



Programmation orientée objet en
Java

Visibilité, portée des déclarations et encapsulation des données

Dominique Blouin

Télécom Paris, Institut Polytechnique de Paris

dominique.blouin@telecom-paris.fr





Objectifs d'apprentissage

- **Visibilité des déclarations.**
- **Encapsulation des données.**
- **Portée des déclarations.**
- **Les classes `String` et `ArrayList`.**

Rappels

- Les **objets** sont des entités informatiques qui communiquent par envois de messages.
- Les objets contiennent des valeurs appelées **attributs**. Parmi les attributs, on peut trouver des **références** sur d'autres objets.
- Une référence sur un objet permet de lui envoyer un **message**.
- Pour chaque type de message que l'objet peut recevoir, l'objet connaît une **méthode** associée au type de message.
- Cette méthode est une procédure qui est **exécutée par l'objet** lorsqu'il reçoit le type de message associé.

Rappels

- Un type d'objet est décrit par une **classe**.
 - La classe décrit les attributs : nom et type de valeur.
 - La classe décrit les méthodes utilisées pour répondre aux messages.
- Le programmeur peut créer des objets à partir de la classe. C'est le processus d'**instanciation**.
- On dit que les objets sont des **instances** de la classe ou que les objets **appartiennent** à la classe.

Délégation et non-intrusion

- Une bonne pratique de l'OO est d'éviter d'agir depuis l'extérieur sur l'état d'un objet (**non-intrusion**).
- Ainsi, on évitera autant que possible les opérations de lecture et d'écriture **directes** des attributs.
- Dans le cas de notre exemple de la classe **Point**, on va donc **demander à l'objet** de type **Point** d'afficher ses coordonnées (principe de **délégation**).
- Tel que vu lors du cours précédent :

```
Point myPoint = new Point(10, 10);  
myPoint.writeCoordinates();
```

Bonne pratique

- Exemple : afficher les coordonnées d'un point.

```
class Point {  
    int xCoord;  
    int yCoord;  
    ...  
    void writeCoordinates() {  
        System.out.print("x = ");  
        System.out.println(xCoord);  
        System.out.print("y = ");  
        System.out.println(yCoord);  
    }  
}
```

- Rappel : pour **accéder** à un attribut ou à une méthode d'un objet, on utilise le caractère « . ».

```
Point myPoint = new Point(10, 10);  
myPoint.writeCoordinates();
```

Mauvaise pratique

- Accès direct aux attributs :

```
Point myPoint = new Point(10, 10);  
System.out.print("x = ");  
System.out.println(myPoint.xCoord);  
System.out.print("y = ");  
System.out.println(myPoint.yCoord);
```

- Question : comment **empêcher** cette mauvaise pratique en Java?
- Réponse : utiliser le mécanisme de **visibilité**.

Visibilité en Java

- La visibilité détermine quels **éléments d'une classe** (attributs, méthodes ou même les classes) sont **accessibles** par les autres classes.
- Quatre types de visibilité :
 - **public** : Tout est accessible.
 - **private** : Accessible que depuis la **classe**.
 - **package** (valeur par défaut si rien n'est spécifié) : Accessible que depuis les classes du même **package**.
 - **protected** : Accessible que depuis les classes d'**héritage** (notion à voir au prochain cours) ou du même **package**.

Visibilité **private**

- Exemple de visibilité **private** :

```
class Point {  
    private int xCoord;  
    private int yCoord;  
}
```

- L'utilisation de la visibilité **private** empêche d'accéder aux attributs depuis l'**extérieur** de la classe.
 - Sinon il y aura des **erreurs** de compilation.
- A l'intérieur de la classe, tel que dans la méthode **writeCoordinates()**, les attributs sont toujours accessibles.
- La visibilité peut également s'appliquer aux **méthodes**, incluant les **constructeurs**.
- La méthode **writeCoordinates()** sera déclarée de visibilité **public**, afin de la rendre accessible à toutes les classes.

```
Point myPoint = new Point(10, 10);
```

```
System.out.print("x = ");  
System.out.println(myPoint.xCoord);  
System.out.print("y = ");  
System.out.println(myPoint.yCoord);
```

```
public void writeCoordinates() {  
    System.out.print("x = ") ;  
    System.out.println(xCoord) ;  
    System.out.print("y = ") ;  
    System.out.println(yCoord) ;  
}
```

