
    }
}
```

Lorsqu'on invoque une méthode, le receveur est l'objet indiqué à *gauche* du point. Il correspond à `this` à l'exécution du code :

```
Couleur c = new Couleur(61, 194, 93);
String s1 = c.codeHTML(); // c prend la place de this ; au final, s1 : "3DC25D"
c.enRouge(); // c prend la place de this ; au final, c.r : 255, c.v : 0, c.b : 0
String s2 = c.codeHTML(); // c prend la place de this ; au final, s2 : "FF0000"
```

Surcharge de méthodes Le corps d'une classe peut comporter plusieurs méthodes portant le même nom, à condition qu'elles diffèrent en nombre et/ou en type de paramètres. Ces méthodes sont dites *surchargées* :

```
class Couleur { // attributs, constructeurs, méthodes...
    void melange(Couleur c) { // mélange les couleurs de this et de c
        this.r = (this.r + c.r) / 2;
        this.v = (this.v + c.v) / 2;
        this.b = (this.b + c.b) / 2;
    }
}
```


2.3 Encapsulation et associations

2.3.1 Principe d'encapsulation

Propriétés fondamentales En principe, on interagit avec un objet exclusivement par l'intermédiaire de son interface. Concrètement, on peut invoquer des méthodes sur un objet, mais on ne doit pas accéder directement à ses attributs. Ce principe dit *d'encapsulation* permet de garantir les propriétés fondamentales suivantes :

- les contraintes d'intégrité sur les attributs sont respectées
- la modification des attributs d'une classe n'impacte pas le reste du programme

Accesseurs et mutateurs Le principe d'encapsulation interdit l'accès direct aux attributs. Si cet accès est nécessaire, il doit être fourni par des méthodes dédiées. On appelle ces méthodes *accesseurs* (ou *getters*) et *mutateurs* (ou *setters*). Un accesseur fournit la valeur d'un attribut. Un mutateur met à jour la valeur d'un attribut. Illustration :

```
class Couleur {
    int r, v, b; // attributs

    // constructeurs et méthodes (codeHTML, enRouge, melange)...

    // Getters :
    int getR() { return this.r; }
    int getV() { return this.v; }
    int getB() { return this.b; }

    // Setters :
    void setR(int r) { this.r = r; }
    void setV(int v) { this.v = v; }
    void setB(int b) { this.b = b; }
}
```

Ces nouvelles méthodes permettent de rajouter une vérification sur les valeurs attendues pour *r*, *v* et *b* et d'assimiler par exemple à 0 une valeur négative et à 255 une valeur supérieure à 255. On peut enfin mettre à jour le premier constructeur afin de bénéficier de ce contrôle :

```
Couleur (int r, int v, int b) { // 1er constructeur
    this.setR(r); // remplace : this.r = r;
    this.setV(v); // remplace : this.v = v;
    this.setB(b); // remplace : this.b = b;
}

void setR(int r) { this.r = (r < 0 ? 0 : (r > 255 ? 255 : r)); }
void setV(int v) { this.v = (v < 0 ? 0 : (v > 255 ? 255 : v)); }
void setB(int b) { this.b = (b < 0 ? 0 : (b > 255 ? 255 : b)); }
```

Ces setters garantissent le respect de la contrainte, contrairement à l'accès direct aux attributs :

```
Couleur c = new Couleur(61, 194, 93);
c.setR(308); // c.r : 255 (ok)
c.r = 308; // c.r : 308 (violation de la contrainte sur r !)
```

Tolérance à la modification de structure Grâce au principe d'encapsulation, on peut considérablement modifier la structure d'une classe sans remettre en cause le reste du programme. Par exemple dans la classe *Couleur*, on décide de remplacer les attributs *r*, *v* et *b* qui représentent des *valeurs* d'intensité par les attributs *tr*, *tv* et *tb* qui représentent des *taux* d'intensité (valeurs flottantes comprises entre 0 et 1). Cette modification nécessite d'adapter toutes les méthodes, et en particulier les accesseurs et les mutateurs des anciens attributs :

```
class Couleur {
    // Attributs :
    double tr, tv, tb; // remplace : int r, v, b;

    // constructeurs, autres méthodes mises à jour...

    // Getters des anciens attributs r, v et b :
    int getR() { return (int)(this.tr * 255); }
    int getV() { return (int)(this.tv * 255); }
    int getB() { return (int)(this.tb * 255); }

    // Setters des anciens attributs r, v et b :
    void setR(int r) { this.tr = (r < 0 ? 0 : (r > 255 ? 1 : r/255.0)); }
    void setV(int v) { this.tv = (v < 0 ? 0 : (v > 255 ? 1 : v/255.0)); }
    void setB(int b) { this.tb = (b < 0 ? 0 : (b > 255 ? 1 : b/255.0)); }
}
```

Cette modification impacte le reste du code partout où on s'est permis d'accéder directement aux attributs (i.e. partout où on s'est permis de ne pas respecter le principe d'encapsulation) :

```
Couleur c = new Couleur(61, 194, 93);
c.r = 137; // ERREUR : l'attribut r n'existe plus !
```

En revanche, cette modification n'impacte pas le reste du code là où on a utilisé les méthodes dédiées à l'accès aux attributs :

```
Couleur c = new Couleur(61, 194, 93);
c.setR(137); // c.tr : 0.54
```

2.3.2 Visibilité des membres

Les accesseurs et les mutateurs permettent de respecter le principe d'encapsulation mais ils n'interdisent pas l'accès direct aux attributs. Pour le faire, il faut définir les bons niveaux de *visibilité*.

La visibilité d'un membre (attribut ou méthode) correspond au niveau d'autorisation d'accès à ce membre selon le contexte de l'accès. Un membre *m* d'une classe *A* d'un package *p* a un niveau d'accès par défaut qui peut être modifié à sa déclaration par *public*, *protected* ou *private* :

membre <i>m</i> de p.A	accès depuis			
	classe p.A	autre classe de p	sous-classe de p.A	partout ailleurs
public <i>m</i>	ok	ok	ok	ok
protected <i>m</i>	ok	ok	ok	-
<i>m</i>	ok	ok	-	-
private <i>m</i>	ok	-	-	-

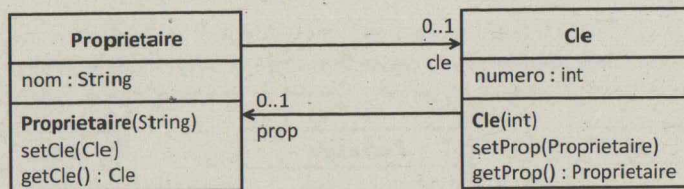
Double canaux unidirectionnels Pour corriger ce problème, on peut commencer par mettre en place une référence réciproque comme dans l'exemple suivant au niveau de la classe Cle :

```
class Cle { // représente une clé
    int numero; // numéro de la clé
    Proprietaire prop; // propriétaire de la clé

    Cle(int n) { // constructeur
        this.numero = n;
        this.prop = null; // initialement, la clé n'a pas de propriétaire
    }

