UO LaTeX - Programmer en Java

Thomas BOCQUELET

6 mai 2018

Résumé

Ce document a été réalisé à partir du cours (datant de 2017) de M^{r} Grégory Bourgin, enseignant-chercheur et Maître de conférence à l'Université du Littoral Côte d'Opale.

Email : bourguin@lisic.univ-littoral.fr

Page internet : `http://www-lisic.univ-littoral.fr/spip.php?article50&membre=10`

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Présentation du langage

1.1 Qu'est-ce que le Java ?

Simula 67 a été conçu par une équipe scandinave et a été publié en 1967.

Au début des années 70, Alan Kay conçoit au PARC (Rank Xerox) le langage SmallTalk, qui est encore aujourd'hui *la* référence dans les langages orientés objet.

A la fin des années 70 et au début des années 80, on assiste à la naissance de nombreuses extensions objet d'anciens langages : Object Pascal, Objective C, C++, CLOS, ADA...

Au milieu des années 90, Sun publie Java. Ses qualités intrinsèques et le fait qu'il soit particulièrement adapté à Internet en font immédiatement un standard.

Une énorme masse de documentation peut être trouvée sur Internet.

Le Java est :

- Un langage orienté objet (*POO*[1])
- Une architecture *Virtual Machine*
- Un ensemble d'API variées
- Un ensemble d'outils : le *JDK*[2]
- Portable :
  - La compilateur Java génère du langage *byte code*
  - La *JVM*[3] est présente sur la majeure partie des systèmes d'exploitation Windows, Mac, Unix...
  - Il dispose d'une sémantique très précise
  - Il supporte un code écrit en *Unicode*
  - Il est accompagné d'une librairie standard
- Robuste :
  - Orienté à l'origine pour des applications embarquées
  - Gère la mémoire par *garbage collector*
  - Dispose d'un mécanisme d'*exceptions*
  - Convertissions sûres automatiques uniquement
  - Contrôle des *cast* à l'exécution

ATTENTION:

- **Le Java n'est pas du JavaScript, car le Java est un langage généraliste, contrairement au JavaScript qui est un langage orienté sur la programmation Web !**

1. Programmation Orientée Objet
2. Java Development Kit
3. Java Virtual Machine

— **Le Java n'est pas du C++ ! Java est un langage objet purement objet, et de plus haut niveau.**

1.2 Les outils

Pour programmer en Java sur ordinateur, nous utiliserons l'un des environnements de développement (*IDE*) suivant :

— SunJDK (compilateur, interpréteur...)
— Eclipse (gratuit)
— IntelliJ (version « community » gratuite, mais version commerciale payante)
— NetBeans

Liste des outils utilisés dans la programmation Java :

javac : compilateur de sources Java

java : interpréteur de byte code

appletviewer : interpréteur d'applet

javadoc : générateur de documentation (HTML, MIF)

javah : générateur de header pour l'appel de méthodes natives

javap : désassembleur de byte code

jdb : debugger

javakey : générateur de clés pour la signature de code

Liste des *API* standards :

java.lang : types de bases, etc.

java.util : HashTable, Vector, Stack, Date...

java.io : accès aux entrées/sorties par flux

java.net : socket, URL...

java.sql : accès homogène aux bases de données

java.security : signatures, cryptographie, authentification...

1.3 Références

Java dispose d'un grand nombre de ressources sur internet. A l'heure actuelle, nous sommes à la 8^{e} version de Java : `https://docs.oracle.com/javase/8/`

Chapitre 2

Les éléments du langage

2.1 Les types primitifs

Type	Taille	Valeur minimale	Valeur maximale	Exemple
`byte`	8 bit	-128	127	`byte b = 64;`
`char`	16 bit	0	$2^{16}-1$	`char c = 'A'; char d = 64;`
`short`	16 bit	-2^{15}	$2^{15}-1$	`short s = 65;`
`int`	32 bit	-2^{31}	$2^{31}-1$	`int i = 1;`
`long`	64 bit	-2^{63}	$2^{63}-1$	`long i = 65L;`
`float`	32 bit	-2^{-149}	$2-2^{-23}\times 2^{127}$	`float f = 65f;`
`double`	64 bit	-2^{-1074}	$2-2^{-52}\times 2^{1023}$	`double d = 65.55;`
`boolean`	1 bit			`boolean b = true; boolean c = false;`
`void`				

TABLE 2.1 – Tableau des types primitifs

2.2 Les variables

En Java, les variables sont typées, et peuvent être déclarées dans n'importe quel bloc du code.

Exemple 2.2.1 *On considère le code suivant :*

```
if(...) { //BLOC 1
          int x;
          ...
          if(...) { //BLOC 2
                    int y; ...
          }
          else { //BLOC 3
                    ...
          }
          ...
}
```

Résultat : La variable `x` sera utilisable dans les blocs 1, 2 et 3. La variable `y` ne sera utilisable que dans le bloc 2.

Liste des opérateurs d'affectation :

- `=`
- `+=`
- `-=`
- `*=`
- `/=`
- `%=`

2.3 Expressions

Définition 1 (Expression ternaire) *Une* expression ternaire *est une notation « simplifiée » d'un test logique.*

Exemple 2.3.1 *Test classique*

```
int i=100;
int y=20;
int maximum;
if(x > y) {
        maximum = x;
}
else {
        maximum = y;
}
```

Exemple 2.3.2 *Test sous forme d'opérateur ternaire*

```
int i=100;
int y=20;
int maximum = (x > y) ? x : y;
```

Définition 2 (Type casting) *Il est nécessaire de* caster *des affectations lorsque celles-ci ne sont pas implicites, sinon des erreurs de compilation sont détectées.*

```
int i=258;
long l=i; //OK
byte b=i; //ERROR: Explicit cast needed to convert int to byte
byte b=258; //ERROR: Explicit cast needed to convert int to byte
byte b=(byte)i; //OK mais b=2
```

Remarque 1 *Levé d'ambiguité entre* `float` *et* `double`

```
float f=2564.5; //ERREUR de compilation
float f=2564.5f; //OK
```

2.4 Les méthodes

Définition 3 *Une* méthode *est une* fonction *appartenant à une* classe.