Encapsulation des données

- Les valeurs des attributs d'un objet représentent son **état**.
- Si les attributs d'un objet sont déclarés avec le mot-clé **private**, alors seul l'objet lui-même peut y accéder.
- On parle alors d'**encapsulation des données** dans la classe.

```
class Point {  
    private int xCoord;  
    private int yCoord;  
}
```

Encapsulation des données

- L'encapsulation des données est **fondamentale** en OO.
- Si les données ne sont pas encapsulées, n'importe quel autre objet peut les modifier.
 - Cela peut conduire à des incohérences.
- L'encapsulation des données concerne :
 - La **consultation** des données.
 - La **modification** des données.

Consultation des données

- Si l'on désire **connaître** la valeur d'un attribut d'un objet, il faut le **demander** à l'objet.
- Pour cela, on définit une méthode spécifique appelée un **getter**.
 - Car le nom de la méthode commence habituellement par **get**.
- Un getter peut aussi être appelé **accesseur** ou **méthode d'accès**.

Exemples de déclaration d'accesseurs

```
class Point {  
    private int xCoord;  
    private int yCoord;  
  
    public Point(int xCoord,  
                 int yCoord) {  
        this.xCoord = xCoord;  
        this.yCoord = yCoord;  
    }  
  
    public int getxCoord() {  
        return xCoord;  
    }  
  
    public int getyCoord() {  
        return yCoord;  
    }  
    ...  
}
```

Exemples d'utilisation d'accesseurs

```
Point myPoint = new Point(5, 7);  
int xCoord = myPoint.getxCoord();
```

Modification des données

- Pour modifier la valeur d'un attribut d'un objet, il faudra le lui **demander**.
- Pour cela, on définit une méthode spécifique appelée un **setter**.
 - Car le nom de la méthode commence généralement par **set**.
- Un setter peut aussi être appelé **mutateur** ou **méthode de modification**.

Exemples de déclaration de mutateurs

```
class Point {  
    private int xCoord;  
    private int yCoord;  
  
    public Point(int xCoord,  
                  int yCoord) {  
        this.xCoord = xCoord;  
        this.yCoord = yCoord;  
    }  
    ...  
  
    public void setxCoord(int xCoord) {  
        this.xCoord = xCoord;  
    }  
  
    public void setyCoord(int yCoord) {  
        this.yCoord = yCoord;  
    }  
}
```

Exemples d'utilisation de mutateurs

```
Point myPoint = new Point(5, 7);  
myPoint.setxCoord(9);
```

Pourquoi encapsuler ?

- L'encapsulation des données est un principe **très important**.
- Tous les attributs doivent être qualifiés avec le mot clé **private**, ce qui signifie que les attributs ne peuvent être consultés ou modifiés que depuis les **méthodes** de l'objet.
- Les raisons de ce principe sont multiples. Nous allons en voir quelques-unes.

Valider les valeurs des attributs

- Utiliser des **setters** permet de **contrôler la validité** des valeurs que l'on veut attribuer aux attributs.

```
class Person {  
    private int age;  
  
    ...  
  
    public boolean setAge(int age) {  
        if (age >= 0 && age < 150) {  
            this.age = age;  
            return true;  
        }  
        return false;  
    }  
    ...  
}
```

Cohérence des données

- L'encapsulation permet de préserver la **cohérence** et l'**intégrité** des structures de données.
- Exemple :

```
class Vector {  
    private int dx;  
    private int dy;  
  
    private double length;  
  
    public Vector(int dx, int dy) {  
        setDxDy(dx, dy);  
    }  
  
    public void setDxDy(int dx, int dy) {  
        this.dx = dx;  
        this.dy = dy;  
        length = Math.sqrt(dx*dx + dy*dy); // Maintain data consistency  
    }  
  
    public double getLenth() {  
        return length;  
    }  
}
```

