

# Guide

Formation Java et WPILib

Étienne Beaulac  
Ultime FRC 5528

Dernière modification  
29 mai 2018





# Table des matières

<b>Table des extraits de code</b>	<b>iii</b>
<b>Table des figures</b>	<b>iv</b>
<b>I Les bases de Java</b>	<b>1</b>
<b>1 Introduction au Java</b>	<b>2</b>
1.1 Les langages de programmation . . . . .	2
1.2 Qu'est-ce que le Java ? . . . . .	2
<b>2 Votre premier programme</b>	<b>4</b>
2.1 L'IDE Eclipse . . . . .	4
2.2 Création du projet . . . . .	5
2.3 Les instructions . . . . .	7
2.4 Les chaînes de caractères . . . . .	7
2.5 La méthode <code>println()</code> . . . . .	7
2.6 L'indentation . . . . .	7
2.7 Les commentaires . . . . .	8
2.7.1 Les commentaires standards . . . . .	8
2.7.2 Les commentaires Javadoc . . . . .	9
<b>3 Variables et opérateurs</b>	<b>11</b>
3.1 La déclaration de variables . . . . .	11
3.2 Lire la console . . . . .	13
3.3 Les variables de type primitif . . . . .	15
3.4 Les constantes . . . . .	17

3.5	Les opérateurs arithmétiques . . . . .	17
3.6	Les opérateurs d'affectation . . . . .	20
3.7	La librairie Math . . . . .	21
<b>4</b>	<b>Les instructions conditionnelles</b>	<b>23</b>
4.1	Le <i>if else</i> . . . . .	23
4.2	Les opérateurs de comparaison et les opérateurs logiques . . . . .	24
<b>II</b>	<b>La programmation orientée objet</b>	<b>26</b>
<b>III</b>	<b>La librairie WPILib</b>	<b>27</b>
<b>5</b>	<b>La classe TimedRobot</b>	<b>28</b>
5.1	Création d'un projet . . . . .	28
5.2	Les méthodes Init et Periodic . . . . .	28
5.3	Faire avancer le robot . . . . .	30
5.3.1	La classe VictorSP . . . . .	31
5.3.2	La classe DifferentialDrive . . . . .	31
5.3.3	La classe Joystick . . . . .	31

## Table des extraits de code

2.1	Programme de base . . . . .	6
2.2	Programme de base avec commentaires . . . . .	8
2.3	Ajout de commentaires Javadoc . . . . .	9
3.1	Utilisation d'une variable String . . . . .	12
3.2	Demander et afficher un nom . . . . .	14
3.3	Affichage de variables primitives . . . . .	15
3.4	Demande de l'âge et de la taille . . . . .	16
3.5	Années avant la majorité . . . . .	18
3.6	Liquidation d'un inventaire . . . . .	19
3.7	Utilisation de la classe Math . . . . .	22
4.1	Validation d'une année de naissance . . . . .	25
5.1	Utilisation de DifferentialDrive avec Joystick . . . . .	30

# Table des figures

1.1	Le processus de compilation. . . . .	2
2.1	L'interface principale de Eclipse. . . . .	5
2.2	Compiler, exécuter et déboguer un programme avec Eclipse. . . . .	6
2.3	Écriture dans la console. . . . .	6
2.4	Visualisation de la Javadoc dans Eclipse . . . . .	9
3.1	Une variable contenant l'âge de l'utilisateur en mémoire. . . . .	11
3.2	Les types primitifs les plus utilisés. . . . .	15
3.3	Les opérateurs arithmétiques. . . . .	18
3.4	Les opérateurs d'affectation. . . . .	21
4.1	Les opérateurs de comparaison et les opérateurs logiques. . . . .	24
5.1	Séquence d'exécution des méthodes de la classe Robot. . . . .	29

# **Première partie**

## **Les bases de Java**

# Chapitre 1

## Introduction au Java

### 1.1 Les langages de programmation

La programmation, en somme, est l'art de formuler ses algorithmes de manière à les faire comprendre à un ordinateur (Ada Lovelace, vers 1840<sup>1</sup>). Cependant, à la base, les ordinateurs ne comprennent que le binaire (Alan Turing, 1936<sup>2</sup>). Pour se simplifier la vie, les informaticiens ont créé des langages intermédiaires qui font le pont entre nous et les ordinateurs (Grace Hopper, 1951<sup>3</sup>). Tous les langages de programmation ont le même but : vous permettre de parler à un ordinateur plus simplement qu'en binaire.



FIGURE 1.1 – Le processus de compilation.

### 1.2 Qu'est-ce que le Java ?

Le langage Java a été créé , entre autres, par James Gosling, Patrick Naughton et Mike Sheridan, tous les trois employés chez *Sun Microsystems* dans les années 1990. Sa première version parut en 1995. Java est maintenant propriété de *Oracle Corporation*.

---

1. On attribue à Ada Lovelace, mathématicienne britannique, la création des premiers programmes informatiques. Ils furent conçus pour être exécutés sur la machine analytique de William Babbage, entièrement mécanique.

2. Alan Turing, mathématicien, cryptologue et logicien britannique, formalisa en 1936 le concept mathématique de *machine de Turing*.

3. Grace Hopper, informaticienne et *rear admiral (lower half)* de l'armée américaine, conçut en 1951 *A-0 System*, le premier compilateur pour ordinateur.