Une *méthode* se définit comme suit :

```
TypeRetour nomMethode(parametre1, parametre2...) {
        ... corps ...
}
```

Remarque 2 *Le type de retour est un type primitif, une classe ou* `void`.

Remarque 3 *La liste des paramètres peut être vide.*

Remarque 4 *Si le type de retour n'est pas un* `void`, *la fonction doit se terminer par un* `return`.

Passage de paramètres :
— Les paramètres de type simple (`int`, `float`) sont passés par valeur uniquement
— Les paramètres de type *objet* ou tableau sont passés par *référence*

2.5 Les structures de contrôle

2.5.1 Si (`if`)

Le code à l'intérieur d'un `if` s'exécute uniquement si la condition est vraie.
ATTENTION: Plusieurs notations sont possibles ! Soyez vigilants au nombre de lignes de code dans votre bloc d'instructions (si votre condition est vraie) pour bien choisir la notation.
Différentes notations :
— `if(condition){...} else {...}`
— `if(condition)instruction;`
— `if(condition)instruction; else instruction;`
— `if(condition)instruction; else {...}`
— `if(condition){...} else instruction;`

Exemple 2.5.1 *Exemple d'un test simple, avec deux notations différentes :*

```
int i=0;
//NOTATION 1
if(i == 0) {
        i++; //i=i+1;
}
else {
        i+=2; //i=i+2;
}

//NOTATION 2
if(i==0) i++; else i+=2;
```

2.5.2 Tant que (`while`)

Le code à l'intérieur d'un `while` s'exécute *tant que* la condition est vraie.

Remarque 5 *Comme pour le `if` (voir paragraphe n° 2.5.1, page 9), il existe plusieurs notations possibles.*

Différentes notations :
— `while(condition){...}`
— `while(condition)instruction;`

Exemple 2.5.2 *Exemple d'une boucle qui affiche les chiffres de 0 à 9 :*

```
int i=0;
while(i < 10) {
        System.out.println(i);
        i++;
}
```

Remarque 6 *Le `while`, tel que définit jusqu'à présent, vérifie la condition avant d'exécuter (au moins une fois) les instructions. Dans certains cas, il pourrait être utile d'exécuter les instructions une première fois, avant de vérifier si la condition est vraie : on utilisera `do`.*

Exemple 2.5.3 *Exemple d'une boucle qui affiche également les chiffres de 0 à 9 :*

```
int i=0;
do{
        System.out.println(i);
        i++;
}while(i >= 1 && i < 10)
```

Explication du fonctionnement :

1. A la 1re exécution, le code à l'intérieur du bloc `do` sera exécuté : `i` vaudra donc 1
2. Puisque la condition du `while` est vrai, le code sera exécuté jusqu'à ce que `i` soit égal à 9 (lors de l'exécution, vous verrez les chiffres de 0 à 9)
3. `i` valant ensuite 10, le code du `while` ne sera plus exécuté, car la condition est fausse

Remarque 7 *Sans l'instruction `do`, rien ne se serait affiché sur votre terminal, et le bloc d'instructions n'aurait pas été exécuté une seule fois !*

2.5.3 Pour (`for`)

Le code à l'intérieur d'une boucle `for` s'exécute un nombre *défini* de fois, contrairement au `while` qui peut s'exécuter à l'infini (à proscrire bien sûr).
Différentes notations :
— `for(initialisation, condition, incrementation){...}`
— `for(initialisation, condition, incrementation)instruction;`
Avec :

initialisation : Initialisation de la (ou des) variables de boucle

condition : La boucle sera répétée *tant que* la condition sera vraie

incrementation : Incrémente la variable de boucle (permet de passer « d'une étape à une autre »)

Exemple 2.5.4 *Exemple d'une boucle* `for` *affichant les chiffres de* 0 *à* 9 *:*

```
for(int i=0; i<10; i++) {
        System.out.println(i);
}
```

Remarque 8 *Le code suivant est également correct : (la condition de fin est écrite différemment)*

```
for(int i=0; i<=9; i++) {
        System.out.println(i);
}
```

2.5.4 Tests en série (`switch`)

Un `switch` est un bloc contenant une série de tests. On peut le comparer à une succession de `if`.

Exemple 2.5.5 *Supposons un programme destiné à afficher le résultat d'un étudiant à son semestre avec les consignes suivantes :*

- *Note supérieure ou égale à* 0 *et strictement inférieure à* 10 *: ajourné*
- *Note supérieure ou égale à* 10 *et strictement inférieure à* 12 *: admis*
- *Note supérieure ou égale à* 12 *et strictement inférieure à* 14 *: admis mention assez bien*
- *Note supérieure ou égale à* 14 *et strictement inférieure à* 16 *: admis mention bien*
- *Note supérieure ou égale à* 16 *et strictement inférieure à* 20 *: admis mention très bien*
- *On doit traiter les erreurs de saisie : un message d'erreur sera retourné si la note est inférieure à* 0 *et supérieure à* 20

Avec la structure en `if` :

```
float note = 12.52;
if(note >= 0 && note < 8) {
        System.out.println("Aie ! J'ai mal !");
}
else if(note >= 8 && note < 10) {
        System.out.println("Un petit effort");
}
else if(note >= 10 && note < 12) {
        System.out.println("Admis");
}
else if(note >= 12 && note < 14) {
        System.out.println("Admis mention assez bien !");
}
else if(note >= 14 && note < 16) {
        System.out.println("Admis mention bien !");
}
else if(note >= 16 && note <= 20) {
        System.out.println("Admis mention tres bien !");
```

```
}
else {
        System.out.println("ERREUR : La note doit etre comprise
            entre 0 et 20 !");
}
```