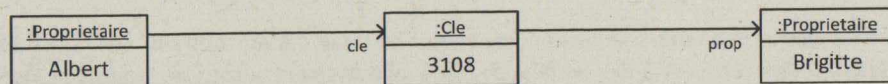
    void setProp(Proprietaire p) { // setter : définit un propriétaire
        this.prop = p;
    }

    Proprietaire getProp() { // getter : récupère le propriétaire de la clé
        return this.prop;
    }
}
```



Ce canal de communication unidirectionnel réciproque n'interdit pas les incohérences comme dans l'exemple suivant :

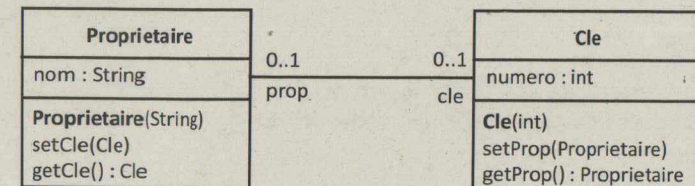
```
Cle c = new Cle(3108);
Proprietaire p1 = new Proprietaire("Albert");
Proprietaire p2 = new Proprietaire("Brigitte");
p1.setCle(c); // Albert possède maintenant la clé 3108...
c.setProp(p2); // mais pour la clé 3108, c'est Brigitte la propriétaire !
```



Canal bidirectionnel La relation entre clé et propriétaire implique une contrainte d'intégrité : si une clé *c* à *p* pour propriétaire, alors *p* a *c* pour clé, et réciproquement. Une association entre deux classes représente un canal de communication bidirectionnel entre les instances de ces classes. Elle garantit par construction ce type de contrainte d'intégrité. En java, une association doit être programmée. Par exemple, les nouvelles versions de setCle et de setProp suivantes garantissent la cohérence du lien bidirectionnel :

```
// Dans la classe Cle :
void setProp(Proprietaire p) { // setter : définit un propriétaire
    if (this.prop != null) // this a déjà un propriétaire
        this.prop.cle = null; // le propriétaire actuel de this perd sa clé (this)
    this.prop = p; // p est le nouveau propriétaire de this
    if (p.cle != null) // p avait déjà une clé
        p.cle.prop = null; // l'ancienne clé de p n'a plus de propriétaire
    p.cle = this; // this est la nouvelle clé de this
}

// Dans la classe Proprietaire :
void setCle(Cle cle) { // setter : donne une clé au propriétaire
    if (this.cle != null) // this a déjà une clé
        this.cle.prop = null; // la clé actuelle de this perd son propriétaire (this)
    this.cle = cle; // cle est la nouvelle clé de this
    if (cle.prop != null) // cle avait déjà un propriétaire
        cle.prop.cle = null; // l'ancien propriétaire de cle de p n'a plus de clé
    cle.prop = this; // this est le nouveau propriétaire de this
}
```



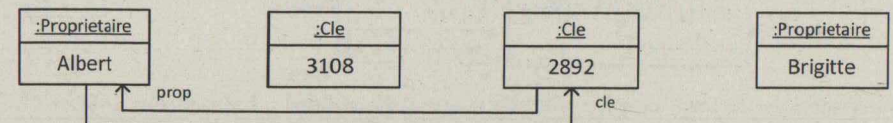
Dans l'exemple suivant, deux propriétaires ont chacun leur propre clé :

```
Cle c1 = new Cle(3108);
Cle c2 = new Cle(2892);
Proprietaire p1 = new Proprietaire("Albert");
Proprietaire p2 = new Proprietaire("Brigitte");
p1.setCle(c1); // Albert possède maintenant la clé 3108
p2.setCle(c2); // Brigitte possède maintenant la clé 2892
```



Un simple appel à setCle (ou de manière symétrique à setProp) permet maintenant d'assigner la clé d'un propriétaire à l'autre en maintenant la cohérence de l'ensemble des objets :

```
p1.setCle(c2); // Albert possède maintenant la clé 2892
// Brigitte n'a plus de clé, et la clé 3108 est libre
```



3.2 Classes dérivées

3.2.1 Héritage

Déclaration La clause `extends` dans l'entête d'une classe indique le nom d'une classe héritée. En Java, il n'est possible d'hériter que d'une seule classe (*héritage simple*). Illustration :

```
class CouleurT extends Couleur {
    /* corps de la classe CouleurT */
    ...
}
```

CouleurT introduit la *transparence par couche alpha* en ajoutant aux trois composantes d'une couleur un niveau de transparence compris entre 0 (opaque) et 255 (invisible).

Nouveaux membres Par définition, une classe qui en hérite d'une autre dispose *de facto* de tous ses membres. Elle peut en déclarer de nouveaux. Par exemple, la classe `CouleurT` hérite des trois attributs `r`, `v` et `b` de `Couleur`, ainsi que des méthodes `codeHTML` et `enRouge`. Elle complète cet héritage avec l'attribut `t` qui représente le niveau de transparence compris entre 0 et 255, et la méthode `setT` qui permet de mettre à jour ce nouvel attribut :

```
class Couleur {
    int r, v, b; // attributs
    // méthodes :
    String codeHTML() { ... }
    void enRouge () { ... }
}
```

```
class CouleurT extends Couleur {
    int t; // nouvel attribut

    // nouvelle méthode :
    void setT (int t) { ... }
}
```

Constructeurs Une classe dérivée comporte *obligatoirement* au moins un constructeur. La première ligne de ce (ou ces) constructeur(s) est *obligatoirement* un appel direct par `this` ou indirect par `super` à l'un des constructeurs de la classe héritée. Illustration :

```
class Couleur {
    int r, v, b; // attributs

    // 1er constructeur :
    Couleur (int r, int v, int b) {
        this.r = r;
        this.v = v;
        this.b = b;
    }

    // 2ème constructeur :
    Couleur () {
        // appel au 1er constructeur :
        this(255, 255, 255);
    }
}
```

```
class CouleurT extends Couleur {
    int t; // nouvel attribut

    CouleurT (int r, int v, int b, int t) {
        // appel direct au 1er constructeur :
        super(r, v, b);
        this.t = t;
    }

    CouleurT (int t) {
        // appel direct au 2ème constructeur :
        super();
        this.t = t;
    }

    CouleurT () {
        // appel indirect au 2ème construct. :
        this(255);
    }
}
```

Cet appel direct ou indirect à un constructeur de la classe héritée est parfaitement logique. Avant d'initialiser ce qui peut être spécifique à une classe dérivée, on doit initialiser les attributs hérités. Il serait maladroit (voire erroné dans certain cas) d'initialiser les attributs hérités sans passer par un constructeur de la classe héritée.

Appel à super implicite Si la première ligne d'un constructeur n'est ni un appel à `this` ni un appel à `super`, alors la première ligne est *implicitement* un appel à `super` sans paramètres :

```
// ni this ni super en 1ère ligne :
CouleurT (int t) {
    this.t = t;
}

// implicitement :
CouleurT (int t) {
    super();
    this.t = t;
}
```

Attention : dans ce cas, il **doit** exister un constructeur sans paramètre dans la classe héritée :