Java est un langage presque entièrement **orienté objet**. Il reprend une grande partie de la syntaxe du C/C++, tout en y ajoutant certaines fonctionnalités : une librairie standard très complète, la réflexivité, les expressions lambdas, l'*autoboxing* et l'*unboxing*, les interfaces, et plusieurs autres. Toutefois, les pointeurs et l'héritage multiple ne sont pas supportés. Ils ajouteraient une trop grande complexité au langage, alors que le but de Java est d'être simple, sécuritaire et robuste.

Le Java compte un nombre impressionnant d'utilisateurs. Une de ses forces est d'ailleurs sa portabilité. Tout programme Java, une fois compilé en *bytecode*, peut fonctionner sur n'importe quelle machine, tant qu'une machine virtuelle Java (JRE, ou *Java Runtime Environment*) y est installée.

# Chapitre 2

## Votre premier programme

*Manuel de référence : p. 1 à 22 et 33 à 38.*

### 2.1 L'IDE Eclipse

Pour programmer, il est préférable d'utiliser un bon environnement de développement (**IDE**, ou *Integrated Development Environment*). De tels logiciels comprennent un **éditeur de texte**, un **compilateur** et un **débogueur**. Nous utiliserons l'IDE Eclipse <sup>1</sup> avec l'extension WPILib fournie par FIRST.

Eclipse est disponible gratuitement sur [eclipse.org](http://eclipse.org). Vous devrez également vous assurer d'avoir installé une version récente du **JDK** (*Java Development Kit*). Les étapes d'installation sont également détaillées [ici](#).

Eclipse est un logiciel ayant plusieurs fonctionnalités. On peut d'ailleurs lui en ajouter à l'aide d'extensions (*plugins*), comme celle que nous utiliserons pour développer sur le roboRIO. Voici les fenêtres qui nous intéresseront le plus :

<b>Package Explorer</b>	Cette fenêtre regroupe tous vos projets, subdivisés en dossiers et paquetages ( <i>packages</i> ), jusqu'aux fichiers Java.
<b>Fenêtre d'édition</b>	Cette fenêtre affiche tous les fichiers que vous êtes en train d'éditer, vous permettant facilement de naviguer entre différents documents.
<b>Problems</b>	Comme son nom l'indique, on y retrouve une liste de tous les avertissements et erreurs concernant votre code. Chaque item précise la nature de l'erreur et où elle se trouve.
<b>Console</b>	La console est un outil essentiel, c'est le premier lien entre vous et l'exécution de votre programme. Vous pourrez y afficher du texte et en insérer.
<b>Javadoc</b>	Java a l'avantage de fournir son propre outil de documentation. Il suffit de cliquer sur un mot (classe, variable, méthode, etc.) et sa description y apparaîtra. Nous verrons plus loin comment créer ses propres entrées pour Javadoc.