Avec la structure `switch` :

```
float note = 12.52;
switch(note) {
        case 0:
                System.out.println("Aie ! J'ai mal !");
                break;
        case 8:
                System.out.println("Un petit effort !");
                break;
        case 10:
                System.out.println("Admis");
                break;
        case 12:
                System.out.println("Admis mention assez bien");
                break;
        case 14:
                System.out.println("Admis mention bien");
                break;
        case 16:
                System.out.println("Admis mention tres bien");
                break;
        default:
                System.out.println("ERREUR : La note doit etre
                    comprise entre 0 et 20 !");
}
```

Remarque 9 *Afin de mieux gérer les erreurs, on pourrait ajouter une boucle `while`, de manière à redemander la saisie si la valeur n'est pas celle attendue.*

2.5.5 Éléments supplémentaires des structures de contrôle

break : permet de stopper une boucle (par exemple infinie)

continue : permet de passer automatiquement à l'itération suivante sans exécuter les instructions suivantes de la boucle

2.6 Entrées / Sorties

Les entrées/sorties sont possibles grâce à 2 flux :
— `System.in` pour les *entrées* (au clavier)
— `System.out` pour les *sorties* (affichage dans le terminal)

Pour afficher des éléments dans le terminal, on peut utiliser les commandes `print` et `println`. `println` affiche dans un terminal avec un retour chariot, contrairement à `print` qui ne se contente que d'afficher.

Exemple 2.6.1 *Exemple d'affichage :*

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

String name = "Toto";
System.out.prinln("Nom:" + name);
```

Remarque 10 *Notez la présence d'un « S » majuscule dans le type de la variable* `name`.
ATTENTION: En Java, pour déclarer une chaîne de caractères, il faut utiliser le type `String`, et non `string` comme en C(++).

Certaines commandes permettent d'*organiser les informations* ou afficher des *caractères spéciaux* sur la console :

Caractère	Affichage
\n	Retour chariot
\t	Tabulation
\\	\
\"	«

TABLE 2.2 – Caractères de contrôle d'affichage

2.7 Les commentaires

Définition 4 *Les commentaires sont des éléments inscrits dans le code qui ne seront pas exécutés. Ils permettent aussi de générer la documentation :*
- *avec des commentaires en HTML*
- *avec des balises spécifiques à Java :* `@...`

Les commentaires serviront à générer la documentation avec la commande `javadoc`.

Exemple 2.7.1 *Commentaires dans un programme, qui composeront la documentation*

```
/** Exemple de classe simple
 * @author Toto
 * @version 1.0
 */

public class Hello {
        /** Methode principale qui affiche un message sur la
            sortie standard
         * @param args : arguments de la commande
         */
        public static void main(String[] args) {
                System.out.println("Hello World!");
        }
};
```

2.8 Les tableaux

Les tableaux sont la plus simple et la plus efficace solution pour stocker des éléments du même type.

ATTENTION: Un tableau est une entité de *taille fixe*, et celle-ci ne peut pas changer.

2.8.1 Tableaux à une dimension

Déclaration :

- `type[] tab1, tab2;`
- `type tab3[];`

Exemple 2.8.1 *Exemple d'une déclaration d'un tableau de 7 entiers :*

```
public class TestTab {
        public static void main(String[] args) {
                int[] t1 = [1,2,3,4,5};
                int[] t2;

                t2=t1; //t1 et t2 referencent le meme tableau

                for(int i=0; i<t2.length; i++) {
                        t2[i]++;
                }

                for(int i=0; i<t1.length; i++) {
                        System.out.println("t1[" + i + "] = " t1[i
                           ]);
                }
        }
}
```

Il est possible de créer *dynamiquement* un tableau grâce à l'instruction `new`. Ce type de création est utilisé lorsqu'on ne connait pas la taille du tableau au moment de l'écriture du programme, ou lorsqu'elle dépend du contenu d'une ou plusieurs variables.

```
import java.util.Random; //generateur de nombre aleatoire

public class TestTab {
        public static void main(String[] args) {
                int tab[];
                Random rand = new Random();
                final int MAX = 20; //declaration d'une constante

                int nb_aleatoire = rand.nextInt(MAX); //nombre
                   aleatoire
                tab = new int[nb_aleatoire]; //creation du tableau

                System.out.println("Taille du tableau: " + tab.
                   length);
```

```
                for(int i=0; i<tab.length; i++) { //on parcourt
                    tout le tableau
                        System.out.println("Contenu␣indice:␣" + i
                            + "␣>␣" + tab[i]);
                }
        }
}
```

2.8.2 Tableaux multidimensionnels

Les tableaux multidimensionnels sont des tableaux qui ont pour éléments des tableaux.

Exemple 2.8.2 *Exemple d'une déclaration d'un tableau multidimensionnel :*

```
int tab2D[][] = {
        {1,2,3}, {4,5,6}
};
```

2.9 Programme principal

Comme en C, il est indispensable d'avoir un programme principal pour pouvoir exécuter les *fonctions* ou *méthodes* créées. Cependant en Java, tout est objet : c'est la raison pour laquelle on ne peut pas définir une *méthode globale* `main`.

2.9.1 Code minimal

