



Programmation orientée objet et
temps réelle avec Java

Programmation des méthodes

Dominique Blouin

Ingénieur de recherche

Télécom Paris, Institut Polytechnique de Paris

dominique.blouin@telecom-paris.fr





Objectifs d'apprentissage

- **Tableaux et énumérations**
- **Programmation impérative des méthodes**
- **Itérateurs et patrons de conception**
- **Commentaires**

Tableaux

- Les tableaux (**arrays** en anglais) ne sont pas des objets mais une structure de données ordinaire que l'on retrouve dans tous les langages de programmation.
- Un tableau est une table de taille fixe contenant des valeurs d'un type donné.
 - Contrairement aux **ArrayList** qui sont des tables extensibles.
- On préférera l'utilisation des objets **ArrayList** à l'utilisation des tableaux.
- Un tableau peut contenir des valeurs scalaires de type **int**, **double**, **boolean**, etc.
- Ou bien des **références** sur des objets d'un type (classe) donné.

Tableaux

- La **dimension** d'un tableau est le nombre d'éléments qu'il peut contenir.
 - Elle est décidée à la **création** du tableau.
- Les éléments d'un tableau de dimension n sont numérotés de 0 à $n-1$.
- Le numéro d'un élément dans un tableau est appelé son **indice**.
- Pour accéder à un élément d'un tableau, on fait suivre le nom de la variable de type tableau par l'indice entouré par des crochets.

Exemples d'utilisation des tableaux

- Si l'on a déclaré et initialisé la variable `names` :

```
String[] names = new String[10];
```
- On peut accéder à l'élément d'indice 7 avec la notation `names[7]` :

```
System.out.println("Element with index 7 is " +  
names[7]);  
names[7] = "Java";  
System.out.println("Now, it is " + names[7]);
```
- Rappel : Il est préférable d'utiliser des **objets** tels que `ArrayList`.

Enumération

- Type permettant de définir un ensemble fini de **constantes**.

- Exemple d'énumération déclarée dans un fichier:

```
public enum WeekDay {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}
```

- Peut également être déclarée dans une autre classe:

```
public class Week {  
  
    public static enum WeekDay {  
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
    }  
  
    public Week() {  
    }  
}
```

Utilisation des énumérations

- Enumeration attribute type:

```
public class Day {  
  
    private WeekDay weekDay;  
    private Date date;  
  
    public Day(WeekDay weekday,  
              Date date) {  
        this.weekDay = weekday;  
        this.date = date;  
    }  
    ...  
}
```

- Instantiation:

```
new Day(WeekDay.MONDAY, new Date(2023, 11, 13));
```

Programmation des méthodes

- En l'O.O., un programme est constitué d'**objets** qui s'envoient des **messages**.
- Lorsque l'on programme une **méthode** de l'objet, on programme des **instructions**, que l'objet doit exécuter lorsqu'il reçoit le message auquel correspond la méthode.
- Ces instructions peuvent être:
 - Des instructions **simples** :
 - Affectation d'une variable.
 - Envoi de message.
 - Terminaison d'une méthode.
 - Des instructions **composées** :
 - Exécution conditionnelle d'une instruction.
 - Itération contrôlée d'une instruction.
 - Séquencement d'instructions (bloc d'instructions).
- Les instructions composées sont aussi appelées **structures de contrôle** car elles permettent de contrôler **ce** qui doit être exécuté et **quand** cela doit être exécuté.

Variables de méthode

- Les variables locales aux méthodes ne sont **pas initialisées** par Java.
 - Quand une attribut est déclaré, il est initialisé comme nous l'avons vu.
- C'est le programmeur qui est responsable de l'initialisation des variables locales.
- Exemple :

```
int myIntVar = 0;  
String myString = null;
```

Affectation des variables

- Il est possible de **modifier** la valeur d'une variable ou d'un attribut en utilisant l'**instruction d'affectation**.
- Si **myVar** est une variable ou un attribut de type **T** et si **E** est une expression dont la valeur a le même type **T**, alors on peut écrire une instruction d'affectation :
`myVar = E;`
- Lorsqu'il exécute cette instruction, l'objet **calcule** la valeur de **E** et la variable **myVar** prend cette valeur.
- Notons que **T** peut être un type scalaire ou un type référence.
- Le compilateur est vigilant et vérifiera la **compatibilité** des types.