---

1. Pour plus d'information concernant Eclipse, consultez ...

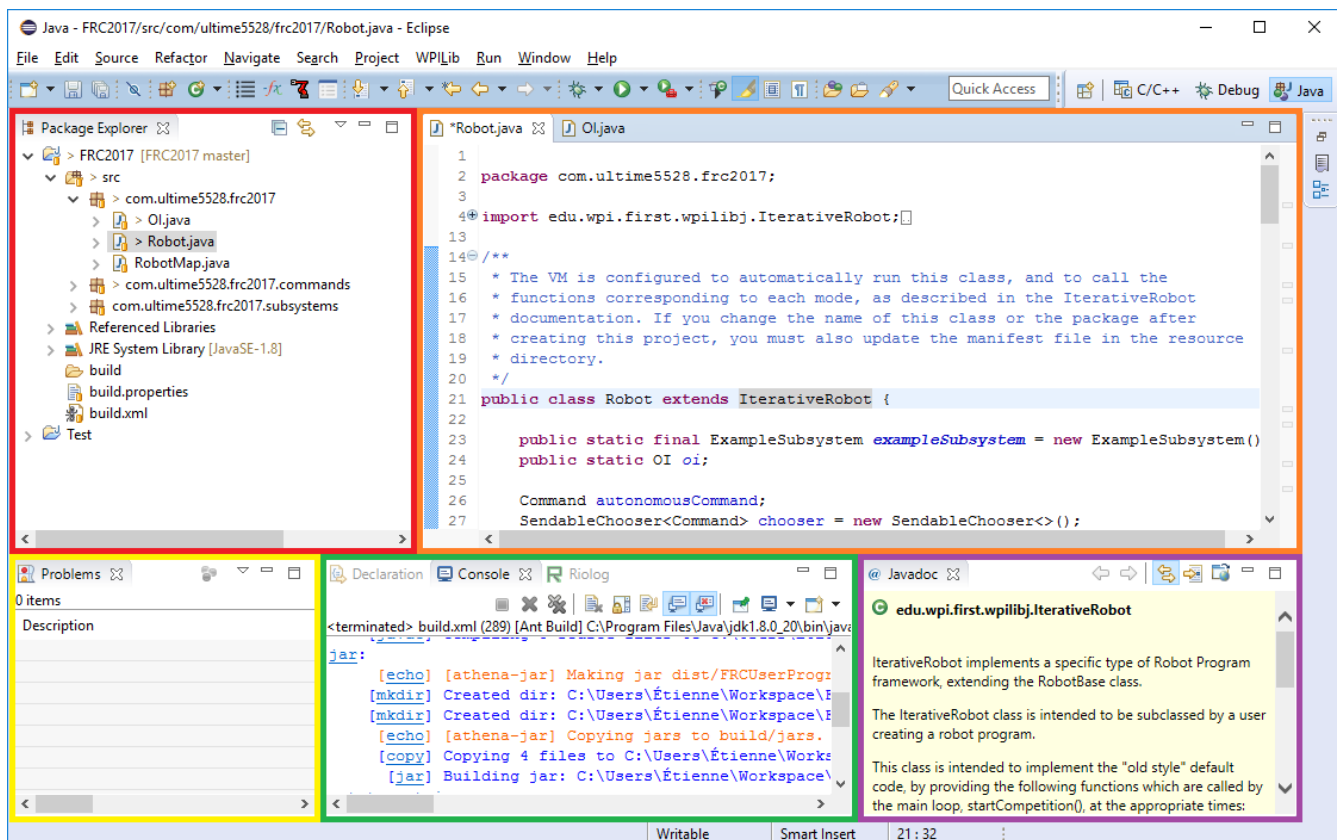


FIGURE 2.1 – L'interface principale de Eclipse.

Évidemment, toute l'interface est entièrement personnalisable. À vous de l'adapter comme il vous plaira !

## 2.2 Création du projet

1. Dans Eclipse, créez un nouveau projet avec **File > New > Java Project** . Donnez un nom à votre projet, puis cliquez sur **Finish** .
2. Ajoutez une classe à votre projet : **Clic droit sur votre projet > New > Class** . Donnez un nom à votre classe, cochez l'ajout de la méthode **main** , puis cliquez sur **Finish** .
3. Complétez le corps de la méthode avec l'exemple suivant, puis compilez et exécutez votre programme.

## CODE 2.1 — Programme de base

```
1 public class MonPremierProgramme {  
2  
3     public static void main(String[] args) {  
4  
5         System.out.println("Hello, world");  
6  
7     }  
8  
9 }
```

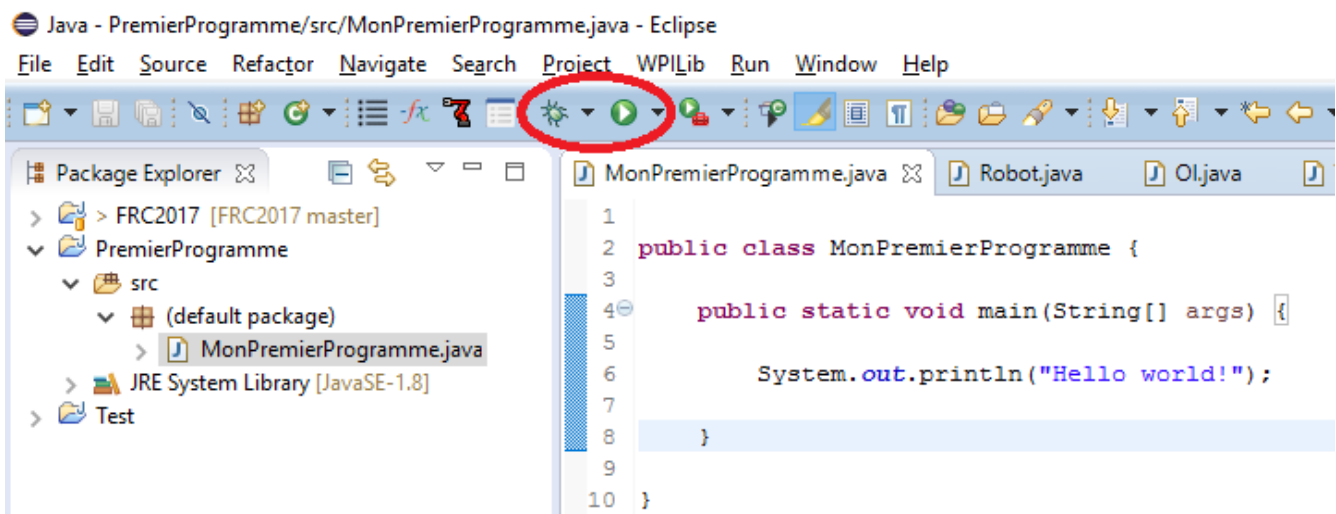




FIGURE 2.2 – Compiler, exécuter et déboguer un programme avec Eclipse.

Le bouton  vous permet de lancer votre programme en mode débogage. La flèche verte , quant à elle, compile et exécute. Après avoir cliqué dessus, vous devriez voir apparaître du texte dans votre console.

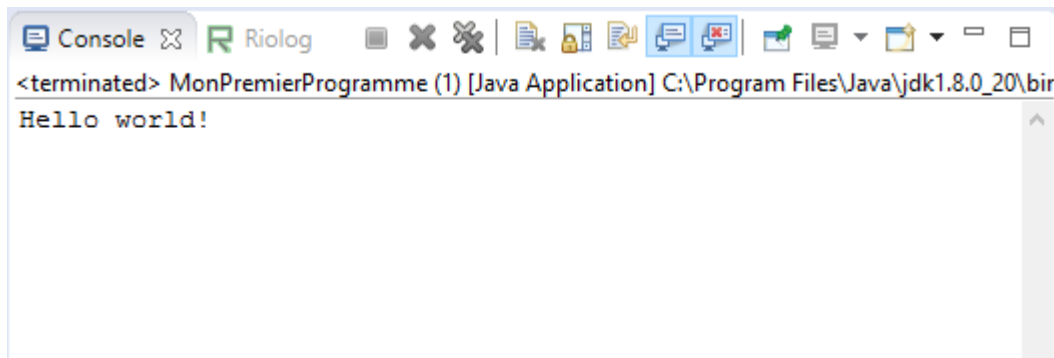


FIGURE 2.3 – Écriture dans la console.