```
public class NomClasse {
        public static void main(String[] args) {

        }
}
```

2.9.2 Compilation

En Java, il est nécessaire de *compiler*[1] ses fichiers avant de pouvoir les exécuter.
Supposons le code suivant : (dans un fichier `Hero.java`)

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

compilation : `javac Hello.java`

exécution : `java Hello`

1. Traduire un programme en langage « machine »

Chapitre 3

Le modèle objet

A ce stade, nous pouvons constater que :
— La complexité des projets est revue à la hausse
— Il y a un réel besoin de gain de productivité
Nous devons alors nous préparer pour que le projet soit modulable et résistant aux modifications, réutilisable, lisible et compréhensible. La solution à cela est l'*objet*.

Définition 5 *Un objet est constitué de* données*. Ses données sont stockées dans les* attributs *de l'objet.*

Exemple 3.0.1 *Un rectangle possède deux attributs : sa longueur et sa hauteur.*

Définition 6 *Un objet manipule ses données pour effectuer des* opérations*. Les opérations d'un objet sont réalisées par l'exécution des* méthodes *correspondantes.*

Exemple 3.0.2 *Un rectangle peut se dessiner, se déplacer, se redimensionner, etc...*

Définition 7 (Classe) *La structure des* attributs *et des* méthodes *d'un objet sont décrits dans sa* classe*.*

3.1 Les classes

Une classe décrit un modèle de données (les *attributs*) et de comportement (les *méthodes*).
— Les attributs et méthodes décrits dans une classe sont appelés les *membres*.
— Une classe peut être vue comme le moyen de décrire de nombreux objets.

Chat : décrit les entités à 4 pattes, 1 queue, et sachant miauler

Voiture : décrit les entités ayant des portes, des roues et sachant rouler

Tout objet est l'*instance* d'une classe : instancié (créé) à partir du modèle décrit dans sa classe, cet objet possède les mêmes attributs et les mêmes opérations que les autres instances issues de la même classe.

Exemple 3.1.1 *Exemples d'instances de classes :*

Chat : *Garfield, Félix, etc...*

Jeu : *Overwatch, World of Warcraft, PUBG, StarCraft II*

PresidentRepublique : *Emmanuel* Macron*, François* Hollande*, Nicolas* Sarkozy

3.1.1 Fonctionnement et structure

Du point de vue de l'informaticien :
- Les classes correspondent au programme
  - description des structures de données (liste, nom, type des attributs)
  - description des opérations (liste, nom et algorithme des méthodes)
- L'informaticien écrit des programmes, donc des classes
- Chaque instance s'exécute conformément à sa classe
- On peut créer de nombreuses instances à partir de la même classe
- L'exécution d'un programme orienté objet correspond à un ensemble d'objets qui interagissent entre eux

Déclaration

Une classe se définit de la manière suivante :

```
visibilite class NomClasse {
        visibilite type nomAttribut;
        ...

        visibilite type nomMethode(parametres) {
                ...
        }
        ...
}
```

Exemple 3.1.2 *Exemple d'une classe* `Circle` *:*

```
public class Circle {
        private double x, y, rayon;

        /** Calcul de l'aire
         * @return l'aire du cercle
         */

         public double getArea() {
                 return Math.PI * Math.pow(rayon, 2);
         }

         /** Translation sur l'axe X
          * @param value: decalage de l'axe
          */

          public void translateX(double value) {
                  x = x+value;
          }
}
```

Instanciation

La création d'un objet implique que celui doit être *instancié* à l'aide de l'opérateur `new`. Grâce à cet opérateur, une nouvelle *instance* de cette classe est allouée en mémoire : il est alors possible de l'utiliser dans notre programme.

Exemple 3.1.3 *Reprenons l'exemple de notre classe* `Circle` *:*

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

        Circle c = new Circle(); //instanciation de l'objet de
            type Circle que l'on nommme c

        c.translateX(50); //on appelle une methode de la classe
            Circle, appliquee a c

        double aire = c.getArea();

}
```

3.1.2 Les constructeurs

Définition 8 *Un* constructeur *est une méthode de la classe qui a pour objectif d'initialiser l'objet en cours de création. Celui porte le nom de la classe et en retourne une instance. Il est déclenché par l'instruction* `new`*.*

Toutes les classes possèdent par défaut un constructeur sans paramètres. Cependant, il peut être redéfini.

La syntaxe pour définir un constructeur est la suivante : `nomClasse();`.

Remarque 11 *Une classe peut avoir plusieurs constructeurs. Le choix du constructeur sera effectué en fonction du nombre de paramètres renseignés.*

Exemple 3.1.4 *On suppose une classe* `Droite` *dans laquelle on souhaite définir 2 constructeurs : un par défaut, et l'autre avec des paramètres qui devront être renseignés lors de son appel.*

```
public class Droite {
        private Point p1, p2;

        public Droite() { //constructeur par defaut
                        this(0,0,0,0);
        }

        public Droite(int x1, int y1, int x2, int y2) { //
            constructeur avec parametres
                p1 = new Point();
                p1.setX(x1);
                p1.setY(y1);

                p2 = new Point();
```

```
                p2.setX(x2);
                p2.setY(y2);
        }
}
```

3.1.3 Constructeur par recopie

Le *constructeur par recopie*, comme son nom l'indique, permet de créer un nouvel objet avec les valeurs d'un autre objet **de la même classe**. Celui-ci prendra en paramètre l'objet à copier.

Exemple 3.1.5 *On souhaite créer un* `Carre` *à partir d'un autre déjà existant. La classe sera alors définie de cette manière :*

```
public class Carre {
        private Point point;
        private int longueur;

        public Carre(int x, int y, int longueur) { //constructeur
            avec parametres
                point = new Point();
                point.setX(x);
                point.setY(y);
                this.longueur = longueur;
        }
        public Carre(Carre carre) { //constructeur par recopie
                this.longueur = carre.longueur;
                this.point=newPoint();
                this.point.setX(carre.point.getX());
                this.point.setY(carre.point.getY());
        }

        public void setX(int x) {
                this.point.setX(x);
        }
}
```

Afin d'effectuer une copie d'un carré nommé `c1`*, nous devrons appeler le constructeur par recopie de la manière suivante :*

```
public static void main(String[] args) {
        Carre c1 = new Carre(5, 0, 10);
        Carre c2 = new Carre(c1);
        c2.setX(0);
}
```

3.2 Packages

Un package en Java regroupe un ensemble de classes sous un même *espace de nommage*. Point de vue de la compilation, le mot clé `package` permet d'indiquer à quel package appartient la ou les classe(s) de l'unité de compilation (le fichier).