```
class Couleur {
    int r, v, b; // attributs
    // seul constructeur :
    Couleur (int r, int v, int b) {
        this.r = r;
        this.v = v;
        this.b = b;
    }
}

class CouleurT extends Couleur {
    int t; // nouvel attribut
    // ERREUR :
    CouleurT (int t) {
        // implicitement : super();
        // -> désigne un constructeur absent !
        this.t = t;
    }
}
```

Constructeur implicite Si une classe dérivée ne comporte *aucun* constructeur (et seulement dans ce cas !), alors un constructeur implicite sans paramètre est introduit dans la classe. La classe héritée doit alors *obligatoirement* disposer dans ce cas d'un constructeur sans paramètre :

```
class Couleur {
    int r, v, b; // attributs
    Couleur (int r, int v, int b) {
        this.r = r;
        this.v = v;
        this.b = b;
    }

    Couleur () { // 2ème constructeur
        this(255, 255, 255);
    }
}

class CouleurT extends Couleur {
    int t; // nouvel attribut

    // implicitement :
    class CouleurT extends Couleur {
        int t; // nouvel attribut

        CouleurT () {
            super();
        }
    }
}
```

Accès aux membres hérités On accède aux membres hérités par l'intermédiaire de l'opérateur point comme s'ils étaient déclarés dans la classe elle-même. Illustration :

```
class Couleur {
    int r, v, b;
    Couleur (int r, int v, int b) { ... }
    void setR (int t) { ... }
}

class CouleurT extends Couleur {
    int t;
    CouleurT (int r, int v, int b, int t) { ... }
    void setT (int t) { ... }
}
```


Annotation Il est d'usage d'annoter la redéfinition d'une méthode avec `@Override` afin que le compilateur vérifie que la signature correspond *effectivement* à celle d'une méthode héritée :

```
class Couleur {
    // attributs et constructeurs...

    String codeHTML() { ... }
}

class CouleurT extends Couleur {
    // attributs et constructeurs...
    @Override
    String codeHTML() { ... } // ok
}
```

Variation de visibilité La redéfinition d'une méthode peut être l'occasion d'*augmenter la visibilité* de cette méthode (de la visibilité par défaut à `protected` ou à `public`, ou de `protected` à `public`). La *réduction de visibilité* est quant à elle *interdite* :

```
class Couleur {
    // attributs et constructeurs...

    String codeHTML() { ... }

    void enRouge() { ... }
}

class CouleurT extends Couleur {
    // attributs et constructeurs...
    @Override
    public String codeHTML() { ... } // OK.
    @Override
    private void enRouge() { ... } // KO !
}
```

3.2.3 Polymorphisme et liaison dynamique

Instances polymorphes Les objets sont typés par la classe dont ils sont instance, ainsi que par *toutes les classes héritées* par cette même classe. Une variable de type `A` peut donc stocker la référence d'une instance de type `A` ou de ses dérivées. Illustration :

```
Couleur c; // variable typée par Couleur
c = new Couleur (61, 194, 93); // ok : un objet Couleur est de type Couleur
c = new CouleurT(61, 194, 93, 64); // ok : un objet CouleurT est aussi de type Couleur
```

Test de type Si une variable référence un objet, alors cet objet a nécessairement le type de la variable. Mais cet objet peut en avoir d'autre s'il est polymorphe. L'opérateur `instanceof` permet de vérifier qu'un objet a *en particulier* un certain type :

```
Couleur c = new Couleur (61, 194, 93);
boolean test = (c instanceof "Couleur"); // true (évidemment !)
test = (c instanceof CouleurT); // false
c = new CouleurT(61, 194, 93, 64);
test = (c instanceof Couleur); // true (toujours évidemment...)
test = (c instanceof CouleurT); // true
```

Invocation de méthode Soit `o`, instance d'une classe `c` référencée par une variable ou un paramètre `x` de type `t`. Si le code comporte l'appel d'une méthode `f` sur `x`, ce n'est pas `t` qui détermine la bonne version de `f` à exécuter. C'est `c` qui le détermine. Or, `c` peut n'être connu qu'à l'exécution. C'est donc grâce à un mécanisme nommé *liaison dynamique* que la bonne version de `f` est déterminée à l'exécution en exploitant un algorithme de recherche appelé *method lookup*. Illustration :

```
class Couleur {
    // attributs et constructeurs...

    String codeHTML() { ... }
}

class CouleurT extends Couleur {
    // attributs et constructeurs...
    @Override
    String codeHTML() { ... }
}

static void afficheCode(Couleur c) {
    String s = c.codeHTML(); // version de Couleur ou de CouleurT ?
    System.out.println(s);
}

afficheCode(new Couleur(61, 194, 93)); // c'est la version de Couleur qui s'exécute
afficheCode(new CouleurT(61, 194, 93, 64)); // c'est la version de CouleurT qui s'exécute
```

3.2.4 Classe Object

Héritage implicite Lorsqu'une classe n'hérite pas explicitement d'une autre classe, elle hérite implicitement de la classe `Object`, qui dispose d'un constructeur sans paramètre et d'un certain nombre de méthodes comme `toString` et `equals` par exemple :

```
public class Object {
    // constructeur :
    public Object() { ... }
    // méthodes :
    public String toString () [ ... ]
    public boolean equals (Object obj) [ ... ]
    // ...
}

// classe en apparence non dérivée :
class Couleur {
    int r, v, b;
    Couleur (int r, int v, int b) {
        this.r = r;
        this.v = v;
        this.b = b;
    }
}

// classe en réalité dérivée implicitement :
class Couleur extends Object {
    int r, v, b;
    Couleur (int r, int v, int b) {
        super(); // appel implicite
        this.r = r;
        this.v = v;
        this.b = b;
    }
}
```

Généricité simple Par construction, tous les objets en Java sont de type `Object` puisque toutes les classes en dehors de `Object` héritent directement ou indirectement de `Object`. Cette caractéristique permet de créer des programmes génériques simples. Par exemple, la fonction suivante indique si un objet est présent (selon `=`) dans un tableau ou non :

```
static boolean present(Object o, Object[] tab) {
    for (Object x : tab)
        if (o == x)
            return true;
    return false;
}
```


TD 1

Fondements impératifs de Java

Les fonctions demandées dans cette première série d'exercices sont des formes particulières de *méthodes* dites *statiques*. C'est le cas de la fonction `select` suivante par exemple, qui renvoie le *i*^{ème} caractère d'une chaîne s'il existe, ou `'\0'` sinon :