Félicitations, vous venez d'exécuter votre premier programme ! Analysons en détail ce qu'il se passe

à l'intérieur.

## 2.3 Les instructions

En Java, une **instruction** est une commande effectuant une certaine action. On écrit une instruction par ligne, et chacune se termine toujours par un **point-virgule** (;). Pour l'instant, votre programme ne contient qu'une instruction :

```
System.out.println("Hello, world!");
```

Vos instructions sont écrites dans la méthode `main`. En Java, tous les programmes ont une méthode `main`. Il s'agit, en quelque sorte, du point d'entrée du programme.

## 2.4 Les chaînes de caractères

Le rôle de votre programme est d'afficher du texte dans la console. Vous avez sûrement remarqué que le texte à afficher est encadré de guillemets anglais ("..."), mais qu'ils n'apparaissent pas dans la console. Ils sont essentiels pour que le compilateur fasse la différence entre du code et du texte. On les appelle des **chaînes de caractères**, ou ***String*** en anglais. Essayer de modifier le texte entre les guillemets et d'exécuter votre programme : vous constaterez que la chaîne de caractères affichée dans la console s'est modifiée !

On peut joindre plusieurs chaînes de caractères ensemble avec l'opérateur `+`. Cette opération s'appelle la **concaténation**. On peut donc écrire :

```
System.out.println("Bonjour " + "à tous" + " et à toutes" + "!");
```

## 2.5 La méthode `println()`

En Java, une **méthode** est une instruction qui réalise une opération prédéfinie. On utilise une méthode en écrivant son nom suivi d'une paire de parenthèses. Certaines méthodes ont besoin de paramètres pour effectuer leur travail. C'est le cas de la méthode `System.out.println()`, qui demande un *String* en paramètre. Elle s'occupe ensuite de l'afficher sur la console.

## 2.6 L'indentation

Dans l'exemple précédent, vous pouvez constater qu'à chaque fois que des accolades (`{...}`) sont ouvertes, on ajoute de l'espace au code qui se situe à l'intérieur. C'est ce que l'on appelle l'**indentation**

du code. C'est essentiel pour rendre le code clair et facile à modifier. Pour indenter son code, on ajoute une tabulation (touche `Tab` ⇌) pour chaque paire ouverte d'accolades. Eclipse s'en occupe automatiquement la plupart du temps.

## 2.7 Les commentaires

### 2.7.1 Les commentaires standards

Lors de l'écriture, il est possible de spécifier au compilateur de ne pas compiler certaines parties du code. C'est ce qu'on appelle les **commentaires**. Ils permettent de spécifier l'utilité des variables, des méthodes, des classes, etc. Il est crucial d'en ajouter, surtout lors d'un projet en collaboration avec plusieurs personnes !

#### CODE 2.2 — Programme de base avec commentaires

```
1  /*
2   * La classe suivante affiche un message
3   * dans la console.
4   */
5  public class MonPremierProgramme {
6
7      /* Fonction principale
8       du programme.          */
9      public static void main(String[] args) {
10
11          //Début du programme
12
13          System.out.println("Hello, world"); //Affichage du message
14
15      }
16
17 }
```

Les plus courants sont les **commentaires en fin de ligne**. Ils débutent par deux barres obliques `//`. Ils informent le compilateur d'ignorer tout le reste de la ligne. Ils sont souvent courts et précis. On les utilise pour mettre en contexte une instruction ou en début de section.

Pour de longs commentaires, on utilise les **commentaires en blocs**. Ils débutent par `/*` et se terminent par `*/`. Le compilateur ignore alors tout ce qui se trouve entre ces deux balises, un peu comme des parenthèses. On les utilise, entre autres, en entête de fichier, pour spécifier le rôle du fichier (ou de la classe), les noms des auteurs et les dates de création et de modification.

Il est important de mettre des commentaires, mais il ne faut pas en abuser (comme dans l'exemple précédent). Il suffit de trouver le juste équilibre entre clarté et concision. Il est également essentiel

de mettre en contexte l'instruction.

Bon commentaire :

```
age += 1; // L'utilisateur vieillit d'un an.
```

Mauvais commentaire :

```
age += 1; // Ajout de 1 à la variable age.
```

## 2.7.2 Les commentaires Javadoc

Ces commentaires spéciaux sont propres au Java. Ils permettent de créer une documentation accessible pour votre projet. Ils sont très semblables aux commentaires en blocs : il suffit de les faire débuter avec deux étoiles `/**` . Vous aurez donc accès au contenu de votre commentaire partout dans votre projet, sans devoir ouvrir à nouveau le fichier d'origine !