ATTENTION: `package` doit être la première instruction de chaque fichier.
Les noms des packages suivent le schéma : `name.subname`.

Exemple 3.2.1 *`java.util`*

Si on souhaite utiliser des méthodes définies dans d'autres packages, il est nécessaire de : soit
— importer le package dans le fichier
— préfixer le nom de la classe (définie dans un autre fichier) par son nom de package

Remarque 12 *Vous pourrez remarquer que préfixer le nom de la classe par le nom du package peut être très long et fastidieux si le projet réalisé comporte beaucoup de fichiers. C'est la raison pour laquelle on privilégiera l'importation des packages.*

Exemple 3.2.2 (Importation du package) *On importe les fichiers `human.java` et `monster.java` qui se trouvent dans le dossier `game/characters`*

```
import game.characters.human;
import game.characters.monster;
```

Exemple 3.2.3 (Classe préfixée du nom de package) *On considère dans cet exemple que l'on souhaite créer un vecteur, qui a un constructeur déjà implémenté dans les fichiers du JDK. Il s'agit du package `java.util.Vector` :*

```
java.util.Vector v = new java.util.Vector();
```

Il existe un « raccourci » permettant d'utiliser toutes les classes se trouvant dans un même package.

Exemple 3.2.4 *On souhaite importer toutes les classes se trouvant dans le package `java.util`. La ligne de code sera la suivante :*

```
import java.util.*
```

ATTENTION: Seules les classes *publiques* d'un package sont utilisables dans un autre package !
Exemple de programme mettant en évidence ce système : (figure 3.1 page 21)

3.3 Accès aux membres

Comme dans d'autres langages (C++, etc...), la visibilité des membres (attributs, méthodes) est définie au niveau de la classe :

Inaccessible hors de l'objet : `private`

Inaccessible hors de la hiérarchie des classes : `protected`

Inaccessible en dehors du package : `friendly`

Accessibilité totale : `public`

Dans la majeure partie des cas, on essaiera toujours de limiter l'accès un minimum. (figure 3.2 page 21)

Circle : classe publique du package **graph**

```java
package graph;

public class Circle { // classe publique !

    private double x, y, rayon ;

    /** Calcul de l'aire
     * @return l'aire du cercle
     */
    public double getArea(){
        return Math.PI * Math.pow(rayon, 2) ;
    }

    /** Translation sur l'axe x
     * @param value : decalage sur l'axe
     */
    public void translateX(double value){
        x = x + value ; // x += value ;
    }

}
```

Autre package : utilisation du vrai nom de **Circle**...

```java
package tests;

public class Test1 {
    public static void main(String[] args) {

        // instanciation d'une classe d'un autre package
        graph.Circle c = new graph.Circle() ;

    }
}
```

... ou importation pour alléger l'écriture

```java
package tests;

import graph.Circle ;

public class Test1 {
    public static void main(String[] args) {

        // instanciation d'une classe d'un autre package
        Circle c = new Circle() ;

    }
}
```

FIGURE 3.1 – Exemple d'un programme utilisant des méthodes et variables d'autres packages

```java
package graph;

public class PointBizarre {

    int x ; // friendly (NB: devrait être privé)
    private int y ; // privé

    // Methode publique
    public void setXandY(int abscisse, int ordonnee){
        x = abscisse ;
        y = ordonnee ;
    }

}
```

```java
package graph ; // Même package que PointBizarre

public class RectangleBizarre {

    private PointBizarre coords = new PointBizarre();

    public int longueur, largeur ; // Devraient être privés !

    public void setCoords(int abscisse, int ordonnee){
        coords.x = abscisse ; // OK : x est friendly dans PointBizarre
        coords.y = ordonnee ; // PAS OK : y est privée dan PointBizarre
    }

}
```

```java
package tests ; // Package différent de graph

import graph.PointBizarre;
import graph.RectangleBizarre;

public class Test1 {
    public static void main(String[] args) {

        PointBizarre p = new PointBizarre() ;
        p.x = 10 ; // PAS OK : x friendly mais autre package
        p.y = 20 ; // PAS OK : y privé
        p.setXandY(10, 20); // OK : setXandY est publique !

        RectangleBizarre rectangle = new RectangleBizarre() ;
        rectangle.longueur = 50 ; // OK : longueur est public

    }
}
```

FIGURE 3.2 – Exemple d'un programme utilisant des attributs et méthodes d'autres classes

3.3.1 Getters et setters

Afin de simplifier et clarifier le programme principal ou les méthodes de chaque classe, nous pouvons implémenter des *getters* et des *setters*. Ils permettront, respectivement, de récupérer ou modifier une valeur d'un attribut de la classe. Il sera nécessaire d'utiliser l'*encapsulation* pour

contrôler les accès.

Exemple 3.3.1 (Getters et Setters) *Exemple d'un programme dans lequel des getters et setters ont été implémentés, et utilisés dans les méthodes :*

```
package graph;

import javafx.scene.paint.Color;

public class Point {
        public int x,y;
        private Color color = Color.BLACK;
        private boolean black = true;

        public int getX() {
                return x;
        }
        public int setX(int x) {
                this.x = x;
        }

        public int getY() {
                return y;
        }
        public int setY(int y) {
                this.y = y;
        }

        public Color getColor() {
                return color;
        }
        private void setColor(Color color) {
                this.color = color;
        }

        public boolean isBlack() {
                return black;
        }

        void setBlack(boolean black) {
                this.blavk = black;
                if(this.black) {
                        setColor(Color.BLACK);
                }
                else {
                        setColor(Color.WHITE);
                }
        }
}
```

Remarque 13 *Le mot clé `this` est utilisé par un objet pour se référencer lui-même.*

3.3.2 Encapsulation

Encapsuler les données permet de :
— les protéger des accès intempestifs
— déclencher des actions spécifiques lorsqu'on y accède

3.3.3 Identité

Chaque objet instancié peut avoir les mêmes valeurs dans leurs attributs ainsi que les mêmes méthodes. Ils ne se confondent pas avec les autres similaires.