Cohérence de l'application

- L'encapsulation de données aide à garantir la **cohérence d'une application**.
- Supposons que des classes de figures géométriques telles que la classe **Point** sont en cours d'édition dans un éditeur :
 - Alors toute modification des caractéristiques d'une figure, c'est-à-dire de ses attributs tels que sa position, doit entraîner un **rafraichissement** de l'affichage dans la fenêtre d'édition.
 - Sinon, l'application sera incohérente : la vue (l'affichage) ne reflètera pas le modèle (les données).
- Ce sont les méthodes de modification des attributs (mutateurs ou setters) qui seront en charge de **déclencher** un rafraîchissement de la fenêtre d'édition.
- A voir plus en détails lors du cours sur le patron de conception **MVC** (Modèle Vue Contrôleur).

Transparence des mises en œuvre

- Les méthodes d'une classe permettant de communiquer avec les objets de cette classe constituent une **interface** de l'objet avec le monde extérieur (les autres objets pour lesquels l'objet a une référence).
- Ces méthodes devront donc être **visibles** par tous les objets et seront donc qualifiées avec le mot clé **public**.
- Les attributs sont cachés et qualifiés avec le mot clé **private**.
- Si l'on encapsule correctement les données, il est possible de modifier la mise en œuvre des méthodes de manière **transparente**.

Exemple de transparence de mise en œuvre

Exemple précédent : l'encapsulation permet de préserver la cohérence et l'intégrité des structures de données.

```
class Vector {  
    private int dx;  
    private int dy;  
  
    private double length;  
  
    public Vector(int dx, int dy) {  
        setDxDy(dx, dy);  
    }  
  
    public void setDxDy(int dx,int dy) {  
        this.dx = dx;  
        this.dy = dy;  
        length = Math.sqrt(dx*dx + dy*dy);  
    }  
  
    public double getLength() {  
        return length;  
    }  
}
```

Exemple de transparence de mise en œuvre

- Supposons que l'on ait **rarement** besoin de connaître la longueur d'un vecteur.
- Alors on peut implémenter la classe différemment, de manière **transparente** :
 - Les interfaces des deux classes sont les **mêmes**.

```
class Vector {  
    private int dx;  
    private int dy;  
    private double length;  
  
    public Vector(int dx, int dy) {  
        setDxDy(dx, dy);  
    }  
  
    public void setDxDy(int dx, int dy) {  
        this.dx = dx;  
        this.dy = dy;  
        length = Math.sqrt(dx*dx + dy*dy);  
    }  
  
    public double getLength() {  
        return length;  
    }  
}
```

```
class Vector {  
    private int dx;  
    private int dy;  
  
    public Vector(int dx, int dy) {  
        setDxDy(dx, dy);  
    }  
  
    public void setDxDy(int dx, int dy) {  
        this.dx = dx;  
        this.dy = dy;  
    }  
  
    public double getLength() {  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
}
```

Transparence de mise en œuvre grâce à l'encapsulation des données

- Parce que les données ont été **encapsulées** dans la classe **Vector**, on peut utiliser l'une ou l'autre implémentation de la classe de manière **transparente**.
- Les deux classes implémentant la classe **Vector** ont la **même interface**, c'est-à-dire qu'elles présentent les **mêmes méthodes** aux objets qui les utilisent.
- Si les données sont correctement encapsulées dans une classe, l'utilisateur de la classe n'a pas besoin de savoir **comment sont structurées** les données et **comment les méthodes sont programmées** à l'intérieur des objets de la classe.