Exemples d'instructions d'affectation

```
int myFirstVar = 2 + 3;
```

```
int mySecondVar = 0;
```

```
mySecondVar = myFirstVar * 2;
```

```
boolean myBoolVar = (myFirstVar == mySecondVar);
```

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

```
Point mySecondPoint = new Point(23,24);
```

```
Vector myVector = myFirstPoint.vectorTo(mySecondPoint);
```

D'autres instructions d'affectation

`a += b; // Equivalent to a = a + b`

`a -= b; // Equivalent to a = a - b`

`a *= b; // Equivalent to a = a * b`

`a /= b; // Equivalent to a = a / b`

`a %= b; // Equivalent to a = a % b`

`a++; // Equivalent to a += 1 or a = a + 1`

`a--; // Equivalent to a -= 1 or a = a - 1`

L'instruction return

- Le corps des méthodes est obligatoirement un **bloc d'instructions**.
- Lorsqu'une méthode doit fournir une réponse, cela est spécifié dans l'en-tête de la méthode en écrivant un **type de retour** plutôt que le mot clé **void**:

```
private double length;  
private double height;  
...  
public double getSurface() {  
    ...  
    return length * height;  
};
```

- Pour spécifier la valeur de la réponse, on utilise l'instruction **return** suivie de la valeur de la réponse.
- La valeur de réponse est une **expression** dont l'évaluation doit être une valeur du type déclaré dans l'en-tête de la méthode.

Exécution de l'instruction `return`

- L'exécution de l'instruction `return` dans une méthode provoque la **fin de l'exécution** de la méthode.
- Si l'en-tête de la méthode spécifie un **type de retour**, l'exécution de la méthode doit toujours se terminer par une instruction `return` appropriée.
- Le compilateur est vigilant sur ces deux points :
 - Si une instruction est placée après une instruction `return`, elle ne pourra jamais être exécutée et le compilateur le signalera.
 - Si l'exécution d'une méthode ayant un type de retour se termine sans avoir exécuter une instruction `return`, le compilateur le signale également.

L'instruction conditionnelle if

- Une **instruction conditionnelle** est caractérisée par le mot clé **if** et sa syntaxe est:

```
if (condition) {  
    instruction1;  
    ...  
    instructionN;  
}
```

- La **condition** doit être une expression dont l'évaluation retournera une valeur booléenne.

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

L'instruction alternative if-then-else

```
if (condition) {  
    instructionSiVrai1  
    ...  
    instructionSiVraiN  
}  
else {  
    instructionSiFaux1  
    ...  
    instructionSiFauxN  
}
```

■ Exemple:

```
boolean setAge(int newAge) {  
    boolean changed;  
  
    if (newAge >= 0 && newAge < 150) {  
        changed = true;  
    }  
    else {  
        changed = false;  
    }  
  
    return changed;  
}
```

L'instruction d'itération `while`

```
while (condition) {  
    instruction1  
    ...  
    instructionN  
}
```

- Suite d'instructions courante:

```
index = 0;
```

```
while (index < maxIndex) {  
    ...  
    ...  
    index = index + 1;  
}
```

- Tous les langages de programmation proposent une instruction d'itération nommée `for` qui est plus compacte mais strictement équivalente.