### CODE 2.3 — Ajout de commentaires Javadoc

```
1  /**
2   * Ceci est un commentaire Javadoc!
3   * @author Etienne
4   *
5   */
6  public class MonPremierProgramme { ... }
```

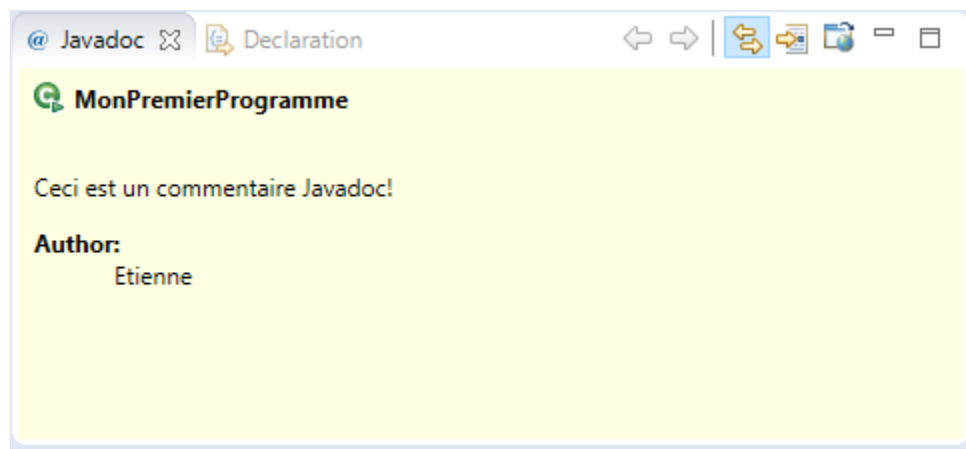


FIGURE 2.4 – Visualisation de la Javadoc dans Eclipse

La Javadoc possède plusieurs attributs spéciaux débutant par un arrobre `@` . Les exemples de ce guide feront appel aux trois attributs suivants.

**@author** On l'utilise dans l'entête d'une classe pour en spécifier l'auteur.

**@param** Dans l'entête de méthodes, il précise le rôle de chaque paramètre.

**@return** Également dans l'entête de méthodes, il précise la valeur de retour.



# Chapitre 3

## Variables et opérateurs

*Manuel de référence : p. 23 à 32.*

### 3.1 La déclaration de variables

Une **variable** est une case mémoire pouvant contenir un certain type de données. Comme son nom l'indique, sa valeur est *variable* : elle peut changer au cours l'exécution.

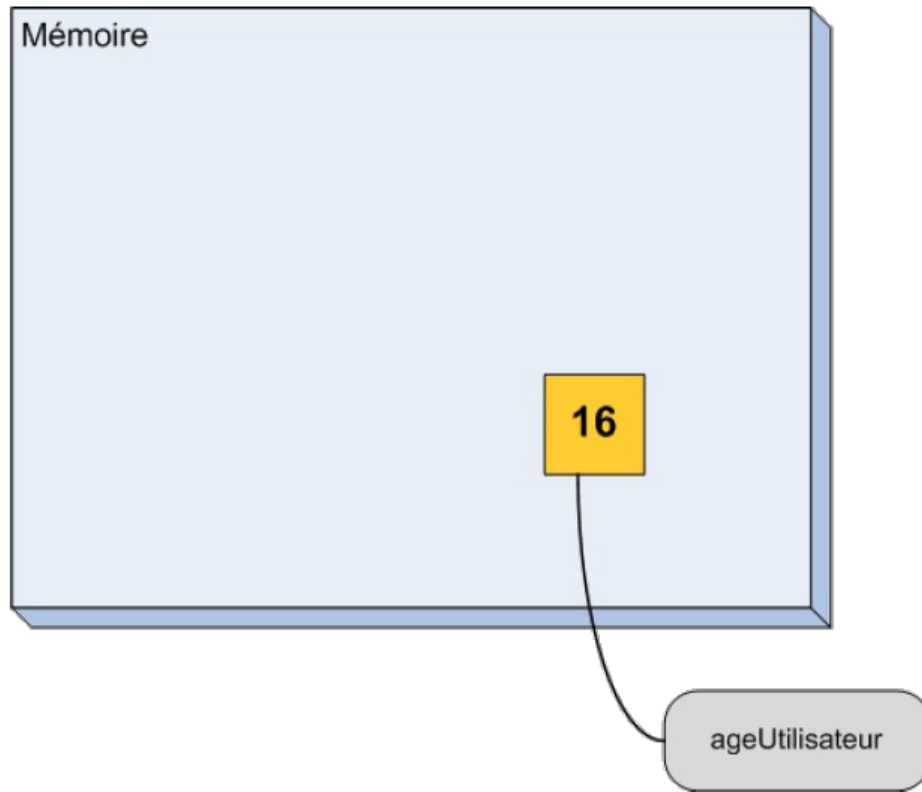


FIGURE 3.1 – Une variable contenant l'âge de l'utilisateur en mémoire.

Pour commencer, regardons un programme utilisant une variable de type *String*.