```
static char select(int i, String s) {
    if (s==null || i<1 || i>s.length()) return '\0';
    // ici : s existe, et le ième caractère de s aussi
    return s.charAt(i-1); // rappel : les indices de caractères commencent à 0
}
```

Le code correspondant aux corrections des exercices est disponible ici : <http://informatique.univ-brest.fr/java/Source>. Évidemment, consulter prématurément ce code ne présente aucun intérêt. Il convient de chercher par soi-même les solutions aux exercices proposés.

Tous les exercices sont à faire en séance si le temps le permet, ou sur temps de travail personnel sinon. Les exercices et les questions marqués d'une étoile ne sont pas à faire prioritairement en séance (mais sont malgré tout à faire!).

Exercice 1

Définissez la fonction `max` qui s'applique à 3 entiers et renvoie la plus grande valeur des 3 :

- version 1 : en utilisant l'*instruction* conditionnelle
- version 2 : en utilisant l'*opérateur* conditionnel

Exercice 2

Définissez la fonction `ordered` qui s'applique à 3 entiers et qui renvoie en résultat un booléen qui indique si les 3 valeurs fournies sont triées par ordre croissant :

- version 1 : en utilisant l'*instruction* conditionnelle
- version 2 : en utilisant l'*opérateur* conditionnel
- version 3 : en n'utilisant aucune forme de conditionnelle

Remarque : la version 3 est toujours à privilégier.

Exercice 3

Définissez la fonction `entier` qui s'applique à un flottant de type `double` et qui renvoie en résultat un booléen indiquant si la valeur fournie correspond à une valeur entière :

- version 1 : sans tenir compte de l'approximation des flottants
- version 2 * : en tenant compte d'une marge d'erreur de l'ordre d'un millionième

Par exemple, les valeurs flottantes 5.00001 et 5.0000001 ne sont pas considérées comme des valeurs entières dans la version 1. Par contre dans la version 2, la valeur 5.0000001 est considérée comme une valeur entière car ce qui la sépare de 5.0 est inférieur à un millionième.

Exercice 4

Définissez la fonction `fact` qui s'applique à un entier de type `int` et qui renvoie en résultat la factorielle de la valeur fournie (type `int` aussi) :

- version 1 * : en utilisant la récursivité
- version 2 : en utilisant une itération
- version 3 : en n'utilisant ni récursivité, ni itération

Si la valeur fournie est négative, alors `fact` renvoie -1. Si la valeur fournie est supérieure à 12, alors le résultat excède la capacité de `int`. Dans ce cas, `fact` renvoie -2.

Exercice 5

Définissez la fonction `maxtab` qui s'applique à un tableau d'entiers non vide et qui renvoie en résultat la plus grande valeur du tableau.

Exercice 6 *

Définissez la fonction `inverse` qui s'applique à un tableau d'entiers et qui inverse l'ordre de ses éléments. Illustration :

```
int[] tab = { 5, -3, 8, 7, 4, -2, 4, 0 };
inverse(tab); // tab : { 0, 4, -2, 4, 7, 8, -3, 5 }
```

Attention : vous ne devez pas créer de tableau pour réaliser cette fonction.

Exercice 7

Définissez la fonction `cptab` qui s'applique à un tableau d'entiers et qui renvoie en résultat une copie du tableau fourni :

- version 1 : en utilisant une boucle de parcours
- version 2 : en utilisant la méthode `arraycopy` de la classe `System`
- version 3 : en utilisant la méthode `copyOf` de la classe `Arrays`
- version 4 : en utilisant la méthode `clone` des tableaux

TD 2

Classes, attributs, méthodes et constructeurs

Pour les besoins d'un programme, on s'intéresse à la notion d'*étudiant*. Pour ce programme, les seules caractéristiques utiles d'un étudiant sont son *nom*, son *prénom* et ses *notes*. On décide de représenter cette notion et ces caractéristiques par la classe suivante :

```
class Etudiant {
    String nom;
    String prenom;
    double[] notes;
}
```

Une *instance* de cette classe est un *objet* qui représente un étudiant particulier, dans un état particulier (e.g. l'étudiant nommé *Einstein*, prénommé *Albert*, et dont les notes connues sont 18,5 et 4). On obtient une instance de la classe *Etudiant* en invoquant sur elle l'opérateur *new*. Les attributs d'une nouvelle instance ont soit des valeurs par défaut, soit des valeurs définies par un constructeur. Le constructeur est invoqué par l'intermédiaire de *new*.

Exercice 1

Définissez un constructeur dans la classe *Etudiant* qui permet d'initialiser les attributs à partir d'un nom et d'un prénom (le tableau des notes doit exister mais ne doit pas avoir de cases). Ce constructeur doit pouvoir être invoqué par *new* de la manière suivante :

```
Etudiant e; // variable de type Etudiant : ne contient rien pour l'instant
e = new Etudiant("Einstein", "Albert"); // e désigne maintenant un objet :
// - dont le nom est "Einstein"
// - dont le prénom est "Albert"
// - dont le tableau des notes existe mais ne contient aucune case
```

Exercice 2

Définissez la méthode *toString* qui renvoie une chaîne de caractères représentant un étudiant conformément au schéma : `NOM Prénom : note1 note2 ... noten`. Le nom doit être en majuscules et le prénom en minuscules sauf l'initiale. Illustration :

```
Etudiant e = new Etudiant("Einstein", "Albert");
String r = e.toString(); // "EINSTEIN Albert : "
```

Rappels :

- cette méthode doit être déclarée **public** (pour des raisons vues plus loin liées à l'héritage)
- les méthodes de *String* utiles sont résumées dans le *Java Cheat Sheet*

Exercice 3

Définissez la méthode *addNote* qui prend un flottant en paramètre, et qui le rajoute dans le tableau des notes de l'étudiant. Le tableau doit au final contenir autant de cases que de notes. La méthode *copyOf* de la classe *Arrays* pourra être utile. Illustration :

```
Etudiant e = new Etudiant("Einstein", "Albert");
e.addNote(18.5); // ajout de la note 18,5
e.addNote(4.0); // ajout de la note 4
String r = e.toString(); // "EINSTEIN Albert : 18.5 4.0 "
```

Exercice 4

Définissez la méthode *moyenne* qui renvoie la moyenne de l'étudiant. S'il n'a pas de note, sa moyenne vaut par défaut 0. Illustration :

```
Etudiant e = new Etudiant("Einstein", "Albert");
e.addNote(18.5); // ajout de la note 18,5
e.addNote(4.0); // ajout de la note 4
double m = e.moyenne(); // 11,25
```

Exercice 5

Définissez la méthode *moyenneGenerale* qui prend un tableau d'étudiants en paramètre, et qui renvoie la moyenne générale de ces étudiants. Si le tableau n'a pas de cases, la moyenne vaut par défaut 0. Déduisez du code une propriété importante de cette méthode. Donnez un exemple d'appel de cette méthode.

Exercice 6

On s'intéresse maintenant à la notion de *promotion* caractérisée par un nom (une chaîne de caractères) et un ensemble d'étudiants (un tableau). Définissez :

- une classe qui représente cette notion
- un constructeur qui permet d'initialiser une promotion
- une méthode *toString* qui fournit la représentation textuelle d'une promotion

Le constructeur doit permettre de fournir un nom à la promotion. À son initialisation, une promotion n'a aucun étudiant. La méthode *toString* fournit une chaîne de caractères conforme au schéma suivant : `Nom [etudiant1] ... [etudiantn]`.

Exercice 2

Créez la méthode `add` dans `Repertoire` qui prend en paramètre un élément de répertoire et qui l'ajoute au contenu, seulement s'il ne porte pas un nom déjà pris dans le répertoire. Cette méthode renvoie l'objet fourni en paramètre si l'ajout a été possible, ou `null` sinon. Illustration :