La boucle for classique

- La forme générale de l'instruction d'itération `for` est :

```
for (initialisation; condition; nextStep) {  
    instruction1  
    ...  
    instructionN  
}
```

- Elle est équivalente à:
`init; // intialisation`

```
while (condition) { // tester la condition  
    instruction1  
    ...  
    instructionN  
    nextStep // préparer l'itération suivante  
}
```

Exemple de boucles équivalentes

```
int index = 0;
```

```
while (index < maxIndex) {  
    ...  
    ...  
    index++;  
}
```

```
for (int index = 0; index < maxIndex; index++) {  
    ...  
    ...  
}
```

L'instruction de traitement par cas `switch`

- Il est possible de programmer un traitement par cas à l'aide de l'instruction `switch`.
- Dans cette instruction, on considère une expression de type **scalaire** ou **énumération**.
 - Ne sont pas concernés les objets et les nombres flottants `float` et `double`.
- L'instruction `switch` permet d'exécuter des instructions différentes selon la valeur de l'expression de cas (**case**).

Exemple

- Dans cet exemple, nous distinguons les cas 1, 2 et les autres valeurs.

```
public void drawFigure(int numberOfSides) {  
    switch (numberOfSides) {  
        case 1:  
            drawPoint();  
  
            break;  
  
        case 2:  
            drawLine();  
  
            break;  
        default:  
            drawPolygone(numberOfSides);  
  
            break;  
    }  
}
```

- L'instruction `switch` permet d'éviter de coder des tests d'égalité en série.

Les instructions `break` et `return`

- Dans cet exemple, nous regroupons plusieurs valeurs dans un seul cas.

```
public String letterType(char letter) {
    switch (letter) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'y':
            return "voyelle";
        default:
            return "consonne";
    }
}
```

- **ATTENTION !** L'instructions `return` (ou `break`) que l'on trouve dans les clauses `case` et `default` permet de sortir de l'instruction `switch`. Si le programmeur l'oublie, l'exécution se poursuit dans la clause suivante.

Les itérateurs

- Un **itérateur** est un objet générique qui permet de parcourir une **collection** de données.
- Toutes les collections proposées par Java ont une méthode qui permet de générer un itérateur.
- En particulier, la classe **ArrayList<E>** a une méthode:

```
public Iterator<E> iterator() {  
    ...  
}
```
- Recherche : [JAVA SE ArrayList](#)

Les itérateurs

- Un objet de type **Iterator<E>** permet de parcourir la collection de données dont il est issu en accédant successivement à chacun des éléments de type **E** de la collection.

- Cela se fait de manière transparente à l'aide de deux méthodes :

```
boolean hasNext(); // Retourne true s'il y a encore des éléments à parcourir, sinon false
```

```
E next(); // Retourne l'élément suivant de l'itération
```

- Nous pouvons donc parcourir une collection **ArrayList** sans avoir à coder la gestion de l'indexation des objets de la collection.