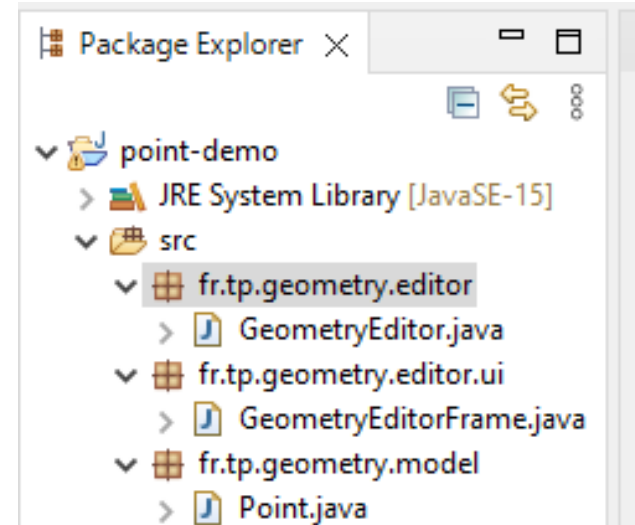
Visibilité package : organisation des classes

- Un **package** (paquetage en français) est une collection **logique** de classes.
- Dans un programme Java, les classes sont regroupées en **packages**.
 - Un package pour les classes réalisant des E/S (I/O).
 - Un package pour les classes réalisant des communications réseaux.
 - Un package pour les classes réalisant l'interface graphique (UI).
 - Etc.
- Principe de **cohésion forte** et de **couplage faible** :
 - Les classes devant être **utilisées et évoluer ensemble** (couplage fort) seront regroupées dans un **même package** (cohésion forte).
 - Les classes ayant **peu de dépendances entre elles** et liées à des **fonctionnalités différentes** seront regroupées dans des **packages différents**.
 - But : faciliter la **réutilisation** des classes.

Déclaration de package

```
package fr.tp.geometry.model;
```

```
class Point {  
    private int xCoord;  
    private int yCoord;  
  
    public Point(int xCoord,  
                 int yCoord) {  
        this.xCoord = xCoord;  
        this.yCoord = yCoord;  
    }  
    ...  
}
```



- Les packages sont **hiérarchiques**.
 - Un package peut également contenir d'autres packages.
- Le fichier de la classe (*Point.java*) sera stocké dans un sous-répertoire ayant la **même arborescence** que celle définie par le **nom du package**.
 - Les points (.) sont remplacés par des barres obliques (/).
 - ***../src/fr/tp/geometry/model/Point.java***
- Conventions de nommage :
 - Les noms de packages ne contiennent **que des lettres minuscules**.
 - On utilise souvent le **nom de domaine** de l'entreprise (*fr.tp*) pour identifier le package de **manière unique**.

Espace de nom de package

- Par défaut, une classe ne pourra accéder à une autre classe déclarée dans un autre package qu'en spécifiant le **nom complet** du package.

```
package fr.tp.geometry.editor;  
  
public class Main {  
  
    public Main() {  
    }  
  
    public static void main(String[] args) {  
        fr.tp.geometry.model.Point myPoint = new  
fr.tp.geometry.model.Point(10, 10);  
  
        myPoint.writeCoordinates();  
    }  
}
```

Import de package

- L'utilisation du nom complet de package rend le code difficile à lire.
- Pour solutionner ce problème, on peut utiliser le mot clé **import** :
`package fr.tp.geometry.editor;`

```
import fr.tp.geometry.model.Point;
```

```
public class Main {  
    public Main() {  
    }  
  
    public static void main(String[] args) {  
        Point myPoint = new Point(10, 10);  
        myPoint.writeCoordinates();  
    }  
}
```

Import de toutes les classes d'un package

```
package fr.tp.geometry.editor;

import fr.tp.geometry.model.*; // "*" imports all classes from package

public class Main {

    public Main() {
    }

    public static void main(String[] args) {
        Point myPoint = new Point(10, 10);

        myPoint.writeCoordinates();
    }
}
```