```
Repertoire r = new Repertoire("racine");
r.add(new Fichier("fich")); // non null (fichier ajouté à r)
r.add(new Repertoire("fich")); // null (pas d'ajout dans r)
```

Exercice 3

Un élément doit d'abord être créé puis ajouté par `add` au répertoire qui doit le contenir, or :

- le répertoire peut ne pas pouvoir recueillir l'élément à cause d'un nom déjà pris
- l'ajout non contrôlé d'un répertoire existant peut créer un cycle dans l'arborescence

Pour éviter ces problèmes, on confie à `Repertoire` la responsabilité *exclusive* de fournir de nouvelles instances d'éléments grâce aux méthodes `createFich`, `createRep` et `createLien`, que vous devez compléter. Ces méthodes créent, ajoutent au répertoire et renvoient en résultat un nouvel élément *seulement* si son nom n'est pas déjà pris. Elles renvoient `null` sinon. Illustration :

```
Repertoire r = new Repertoire("racine");
Fichier e1 = r.createFich("fich"); // non null (fichier créé et ajouté à r)
Repertoire e2 = r.createRep("fich"); // null (répertoire non créé, pas d'ajout dans r)
```

Exercice 4

À ce stade, il est toujours possible d'utiliser librement les constructeurs et de modifier les noms, les contenus de répertoire ou encore les cibles de liens, ce qui peut rendre le système de fichiers incohérent. Par exemple, un répertoire pourrait contenir deux éléments de même nom, ou il pourrait apparaître parmi ses sous-répertoires. Pour éviter ces situations, on place toutes les classes dans un package nommé `fs` (pour *file system*). Modifiez le code et au besoin ajoutez des méthodes de manière à ce que les 6 propriétés suivantes soient respectées :

1. seules les classes `Repertoire`, `Fichier` et `Lien` sont accessibles de l'extérieur de `fs`
2. l'opérateur `new` est inapplicable à `Repertoire`, `Fichier` et `Lien` de l'extérieur de `fs`
3. un nom d'élément est accessible en lecture mais n'est pas modifiable
4. la cible d'un lien est accessible en lecture mais n'est pas modifiable
5. le contenu d'un répertoire est accessible en lecture mais n'est pas modifiable
6. en dehors des éventuelles nouvelles méthodes, les seules méthodes utilisables de l'extérieur de `Repertoire` sont les méthodes `createFich`, `createRep` et `createLien`

Exercice 5

De l'extérieur du package `fs`, créer un fichier, un lien ou un répertoire n'est désormais possible qu'à partir d'un répertoire existant. Il en faut donc au moins un au départ : le répertoire *racine*. Trouvez une solution pour fournir à l'extérieur du package `fs` un répertoire unique et partout accessible qui représente la racine.

Complétez ensuite la méthode `main` d'une classe `Programme` d'un autre package `utilisation.de.fs`. Cette méthode `main` doit permettre de créer les éléments de l'exemple introductif :

- un répertoire `tmp` à la racine
- un fichier `f.txt` dans `tmp`
- un lien à la racine nommé `aux` vers le répertoire `tmp`.
- un lien à la racine nommé `dat` vers le fichier `f.txt`.

Exercice 6 *

Modifiez le code de manière à ce qu'un fichier, un répertoire ou un lien ait accès au répertoire qui le contient. Utilisez ensuite cette nouvelle propriété pour programmer la méthode `getNomAbs` de la classe `ElementDeRepertoire` qui renvoie une chaîne de caractères. Cette chaîne correspond au nom *absolu* de l'élément. Illustration :

```
Repertoire r = /*...*/ ; // création du répertoire : /temp/java/
Fichier f = r.createFich("Prog.java"); // nom de f : Prog.java
String nomAbsF = f.getNomAbs(); // nom absolu de f : /temp/java/Prog.java
```

Exercice 7 *

Dessinez un diagramme de classes UML qui représente toutes les classes du package `fs`. Il n'est pas nécessaire de faire apparaître les constructeurs.

Exercice 8 retour sur le sujet précédent *

On ajoute aux notions du sujet précédent la notion d'*étudiant boursier*. Un étudiant boursier est un étudiant qui dispose d'une bourse caractérisée par un *echelon* (valeur entière comprise entre 1 et 5). Définissez la classe `EtudiantBoursier` qui hérite de `Etudiant` et qui introduit un nouvel attribut nommé `echelon`. Définissez un constructeur pour cette classe qui prend (entre autre) en paramètre une valeur d'échelon. Les valeurs fournies inférieures à 1 seront assimilées à 1, et les valeurs fournies supérieures à 5 seront assimilées à 5.

Exercice 9 *

Définissez un autre constructeur dans la classe `EtudiantBoursier` qui prend en plus en paramètre la promotion de l'étudiant :

- en vous appuyant sur le constructeur à 3 paramètres de `Etudiant`
- en vous appuyant sur le constructeur à 2 paramètres de `Etudiant`
- en vous appuyant sur le constructeur à 3 paramètres de `EtudiantBoursier`

Quelle est la version à privilégier ?

Exercice 10 *

Définissez une méthode `toString` dans la classe `EtudiantBoursier` qui fournit une représentation littérale de la forme :