Exemple avec la classe promotion vue en TP

```
public int newId() {
    int maxId = -1;

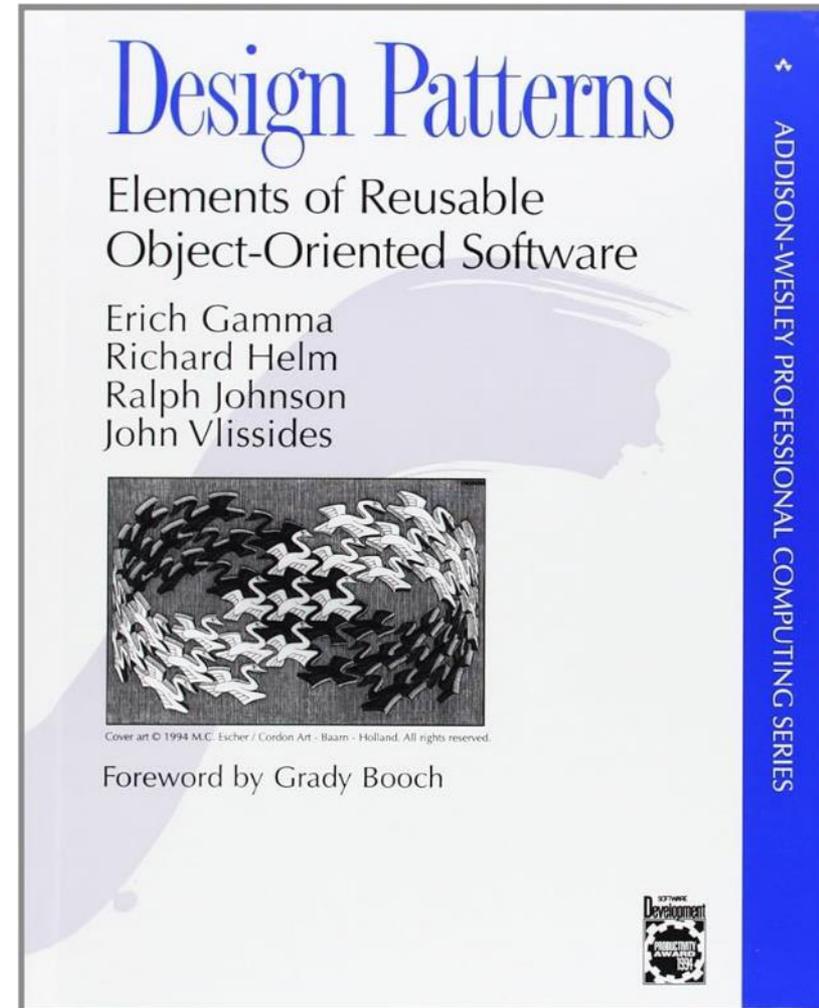
    Iterator<Student> studentsIt = studentList.iterator();
    while (studentsIt.hasNext()) {
        Student currentStudent = studentsIt.next();

        if (student.getId() > maxId) {
            maxId = student.getId();
        }
    }

    return maxId + 1;
}
```

Patrons de conception (design patterns)

- L'itérateur que nous venons de voir est ce qu'on appelle un patron de conception (**design pattern**).
- L'itérateur que nous venons de voir est ce qu'on appelle un patron de conception (**design pattern**).
- Design pattern: Solution **réutilisable** d'un problème fréquent lors de la conception de logiciels.
 - S'applique au paradigme de programmation OO et est donc applicable à **plusieurs language**.
- Popularisé par la bande des quatre (Gang of Four, GoF) en 1994.
- Permet de rapidement comprendre des programmes développés par d'autres développeurs qui ont aussi utilisé les design patterns.
 - Terminologie commune.
- Lors de ce cours, nous allons introduire quelques **design patterns simples** selon le besoin.



Les commentaires dans le code

- Dans un programme, il est souvent utile d'écrire des commentaires qui documentent le programme.
- Ce ne sont pas des instructions mais des **annotations textuelles** destinées aux programmeurs qui voudraient comprendre comment fonctionne le programme.
- Les commentaires sont utiles pour un autre programmeur qui désirerait travailler sur le même programme.
- Ils sont également utiles au programmeur lui-même lorsqu'il doit se relire.

Les commentaires dans le code

- Les commentaires doivent être concis, précis et utiles !
- Les commentaires ne doivent pas dire ce que font les instructions car la lecture des instructions permet de déterminer cela si le programme a été bien écrit.
- Les commentaires doivent décrire le **pourquoi** et le **comment**.

Les commentaires dans le code

- Tout ce qui se situe entre un slash-étoile (/*) et le prochain étoile-slash (*/) est du commentaire. Exemple:

```
int count = 0; /* The counter for photons */
```

- Tout ce qui se situe sur une ligne entre deux caractères slash (//) et la fin de la ligne est du commentaire. Exemple:

```
count = count + 1; // There is a photon detection
```

- Préférable et sans commentaires : des **noms explicites** !

```
int photonsCount = 0;
```