Visibilité de type package

- Si aucune visibilité n'est déclarée, la visibilité sera celle du **package**.
- Une classe ne pourra accéder à un élément de visibilité **package** d'une autre classe que si ces deux classes sont déclarées dans un **même package**.
- Si les classes sont dans deux packages différents, alors seulement les éléments déclarés avec une visibilité **public** seront accessibles.

```
package fr.tp.geometry.model;  
  
class Point {  
    int xCoord; // Package visibility  
    int yCoord;  
  
    public Point(int xCoord,  
                 int yCoord) {  
        this.xCoord = xCoord;  
        this.yCoord = yCoord;  
    }  
}
```

```
package fr.tp.geometry.model;  
  
class Circle {  
    private Point location;  
    private float radius;  
  
    public Circle(int xCoord,  
                 int yCoord,  
                 float radius) {  
        this.location = new Point(xCoord, yCoord);  
        this.radius = radius;  
  
        this.location.xCoord = 0; // Accessible  
    }  
}
```

Visibilité des classes

- On peut également définir la visibilité d'une **classe** :

```
public class Vector {  
    private int dx;  
    private int dy;  
  
    public void setDxDy(int dx, int dy) {  
        this.dx = dx;  
        this.dy = dy;  
    }  
  
    public double getLength() {  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
}
```

```
public class Point {  
    private int xCoord;  
    private int yCoord;  
  
    public Point(int xCoord,  
                 int yCoord) {  
        this.xCoord = xCoord;  
        this.yCoord = yCoord;  
    }  
    ...  
}
```

- Quelle est la signification de la visibilité d'une classe?

Portée des déclarations : trois types de déclaration de variables

```
public class Test {  
    private int myAttribute; // Déclaration d'attribut  
  
    public void myMethod(int myPar) { // Déclaration de parametre  
        ...  
        int myLocalVar = 0; // Déclaration de variable locale  
        ...  
    }  
}
```

- Ces déclarations ont une **portée**.
- Portée : **partie** du programme où la déclaration est **visible**.
 - Les **endroits dans le code** où la variable peut être **utilisée**.