Exemple 3.3.2 *Deux voitures de la même marque, le même modèle, les mêmes options et les mêmes jantes ne se confondent pas. Elles représentent deux entités identiques.*

Réciproquement, les valeurs contenues dans les attributs de chaque objet peuvent changer. L'objet ne changera pas l'*identité*.

Exemple 3.3.3 *Une voiture repeinte ou avec des options ajoutées (après sa conception).*

Le langage orienté objet fournit un moyen de désigner un objet en tant qu'élément unique. Une variable de type objet (ex : `Voiture v`) contient une *référence* ou `null`.
ATTENTION: Il est important de noter que chaque variable de type objet pointe vers un objet. Si cet objet a pour contenu `null`, alors la variable ne pourra pas être utilisée. Sinon, vous aurez des erreurs de compilation !

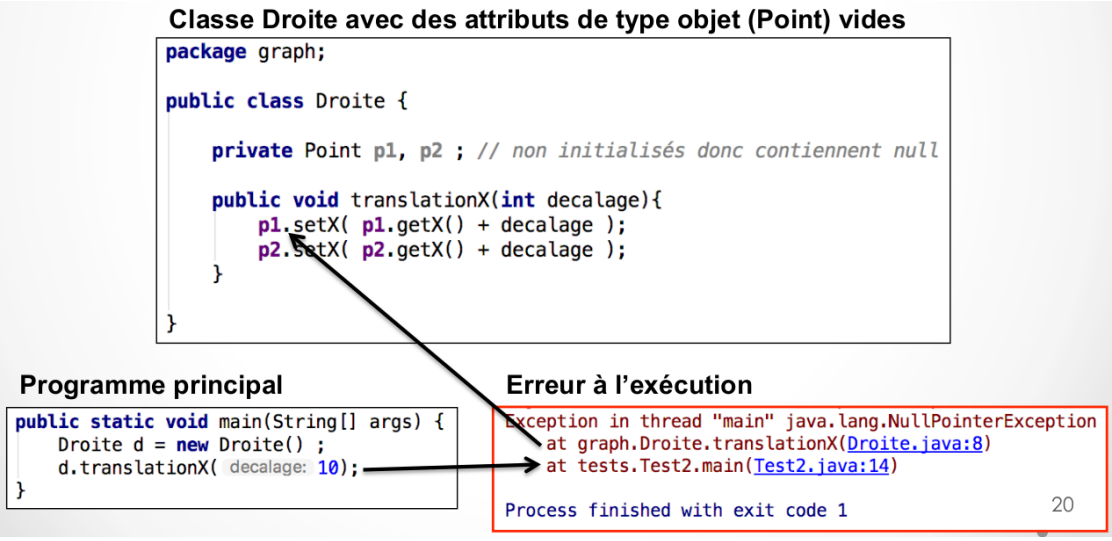


FIGURE 3.3 – Erreurs de compilation à cause d'un objet ayant pour valeur `null`

3.4 static

Proposition : Les classes sont des *modèles* (exemple 3.4.1 page 24), mais chaque modèle n'est pas une instance (exemple 3.4.2 page 24).

Exemple 3.4.1 (Modèle) *Un document papier contenant un plan pour créer un certain modèle des « Twingo ».*

Exemple 3.4.2 (Instance) *Le papier contenant le plan n'est pas une « Twingo ».*

Le modèle a ses propres attributs, qui n'ont aucun rapport avec les objets.

Exemple 3.4.3 *Le papier contenant le plan peut avoir une couleur :* `static Couleur couleur;` *Cette couleur n'est pas celle de la voiture. Ce modèle peut aussi avoir comme méthode :* `static plier();`.

Exemple d'une classe `Etudiant` :

```
public class Etudiant {
        private static int nombre_etudiants = 0;

        private String nom;
        private int numCarte;

        private static void nouvelleInscription() {
                nombre_etudiants++;
        }
        private static int getNbEtudiants() {
                return nombre_etudiants;
        }

        public Etudiant(String nom) {
                nouvelleInscription();
                this.nom = nom;
                this.numCarte = getNbEtudiants();
        }
        public String getNom() {
                return nom;
        }
        public int getNumCarte() {
                return numCarte;
        }
}
```

Programme principal :

```
public static void main(String[] args) {
        Etudiant sacha, helmut, omer;

        sacha = new Etudiant("Touille");
        helmut = new Etudiant("Hardelpik");
        omer = new Etudiant("Dalors");
```

```
        int NB = Etudiant.getNBEtudiants();
        System.out.println(NB);

        //affichage de l'etudiant "helmut"
        System.out.println(helmut.getNom());
        System.out.println(helmut.getNumCarte());
}
```

3.5 Héritage

L'*héritage* est la technique la plus utilisée pour réaliser la généralisation ou la spécialisation.

3.5.1 Principe

Créer des *sous-classes* permet d'utiliser les attributs, des méthodes et des contraintes de la classe dont elle dépend. On parle alors d'*héritage*.