### CODE 3.1 — Utilisation d'une variable String

```
1  /**
2   * Affiche des noms dans la console.
3   *
4   * @author Etienne
5   */
6  public class AffichageNom {
7
8      public static void main(String[] args) {
9
10         String nom = "Étienne"; // Nom de l'utilisateur
11
12         System.out.println("Je m'appelle " + nom + "!"); //Affichage
13
14         nom = "Alexandre"; //Nouvelle valeur
15
16         System.out.println("Je m'appelle maintenant " + nom + "!");
17         ↪ //Affichage de la nouvelle valeur
18     }
19
20 }
```

#### Sortie console

```
Je m'appelle Étienne!
Je m'appelle maintenant Alexandre!
```

On commence par créer la variable `nom` de type *String* et on lui donne la valeur "Étienne". Pour mettre une valeur dans une variable, on utilise le signe égal (=). L'affectation se fait toujours **de la droite vers la gauche** (⇐). On affiche ensuite la valeur de `nom` dans la console. À la troisième instruction, on met la valeur "Alexandre" dans `nom`. L'ancienne valeur est alors **écrasée** par la nouvelle. La dernière instruction affiche la nouvelle valeur de `nom` dans la console.

## Déclaration et initialisation de variables

Déclaration et initialisation ( $\Leftarrow$ ) dans la même instruction

```
type nomVariable = valeur;
```

Déclaration, puis affectation ( $\Leftarrow$ ) d'une valeur plus tard dans le programme

```
type nomVariable;
```

```
...
```

```
nomVariable = valeur;
```

Lorsque c'est possible, on déclare et on initialise une variable en même temps. C'est ce qui a été fait dans l'exemple précédent. Lorsqu'on ne connaît pas quelle valeur lui donner, on peut la déclarer et lui donner une valeur plus tard.

On peut donner n'importe quel nom à une variable, tant qu'il respecte les conditions suivantes :

- pas d'espace ni d'accent ;
- ne commence pas par un chiffre ;
- commence par une minuscule ;
- si son nom est composé de plusieurs mots, les autres mots peuvent commencer par une majuscule.

Par exemple, les identificateurs `prix`, `ageUtilisateur`, `vitesseMoteurGauche1` et `estOuvert` respectent cette convention.

## 3.2 Lire la console

Vous savez déjà comment afficher du texte dans la console avec la méthode `System.out.println()`. Par contre, il pourrait être pratique de lire ce qui est écrit dans la console. Pour effectuer cette tâche, nous utiliserons la class `Scanner` de la manière suivante.

### CODE 3.2 — Demander et afficher un nom

```
1  import java.util.Scanner;
2
3  /**
4   * Demande le nom de l'utilisateur, puis l'affiche.
5   *
6   * @author Etienne
7   */
8  public class DemanderNom {
9
10     public static void main(String[] args) {
11
12         String nom;
13         Scanner scanner = new Scanner(System.in);
14
15         //Demander le nom
16         System.out.print("Saisissez votre nom : ");
17         nom = scanner.nextLine();
18
19         //Affichage
20         System.out.println("Votre nom est " + nom + "!");
21
22     }
23
24 }
```

#### Sortie console

```
Saisissez votre nom : Étienne
Votre nom est Étienne!
```

Ici, on déclare une variable sans l'initialiser. C'est tout à fait logique, car on ne connaît pas encore le nom à afficher. On déclare ensuite une variable spéciale : la variable `scanner` de type `Scanner`. C'est elle qui va nous permettre de lire les entrées dans la console. Remarquez son initialisation : on utilise le `new` suivi de `Scanner`, le type de notre variable. Nous verrons plus loin que c'est parce que `scanner` est un **objet**, une sorte de « super-variable ».

Par la suite, on utilise une variante de `println()` : la méthode `print()`. Elles agissent presque de la même façon, sauf que `print()` n'ajoute pas de saut de ligne après avoir affiché le texte. Essayez les deux et constatez la différence.

Ensuite, on utilise notre `scanner` et on appelle sa méthode `nextLine()`. Cela indique au programme de faire une pause jusqu'à ce qu'on écrive un mot dans la console et qu'on appuie sur la touche `Entrée`. Le texte saisi est ensuite stocké dans la variable `nom` grâce à l'opérateur `=`.

Finalement, on affiche la valeur de `nom` par concaténation avec d'autres chaînes de caractères.

### 3.3 Les variables de type primitif

Jusqu'à présent, nous avons uniquement déclaré des variables de type `String` et `Scanner`. Ces variables sont en vérité des **objets**. Nous verrons plus tard ce que cela signifie. Il existe cependant des types de variables qui sont à la base de tout : les types primitifs.

Type	Ce qu'il contient	Exemple
<code>int</code>	Un nombre entier.	<code>int ageUtilisateur = 20;</code>
<code>double</code>	Un nombre à virgules de précision double.	<code>double prix = 19.95;</code>
<code>boolean</code>	Une valeur booléenne ( <b>true</b> ou <b>false</b> ).	<code>boolean estOuvert = true;</code>

FIGURE 3.2 – Les types primitifs les plus utilisés.

Ces types débutent par une minuscule puisqu'ils sont primitifs, alors que `String` et `Scanner` débutent par une majuscule puisqu'ils représentent une classe d'objets.

#### CODE 3.3 — Affichage de variables primitives

```
1  /**
2   * Affiche des données de type primitif.
3   *
4   * @author Etienne
5   */
6  public class AffichagePrimitif {
7
8      public static void main(String[] args) {
9
10         int age = 14, ageAmi = 13;
11         double taille = 1.45;
12
13         System.out.println("J'ai " + age + " ans!");
14         System.out.println("Mon ami a " + ageAmi + " ans.");
15         System.out.println("Je mesure " + taille + " m.");
16
17     }
18
19 }
```

## Sortie console

```
J'ai 14 ans!  
Mon ami a 13 ans.  
Je mesure 1.45 m.
```

À la ligne 10, on déclare deux variables du même type sur la même ligne. C'est tout à fait légal, il suffit de séparer leurs noms par des virgules.