```
(b) NOM Prénom : note1 note2 ... noten
```


TP 1

Manipulation de tableaux

Exercice 1

Dans la classe `Programme` du package `exercice1`, complétez la méthode `genTab1` qui crée un tableau d'entiers de `n` cases contenant des valeurs aléatoires comprises entre 0 et 100 (rappel : vous pouvez utiliser la méthode statique `random` de la classe `Math` pour obtenir les valeurs aléatoires attendues). Illustration :

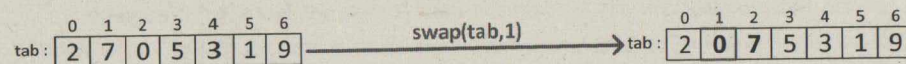
```
int[] tab1Test = genTab1(10); // génération d'un tableau de 10 cases
// tab1Test : [44, 42, 90, 65, 86, 19, 7, 69, 99, 94]
```

Complétez ensuite la méthode `genTab2` qui utilise `genTab1` pour fournir un tableau à 2 dimensions d'entiers dont chaque case contient des valeurs aléatoires comprises entre 0 et 100 :

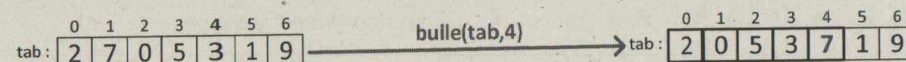
```
int[][] tab2Test = genTab2(2,5); // génération d'un tableau de 2x5 cases
// tab2Test : [80, 94, 72, 78, 95]
//           [59, 50, 97, 64, 80]
```

Exercice 2

Dans la classe `Programme` du package `exercice2`, complétez la méthode `swap` qui échange le contenu des cases d'indices `i` et `i+1` d'un tableau fourni en paramètre. Illustration :

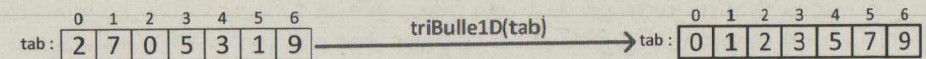


Complétez ensuite la méthode `bulle` qui s'applique à un tableau à une dimension d'entiers et à un entier `n`. Cette méthode utilise `swap` afin de faire remonter dans la case d'indice `n` la plus grande valeur contenue dans le tableau entre les indices 0 et `n` (inclus). Illustration :



Complétez enfin la méthode `triBulle1D` qui s'applique à un tableau à une dimension d'entiers et qui trie le tableau en appliquant le principe du tri à bulles. Le but est d'utiliser la méthode

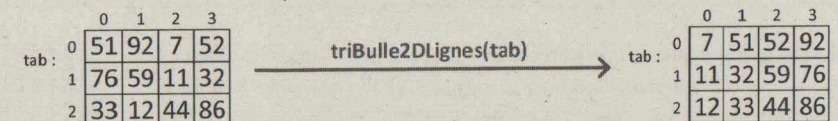
`bulle` afin de placer la plus grande valeur du tableau dans la dernière case, puis de placer la deuxième plus grande valeur du tableau dans l'avant-dernière case, et de continuer ainsi jusqu'à la première case qui doit alors contenir la plus petite valeur du tableau. Illustration :



Exercice 3

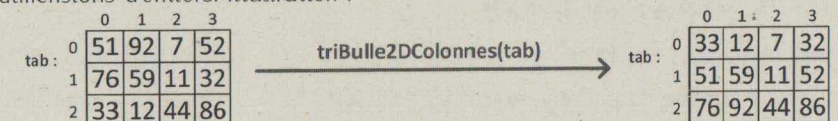
Exercice 3.1

Dans la classe `Programme` du package `exercice3`, complétez la méthode `triBulle2DLignes` qui trie ligne par ligne un tableau à deux dimensions d'entiers. Pour programmer cette méthode, vous pouvez utiliser la méthode précédente `triBulle1D`. Illustration :



Exercice 3.2 *

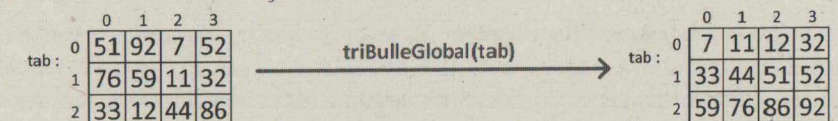
Complétez ensuite la méthode `triBulle2DColonnes` qui trie colonne par colonne un tableau à deux dimensions d'entiers. Illustration :



Indice : vous pouvez recopier et adapter les méthodes `triBulle1D`, `bulle` et `swap` afin qu'elles s'appliquent à une colonne d'indice `j` fourni en paramètre et un tableau à 2 dimensions.

Exercice 3.3 *

Complétez enfin la méthode `triBulleGlobal` qui s'applique à un tableau à deux dimensions d'entiers et qui trie le tableau dans sa globalité, de la première case de la première ligne à la dernière case de la dernière ligne. Illustration :

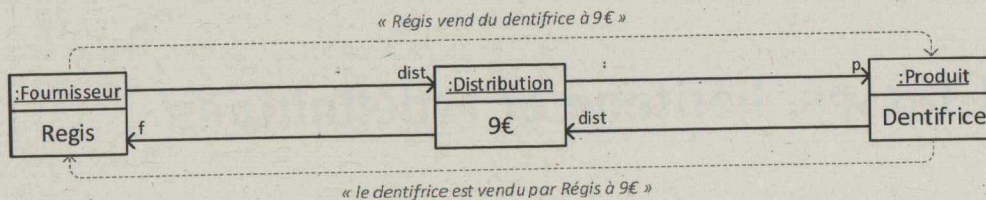


Indice : vous pouvez à nouveau recopier et adapter les méthodes `triBulle1D`, `bulle` et `swap` afin qu'elles s'appliquent à des tableaux à 2 dimensions par l'intermédiaire d'un indice global permettant de les parcourir comme s'il s'agissait de simple tableaux à une dimension :

(ligne,colonne)	(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)	(2,0)	(2,1)	(2,2)	(2,3)
indice global	0	1	2	3	4	5	6	7	8	9	10	11

Exercice 2

Il est très important de bien comprendre que les instances de *Produit* et de *Fournisseur* sont soumises à une *contrainte de cohérence*. Par exemple, si Régis met en vente du dentifrice à 9€, alors réciproquement, on doit pouvoir constater que le dentifrice est bien vendu par Régis à 9€. C'est l'instance de *Distribution* qui assure cette cohérence :



Dans un premier temps, complétez dans les classes *Fournisseur* et *Produit* les deux méthodes *addDistribution* qui ajoutent une distribution au tableau *dist* (celui de *Fournisseur* ou celui de *Produit* selon le cas) :

- en le créant avec une case de plus,
- et en affectant la distribution à la nouvelle case (i.e. la dernière)

Dans un deuxième temps, complétez dans la classe *Fournisseur* la méthode *getDistribution* qui recherche et renvoie la distribution d'un produit si le fournisseur le vend, ou *null* sinon.

Dans un dernier temps, complétez dans la classe *Fournisseur* la méthode *addProduit* qui ajoute ou met à jour la distribution d'un produit par un fournisseur :

- recherche la distribution du produit grâce à la méthode *getDistribution*
- si le produit est déjà mis en vente par le fournisseur : le prix de vente est mis à jour
- si le produit n'est pas déjà mis en vente par le fournisseur : une nouvelle distribution est créée et rajoutée aux tableaux des distributions, côté fournisseur *mais aussi côté produit* grâce aux méthodes *addDistribution*

Illustration :

```

Fournisseur f1 = new Fournisseur("Régis");
Produit p1 = new Produit("Dentifrice");
f1.addProduit(p1, 9); // Régis vend maintenant du dentifrice à 9€
f1.addProduit(p1, 7); // Régis vend maintenant le dentifrice à 7€

```

Exercice 3

Complétez dans la classe *Produit* la méthode *leMoinsCher* qui renvoie le fournisseur qui vend le produit au prix le plus bas (ou *null* si le produit n'est pas en vente). Illustration :

```

Produit p1;
/* ... */
p1.toString() // "Dentifrice [ Régis (9 €) Bernadette (7 €) ]"
p1.leMoinsCher().toString() // "Bernadette [ Dentifrice (7 €) ]"

```

Exercice 4 *

On s'intéresse maintenant à la notion de *commande* de produits auprès de fournisseurs. Dans un premier temps, on s'intéresse à la notion de *ligne de commande* qui rassemble un fournisseur, un produit, un prix unitaire et une *quantité* (i.e. le nombre d'exemplaires du produit commandé). Pour commencer, complétez la classe *CommandeLigne* avec :

- un attribut *dist* qui désigne une *distribution* (donc un fournisseur, un produit et un prix)
- un attribut *qtt* qui désigne la *quantité* commandée

Créez ensuite un constructeur qui prend en paramètre un fournisseur *f*, un produit *p* et une quantité. Si *f* ne vend pas *p*, alors la ligne est dite *erronée* et *dist* doit contenir *null*. Sinon, *dist* référence la distribution qui relie déjà *f* à *p*. Cette distribution peut être obtenue directement grâce à la méthode *getDistribution* de la classe *Fournisseur*. Illustration :