Remarque 14 *Il est possible de créer des attributs et d'autres méthodes dans les sous-classes, mais ils ne seront pas utilisables plus haut dans la hiérarchie.*

Exemple 3.5.1 *Un carré est une figure géométrique. Nous pouvons alors définir une* classe mère *`Figure` qui aura pour* sous-classe *`Carre`. Nous pourrons aussi ajouter d'autres sous-classes : `Triangle`, `Polygone`, etc.*

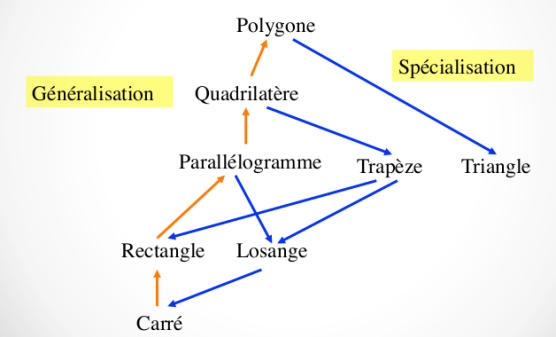


FIGURE 3.4 – Représentation schématique de l'héritage

ATTENTION: Une classe ne peut hériter que d'une seule classe.
Déclaration : `class nomClasse extends nomClasseMere { ... }`

Exemple 3.5.2 *Avec notre classe `Triangle` et notre classe `Figure`*

```
public class Triangle extends Figure {
        ...
}
```

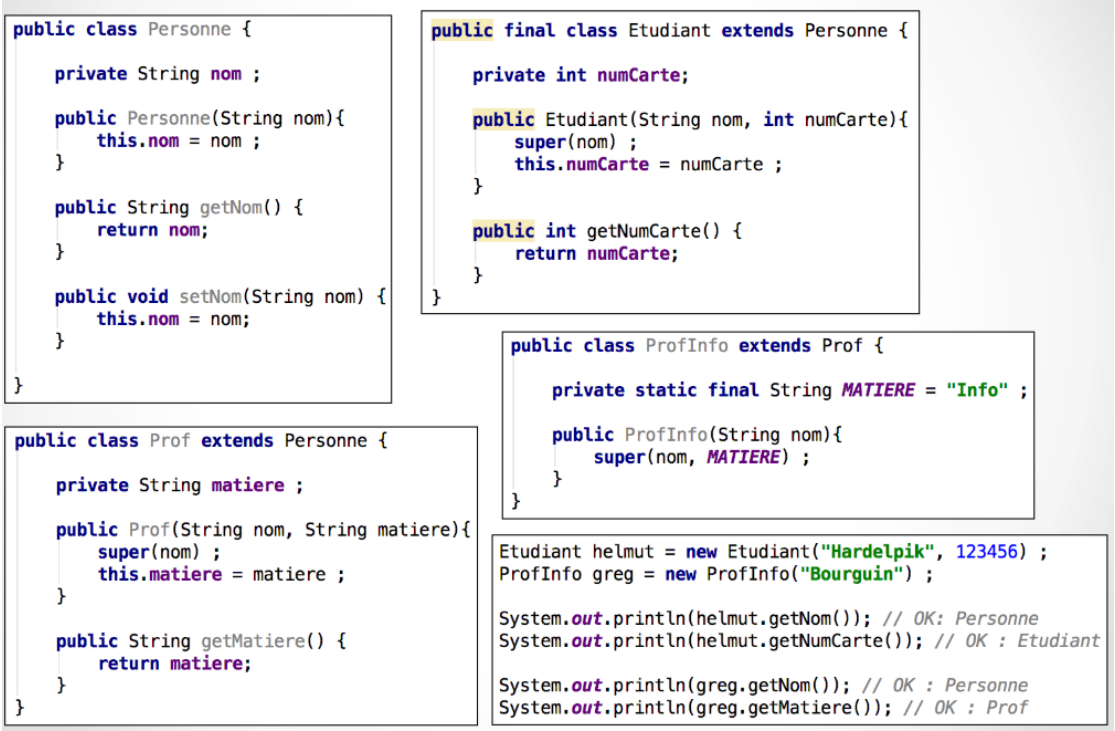


FIGURE 3.5 – Exemple héritage des classes

Il sera souvent utile d'accéder, depuis une classe, aux éléments de la classe mère. On utilisera pour cela le mot clé `super`.

Exemple 3.5.3 *On souhaite créer une classe mère* `Personne` *ayant pour sous-classe* `Etudiant`*. Afin d'éviter de déclarer plusieurs fois un attribut* `nom`*, on va directement affecter une valeur à l'attribut de la classe mère, plutôt que de le redéfinir dans les sous-classes.*

```
public class Personne { //classe mere
        public String nom;
        public String prenom;

        ...
}

public class Etudiant extends Personne {
        public int numCarte;

        public Etudiant(String nom, String prenom) {
                super(nom);
                super(prenom);
                this.numCarte = ... ;
        }
}
```

3.5.2 Surcharge

Principe

Définition 9 Surcharger *une méthode consiste à la redéfinir dans une sous-classe.*

ATTENTION: Une classe peut accéder aux membres de sa classe mère seulement si ils ont comme visibilité minimale `protected`.