Portée des déclarations

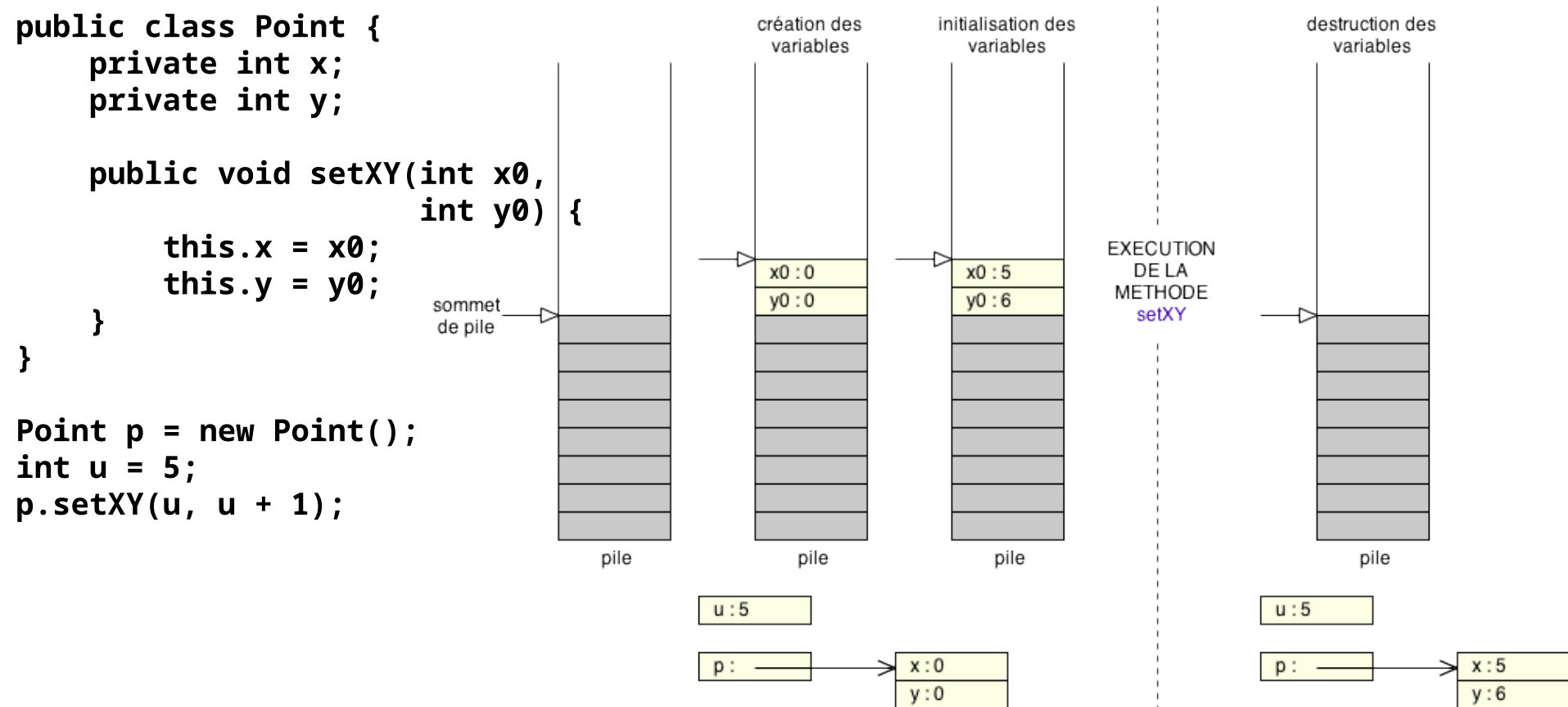
- Variable locale :
 - La portée est l'**intérieur du bloc d'instructions** à partir de la **ligne de la déclaration** de la variable.

```
{  
    int myLocalVar = 0;  
    myLocalVar = myLocalVar + 8;  
}
```
- Attribut :
 - La portée est la **classe entière**.
 - Un attribut est donc accessible dans **toutes les méthodes** de la classe.
- Paramètre de méthode :
 - La portée est la **méthode entière**.
 - La variable paramètre est **créée au début** de l'exécution de la méthode.
 - Elle **disparaît à la fin** de l'exécution de la méthode.

Passage de paramètres : comment ça se passe?

Java pratique le passage de paramètre **par valeur**.

- Effectué à l'aide d'une **pile**.



Les chaînes de caractères

- Chaîne de caractères : **suite de caractères**.
 - Par exemple le mot **Java**.
- En Java (et dans presque tous les langages comme en C), une chaîne de caractères s'écrit entre des guillemets (").

"Java"

"Java is easy"

- Les chaînes de caractères sont des objets de la classe **String**.
 - Elle est fournie par le JDK.

```
String str1 = null;    // Initialized with the null  
value
```

```
String str2 = "Java"; // Initialized with a reference
```

- Sur le web, rechercher « [JAVA SE String](#) »

Concaténation de chaînes de caractères

- L'opérateur **+** peut prendre deux chaînes de caractères en paramètres.

```
String str1 = "Java";  
String str2 = "coffee";  
String str3 = str1 + " " + str2;
```

- L'opérateur **+** réalise la concaténation de ses arguments:

```
String str4 = str1 + str2;
```

- Equivalent à un appel de la méthode **concat** :

```
String str4 = str1.concat(str2);
```

Retour sur l'opérateur d'égalité

- Nous avons vu l'opérateur de test d'égalité noté **x == y**.
- Celui-ci ne fonctionne qu'avec les types scalaires **discret** :
 - **char, byte, short, int, long** et **boolean**.
 - Il ne fonctionne pas avec les types **float** et **double**.
- Qu'en est-il des types références?
 - L'opérateur **==** utilisé avec un type référence signifie que les deux opérandes sont des références sur **le même objet en mémoire**.
- Pour tester que deux objets sont égaux au sens large, les classes proposent une méthode nommée **equals()**.
- Ainsi pour tester l'égalité de deux chaînes de caractères **str1** et **str2** (pour savoir si elles sont composées des mêmes caractères dans le même ordre), on utilisera **str1.equals(str2)**.