```

Produit p1 = new Produit("Dentifrice");
Fournisseur f1 = new Fournisseur("Régis");
Fournisseur f2 = new Fournisseur("Bernadette");
f1.addProduit(p1, 9); // f1 et p1 sont maintenant reliés par une distribution d
CommandeLigne l1 = new Ligne(f1, p1, 3); // ligne valide : dist référence d
CommandeLigne l2 = new Ligne(f2, p1, 3); // ligne erronée : dist contient null

```

Définissez ensuite la méthode *total* dans la classe *CommandeLigne* qui fournit le montant total de la ligne de commande, ou 0 si la ligne est erronée.

Pour terminer, définissez la méthode *toString* qui renvoie la chaîne *Erreur* si la ligne est erronée, ou une représentation de la forme `quantite x nomp (nomf - prix €) = total €` sinon :

```

Produit p1 = new Produit("Dentifrice");
Fournisseur f1 = new Fournisseur("Régis");
Fournisseur f2 = new Fournisseur("Bernadette");
f1.addProduit(p1, 9); // f1 et p1 sont maintenant reliés par une distribution d
CommandeLigne l1 = new Ligne(f1, p1, 3); // ligne valide : dist référence d
CommandeLigne l2 = new Ligne(f2, p1, 3); // ligne erronée : dist contient null
l1.toString() // "3 x Dentifrice (Régis - 9 €) = 27 €"
l2.toString() // "Erreur"

```

Exercice 5 *

Complétez la classe *Commande* avec les attributs *lignes* de type tableau de *CommandeLigne* et *numero* de type *int*. Définissez un constructeur qui permet de créer une commande initialement vide (i.e. sans lignes) et dont le numéro est attribué *automatiquement* : la première commande porte le numéro 1 et le numéro des commandes suivantes doit être incrémenté à chaque nouvelle instance par un procédé que vous devez définir. Complétez ensuite la 1^{ère} version de *addLigne* qui ajoute une ligne au tableau *lignes* :

- en le créant avec une case de plus,
- et en affectant la ligne à la nouvelle case (i.e. la dernière)

Les cases du tableau `pixels` sont de type `Color`, une classe de l'API standard qui permet de manipuler une couleur par ses composantes RGB :

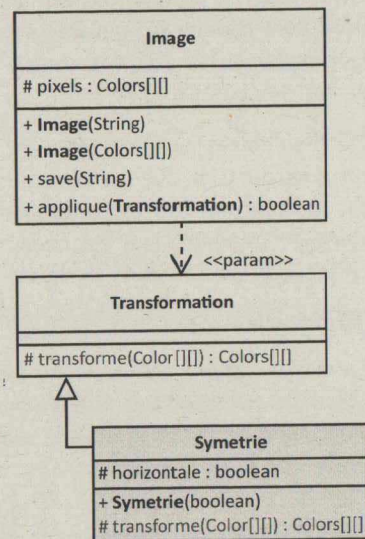
- `Color(int r, int g, int b)` : constructeur de couleur à partir des composantes séparées
- `int getRed()` : renvoie la composante rouge de la couleur (valeur dans [0, 255])
- `int getGreen()` : renvoie la composante verte de la couleur (valeur dans [0, 255])
- `int getBlue()` : renvoie la composante bleue de la couleur (valeur dans [0, 255])

Exercice 1

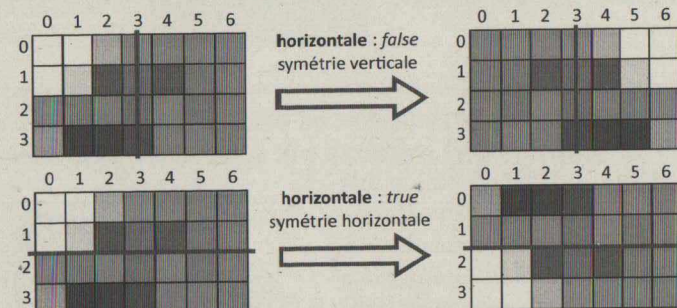
Dans le package `img`, la classe `Image` représente une image par son unique attribut `pixels`. Il peut être initialisé à partir d'un fichier ou d'un tableau déjà existant. La méthode `save` permet de le sauvegarder dans un fichier. Enfin, la méthode `applique` permet de le mettre à jour en lui appliquant une *transformation* (i.e. une instance de la classe `Transformation`). La méthode `transforme` de cette instance renvoie si possible une nouvelle version du tableau de `pixels` qui lui est fourni en paramètre.

Attention :

- cette méthode *ne modifie pas* le tableau en paramètre : le résultat est un *nouveau* tableau
- cette méthode est susceptible de renvoyer `null` en résultat si la transformation n'est pas possible