Avant un Scanner, il est également possible d'obtenir des données de type primitif à partir de la console.

### CODE 3.4 — Demande de l'âge et de la taille

```
1  import java.util.Scanner;  
2  
3  /**  
4   * Demande l'âge et la taille de l'utilisateur,  
5   * puis l'affiche dans la console.  
6   *  
7   * @author Etienne  
8   */  
9  public class AgeTaille {  
10  
11     public static void main(String[] args) {  
12  
13         int age;  
14         double taille;  
15         Scanner scanner = new Scanner(System.in);  
16  
17         //Demande de l'âge  
18         System.out.print("Saisissez votre âge : ");  
19         age = scanner.nextInt();  
20  
21         //Demande de la taille  
22         System.out.print("Saisissez votre taille : ");  
23         taille = scanner.nextDouble();  
24  
25         //Affichage  
26         System.out.println("Vous avez " + age + " ans et mesurez " +  
27             ↳ taille + " m.");  
28     }  
29  
30 }
```

#### Sortie console

```
Saisissez votre âge : 20
Saisissez votre taille : 1,80
Vous avez 20 ans et mesurez 1.8 m.
```

Tout comme `nextLine()`, les méthodes `nextInt()` et `nextDouble()` attendent qu'une valeur soit saisie dans la console. Elles retournent ensuite ces valeurs pour qu'elles puissent être stockées dans des variables de notre choix.

## 3.4 Les constantes

Jusqu'à présent, une variable agit comme une case dans laquelle on range une donnée pour pouvoir y faire référence plus tard. Cette valeur peut changer au cours du programme. Cependant, dans certains cas, on peut vouloir que le contenu d'une variable ne puisse pas changer. C'est ce qu'on appelle une **constante**.

#### Déclaration de constantes

Utilisation du mot-clé **final**

```
final type NOM_CONSTANTE = valeur;
```

Il suffit d'ajouter le mot-clé `final` devant la déclaration pour transformer une variable en constante. Il est alors impossible de redéfinir sa valeur. Par convention, le nom d'une constante est écrit tout en majuscules. On utilise alors la barre de soulignement pour séparer les différents mots.

```
final double NOMBRE_NIVEAUX = 10; //Constante
NOMBRE_NIVEAUX = 12; // Le compilateur affiche une erreur!
```

Les constantes sont très utiles pour les valeurs qui changent rarement et qui sont utilisées à plusieurs endroits. Si cette valeur doit être modifiée, il suffit alors de la changer à un endroit et le tour est joué !

## 3.5 Les opérateurs arithmétiques

Maintenant que l'on peut stocker des nombres dans des variables, voyons comment effectuer des opérations arithmétiques sur ceux-ci. La priorité des opérations s'applique.

Opération	Symbole
Addition	+
Soustraction	-
Multiplication	*
Division	/
Modulo (reste de la division)	%

FIGURE 3.3 – Les opérateurs arithmétiques.

#### CODE 3.5 — Années avant la majorité

```

1  import java.util.Scanner;
2
3  /**
4   * Demande l'âge de l'utilisateur et affiche le nombre
5   * d'années avant qu'il soit majeur.
6   *
7   * @author Etienne
8   */
9  public class Majorite {
10
11      public static void main(String[] args) {
12
13          int age;
14          final int AGE_MAJORITE = 18; // L'âge de la majorité est fixe
15          Scanner scanner = new Scanner(System.in);
16
17          //Demander l'âge
18          System.out.print("Saisissez votre âge : ");
19          age = scanner.nextInt();
20
21          //Âge avant majorité
22          System.out.println("Vous serez majeur dans " + (AGE_MAJORITE -
23              ↪ age) + " ans.");
24      }
25
26  }

```



### CODE 3.6 — Liquidation d'un inventaire

```
1  import java.util.Scanner;
2
3  /**
4   * Gestion de la liquidation
5   * d'un inventaire.
6   *
7   * @author Etienne
8   */
9  public class Liquidation {
10
11      public static void main(String[] args) {
12
13          int joursRestants; // Nombre de jours avant la fermeture
14
15          int nombreItems; // Nombre d'items restants en inventaire
16          double prixItem; // Prix d'un item
17          double prixTotal; // Prix d'un item avec les taxes
18          double valeurInventaire; // Valeur de tous les items
19
20          final double TAXES = 1.15; // Taxes fixes de 15%
21
22          Scanner scanner = new Scanner(System.in);
23
24
25          // Nombre de jours restants
26          System.out.print("Nombre de jours avant la fermeture : ");
27          joursRestants = scanner.nextInt();
28
29          // Nombre d'items
30          System.out.print("Nombre d'items restants : ");
31          nombreItems = scanner.nextInt();
32
33          // Prix d'un item
34          System.out.print("Prix d'un item : ");
35          prixItem = scanner.nextDouble();
36
37          // Prix total d'un item
38          prixTotal = prixItem * TAXES;
39
40          // Valeur de l'inventaire
41          valeurInventaire = prixItem * nombreItems;
42

```

```

43      // Affichage
44      System.out.println("Le prix à payer avec taxes est de " +
        ↳ prixTotal + "$.");
45      System.out.println("Votre inventaire vaut " + valeurInventaire +
        ↳ "$.");
46      System.out.println("Pour tout liquider, vous devrez vendre
        ↳ environ " + (nombreItems / joursRestants) + " items par
        ↳ jour.");
47
48    }
49 }

```

#### Sortie console

```

Nombre de jours avant la fermeture : 10
Nombre d'items restants : 50
Prix d'un item : 11,40
Le prix à payer avec taxes est de 13.11$.
Votre inventaire vaut 570.0$.
Pour tout liquider, vous devrez vendre environ 5 items par jour.