La classe `ArrayList<E>`

- Si **E** est une classe (par exemple **Robot**), un objet de la classe **`ArrayList<E>`** est une **liste de références** sur des objets de la classe **E**.

```
ArrayList<Robot> myRobots = new ArrayList<Robot>();
```
- Cette liste est vide lorsqu'elle est instanciée (elle ne contient aucun élément), mais elle s'allonge automatiquement lorsqu'on lui ajoutera des éléments.
- Les éléments de la liste sont indexés à partir de la valeur **0**.

La classe ArrayList<E>

- Le contenu de la liste est modifié par les méthodes suivantes :

```
boolean add(E eObject)
boolean add(int index, E eObject)
boolean remove(int index)
Etc.
```

- Attention aux erreurs lors de l'utilisation de ces méthodes :
 - Utiliser la méthode **size()** pour vérifier qu'il existe bien un élément à l'index spécifié.

```
Robot myRobot;
```

```
if (index > -1 && index < myRobots.size()) {
    myRobot = myRobots.get(index);
}
else {
    myRobot = null;
}
```

Les classes génériques

- La classe **ArrayList<E>** est ce que l'on appelle une classe **générique**.
- C'est une classe qui est **paramétrée** par une autre classe.
 - La classe **ArrayList<E>** est paramétrée par la classe **E**.
ArrayList<Robot>
- La **programmation** des classes génériques n'est pas au programme.
- Seule l'**utilisation** des classes génériques prédéfinies l'est.

La boucle for : syntaxe classique

```
ArrayList<Robot> myRobots = new ArrayList();  
myRobots.add(...);  
...  
for (int index = 0; index < myRobots.size(); index++) {  
    Robot myRobot = myRobots.get(index);  
    System.out.println(myRobot);  
    ...  
}
```

- Tout comme en langage C, la variable **index** va prendre successivement pour valeur les valeurs de 0 jusqu'à la taille de la liste – 1.
- La variable **myRobot** va prendre successivement pour valeur tous les éléments de la structure **myRobots**.
 - Références sur objets de type **Robot**.

La boucle for : syntaxe simplifiée

```
ArrayList<Robot> myRobots = new ArrayList();  
myRobots.add(...);  
...  
  
for (Robot myRobot : myRobots) {  
    System.out.println(myRobot);  
    ...  
}
```

- La variable **myRobot** va prendre successivement pour valeur tous les éléments de la structure **myRobots**.
 - Références sur objets de type **Robot**.
- Cette méthode de parcours est utilisable avec toutes les structures de données proposées par le JDK, à partir de la version 5.

Conclusion

- Introduction de la notion de **visibilité** des déclarations.
 - **public, private, package...**
- La visibilité est essentielle pour réaliser l'encapsulation des données.
- Importance de l'encapsulation des données..
- Portée des déclarations de variables
 - Attributs, paramètres de méthodes, variables locales...
- Les classes **String** et **ArrayList** qui vous seront très utiles pour le prochain TP.

Programmation des méthodes

- Afin de réaliser le projet, vous aurez besoin de connaître davantage la syntaxe de programmation impérative de Java.
 - Il existe plusieurs autres instructions de contrôle telles que **if**, **while**, **switch**, etc.
 - Heureusement, cette syntaxe est similaire à celle du langage C que vous connaissez déjà.
- Etant donné le peu d'heures dont nous disposons pour ce cours, vous devrez apprendre cette syntaxe par vous-même.
- Pour cela, vous pouvez lire une présentation du cours téléchargeable [ici](#).
- Également, de nombreuses ressources d'apprentissage existent sur Internet tel que le site suivant :
 - <https://www.w3schools.com/java/default.asp>.