- la version de cette méthode définie dans la classe `Transformation` renvoie `null` : elle n'est donc pas utile ; elle est destinée à être *redéfinie* dans les sous-classes



La classe `Symetrie` hérite de `Transformation`. Sa version de `transforme` renvoie un tableau symétrique du paramètre selon l'axe horizontal ou selon l'axe vertical en fonction de son attribut `horizontale`. Illustration :



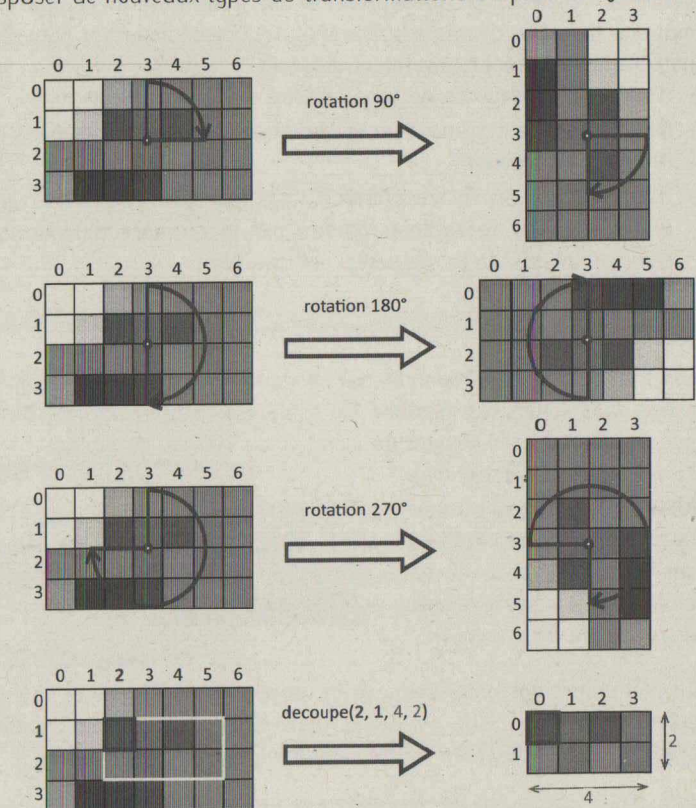
Pour commencer, complétez la méthode `applique` de `Image` qui tente de mettre à jour `pixels` en utilisant la transformation fournie. Si la transformation n'est pas possible (i.e. `null` en résultat), `pixels` ne doit pas être modifié. Le résultat doit indiquer si la transformation a pu être appliquée ou non. Complétez ensuite la méthode `transforme` de `Symetrie`.

Pour tester le code, vous devez utiliser la classe `Programme` du package `img.test`. Les images créées doivent apparaître dans le dossier `images` (à rafraîchir avec la touche F5). Vous pouvez comparer les images obtenues avec les images attendues visibles dans le dossier `images.attendues`.

Exercice 2

On souhaite maintenant disposer de nouveaux types de transformations en plus de `Symetrie`.

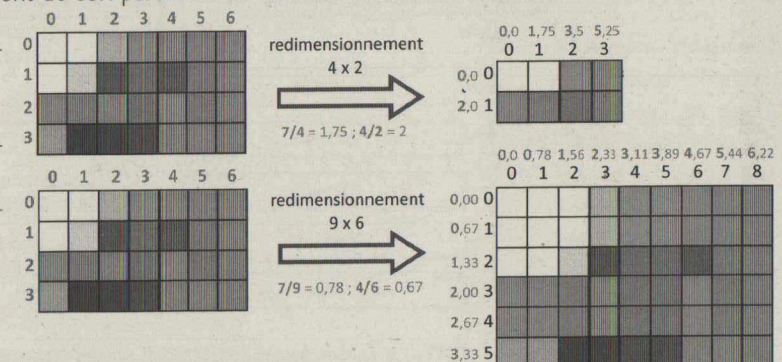
Pour commencer, complétez la méthode `transforme` de `Rotation` qui crée un tableau correspondant à la rotation de son paramètre dans le sens horaire. La rotation est déterminée par l'attribut type dont la valeur doit être 1 (90°), 2 (180°) ou 3 (270°). Pour toute autre valeur, la méthode renvoie `null`.



Complétez ensuite la méthode `transforme` de `Decoupe` qui renvoie un extrait de son paramètre. La zone de découpe est déterminée par les coordonnées de son coin en haut à gauche et par ses dimensions. Si cette zone déborde du tableau, la méthode renvoie `null`.

Complétez enfin la méthode `transforme` de la classe `Dimensions` qui crée un tableau correspondant au redimensionnement de son paramètre comme dans l'illustration suivante :

Le redimensionnement est ici très rudimentaire. Il consiste à dupliquer des pixels en cas d'agrandissement et à supprimer des pixels en cas de réduction (pas d'interpolation).



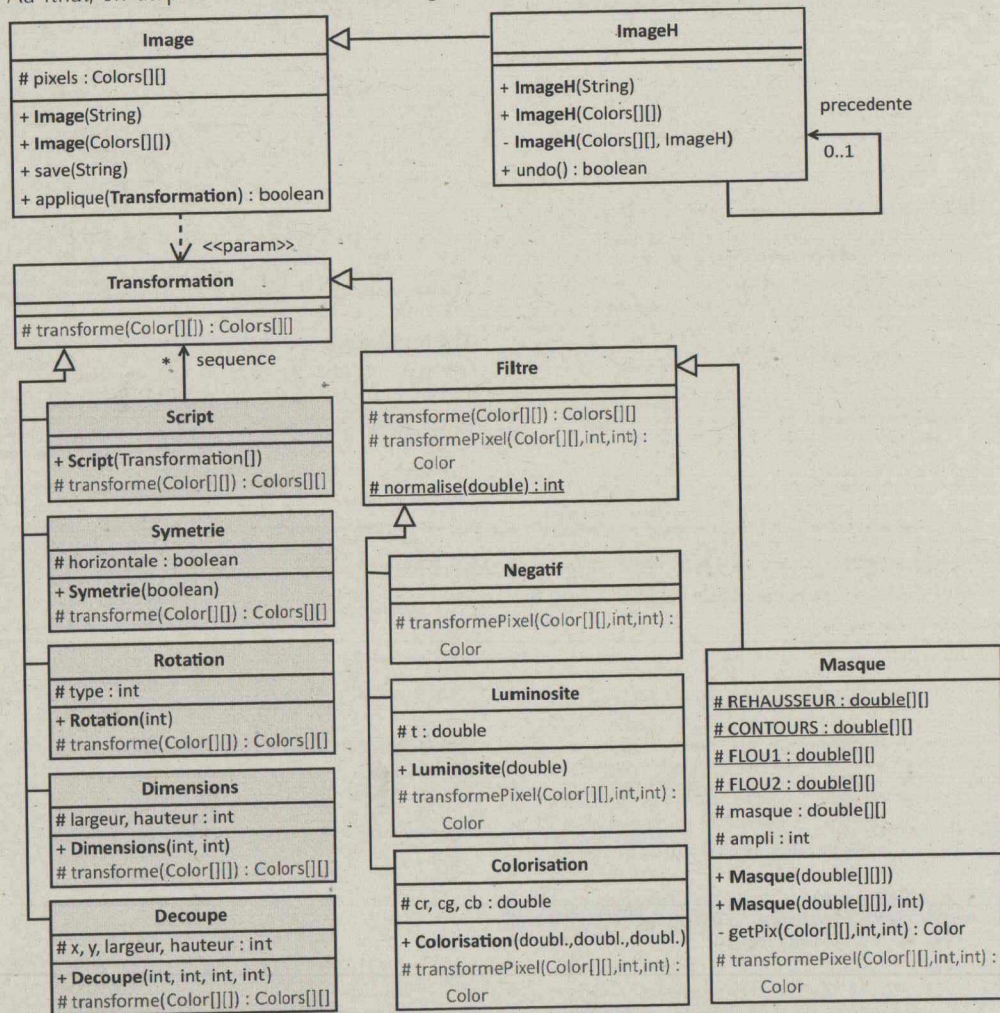
Indice : vous pouvez gérer des indices virtuels flottants de lecture des cases* (en vert dans l'illustration) dont les incréments sont les rapports des dimensions.

Exercice 7

Afin d'ajouter aux images la possibilité d'annuler les transformations appliquées, définissez dans le package `img.h` la classe `ImageH` dérivée de `Image` avec :

- un attribut `precedente` de type `ImageH` qui représente le dernier état de l'image avant sa dernière transformation (null si l'image n'a pas été transformée)
- trois constructeurs : le 1^{er} prend un nom de fichier à charger, le 2^{ème} prend un tableau de pixels préexistant, et le 3^{ème} prend un tableau de pixel *plus* une valeur initiale pour `ImageH`
- une redéfinition de `applique` qui permet d'enregistrer l'état de l'image avant sa transformation
- la méthode `undo` qui annule si possible la dernière transformation et renvoie vrai dans ce cas

Au final, on dispose de 13 classes d'images et de transformations organisées en 2 hiérarchies :



Quatrième partie

Java Cheat Sheet