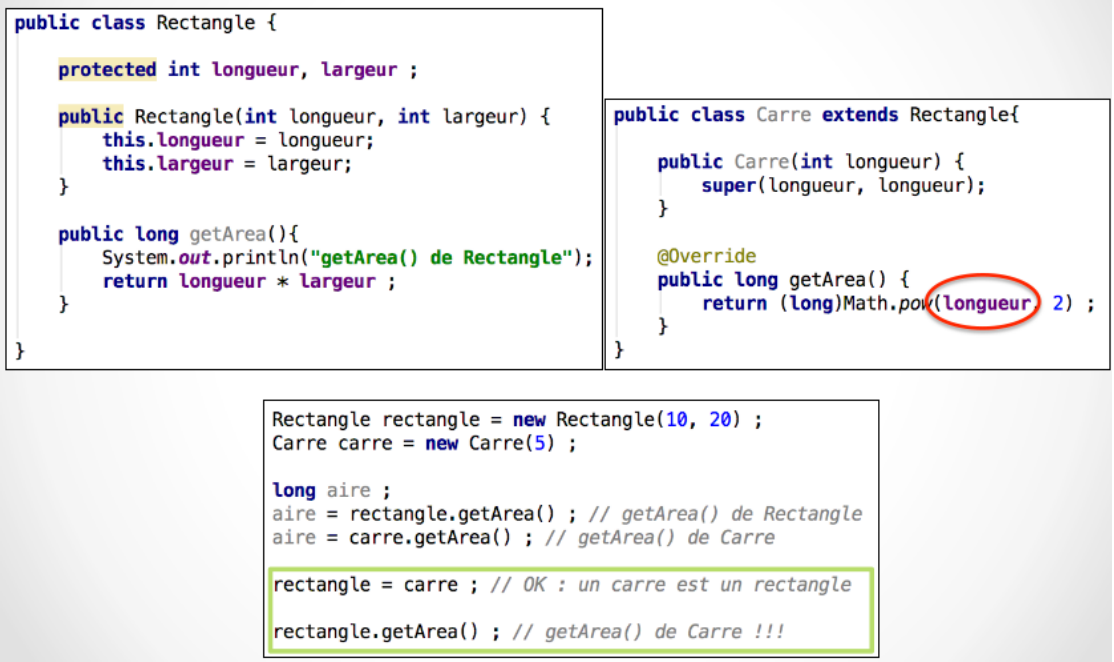


FIGURE 3.6 – Exemple d'un programme avec surcharge de méthodes

Surcharge de toString()

Toutes les classes héritant *implicitement* de `java.lang.Object`, la méthode `toString()`, si elle est redéfinie, sera donc *surchargée*.

Exemple 3.5.4 *Exemple d'une surcharge de la méthode `toString()` :*

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Locate;

public class Personne {
        private String nom, prenom;
        Calendar naissance;
        int numSecu;

        public Personne(String nom, String prenom, int numSecu) {
                this.nom = nom;
```

```
                this.prenom = prenom;
                this.numSecu = numSecu;
        }

        public void setNaissance(Calendar naissance) {
                this.naissance = naissance;
        }

        @Override
        public String toString() { //surcharge de toString()
                String chaine = String.format(Locale.FRANCE, "%s␣%
                    s␣[%3$tA␣%3$te␣%3$tB␣%3$tY]", prenom, nom.
                    toUpperCase(), naissance);
                return chaine;
        }

        //on implemente une fonction main de test
        public static void main(String[] args) {
                Personne greg = new Personne("Bourguin", "Gregory"
                    , 007);
                Calendar calendar = GregorianCalendar.getInstance(
                    Locale.FRANCE);
                greg.setNaissance(calendar);

                System.out.println(greg);
        }
}
```

final

Le mot clé `final` signifie que le changement est interdit. Il interdit notamment :
— la surcharge d'une méthode
— la spécialisation d'une classe
— la modification d'un attribut ou d'un argument d'une méthode

Annexe A

Pages bonus !

A.1 Formules mathématiques

Pour effectuer des calculs de probabilités, il est possible d'utiliser la formule de Bayes :

$$p(B_i/A) = \frac{p(A/B_i)p(B_i)}{\sum_{i=1}^{n} p(A|B_i)p(B_i)} \tag{A.1}$$

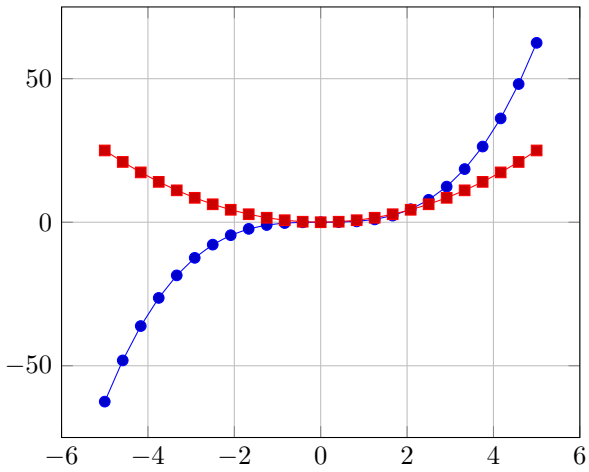
Pour un lancé de « pile ou face », l'univers Ω de cette expérience sera le suivant :

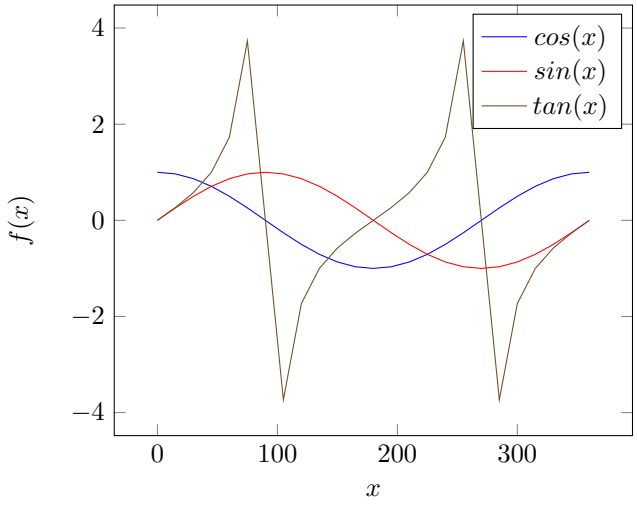
$$\Omega = pp, pf, fp, ff$$

La fonction de répartition de cette expérience sera donc :

$$F_X(x) \begin{cases} 0 & \text{si } x \in]-\infty, 0[\\ \frac{1}{4} & \text{si } x \in [0, 1[\\ \frac{3}{4} & \text{si } x \in [1, 2[\\ 1 & \text{si } x \in [2, +\infty[\end{cases}$$

A.2 Graphiques





A.3 Unités

La vitesse de la lumière est de $3 \times 10^8\,\mathrm{m\,s^{-1}}$.
Nous sommes le jeudi 3 mai 2018 et la température est de 13 °C.

A.4 Tableau de nombres

Températures de la journée
9,6
12
13,24
12,7835