```

Dans les deux exemples précédents, on effectue des calculs avant l'affectation d'une variable ou directement dans un `println()` pour afficher le résultat. Les deux sont acceptables, tant que le code reste clair et que les parenthèses sont placées aux bons endroits. Dans le cas de calculs longs et compliqués, il est préférable de les isoler pour que ce soit plus lisible.

## 3.6 Les opérateurs d'affectation

Vous connaissez déjà l'opération d'affectation de base : le symbole égal (=). C'est le plus commun. Il existe cependant des raccourcis qui peuvent être utiles dans plusieurs cas. Dans tous les cas, l'affectation se fait toujours **de la droite vers la gauche** (⇐).

Rôle	Symbole	Exemple	Équivalent
Ajout de...	<code>+=</code>	<code>age += 2;</code>	<code>age = age + 2;</code>
Retrait de...	<code>-=</code>	<code>vie -= dommages;</code>	<code>vie = vie - dommages;</code>
Multiplier par...	<code>*=</code>	<code>prix *= rabais;</code>	<code>prix = prix * rabais;</code>
Diviser par...	<code>/=</code>	<code>taille /= 3.28;</code>	<code>taille = taille / 3.28;</code>
Incrémentation	<code>++</code>	<code>compteur++;</code>	<code>compteur += 1;</code>
Décrémentation	<code>--</code>	<code>points--;</code>	<code>points -= 1;</code>

FIGURE 3.4 – Les opérateurs d’affectation.

## 3.7 La librairie Math

Java a une librairie standard très fournie. Elle comprend, entre autres, la classe `Math`. Celle-ci fournit plusieurs fonctions mathématiques de base :

- la valeur absolue (`abs`),
- les exposants (`pow`),
- les fonctions trigonométriques (`sin`, `cos`, `tan`, etc.),
- le maximum et le minimum (`max` et `min`),
- la racine carrée (`sqrt`),
- l’arrondi (`round`, `ceil` et `floor`).

### CODE 3.7 — Utilisation de la classe Math

```
1  import java.util.Scanner;
2
3  /**
4   * Affiche la valeur absolue et le cube d'un nombre.
5   *
6   * @author Etienne
7   *
8   */
9  public class TestMath {
10
11     public static void main(String[] args) {
12
13         double nombre;
14         Scanner scanner = new Scanner(System.in);
15
16         //Obtention du nombre
17         System.out.print("Saisissez un nombre : ");
18         nombre = scanner.nextDouble();
19
20         //Calculs
21         System.out.println("\nLa valeur absolue du nombre est : " +
22             ↪ Math.abs(nombre));
23         System.out.println("Le cube du nombre est : " + Math.pow(nombre,
24             ↪ 3));
25     }
26 }
```

La méthode `pow()` prend deux paramètres : le premier est la base et le deuxième est l'exposant. Ainsi,  $\text{pow}(x, y) = x^y$ .

# Chapitre 4

## Les instructions conditionnelles

*Manuel de référence : p. 39 à 46.*

### 4.1 Le *if else*

Jusqu'à présent, vos programmes se sont exécutés de manière **séquentielle** : toutes les instructions sont exécutées les unes après les autres. En réalité, il est bien rare qu'un programme suive une seule séquence. Avec les **instructions conditionnelles**, il sera possible d'exécuter une certaine partie de votre programme uniquement si une certaine condition est respectée. Nous utiliserons alors le *if else*.

#### La structure *if else*

```
if ( condition ) {  
    instruction1;  
    ... // Si la condition est vraie...  
}  
else { // Facultatif  
    instruction2;  
    ... // Si la condition est fausse...  
}
```

Si la condition donnée est vraie, alors les instructions contenues dans le premier bloc seront exécutées. Sinon, ce sont celles du deuxième bloc qui seront exécutées. Le *else* est facultatif. S'il n'y a pas de *else* et que la condition est fausse, alors le programme continue son exécution normalement, sans exécuter le contenu du *if*.

Comme une condition peut être vraie ou fausse, on dit que c'est une condition **booléenne**. C'est d'ailleurs une des principales utilités du type `boolean`.

## 4.2 Les opérateurs de comparaison et les opérateurs logiques

Pour exprimer une condition, on peut utiliser les **opérateurs de comparaison** et les **opérateurs logiques**. Les principaux sont les suivants.

Rôle	Symbole	Exemple
égal	<code>==</code>	<code>a == 2</code>
n'est pas égal	<code>!=</code>	<code>prix != 10</code>
est plus grand	<code>&gt;</code>	<code>rabais &gt; prix</code>
est plus grand ou égal	<code>&gt;=</code>	<code>age &gt;= AGE_MAJORITE</code>
est plus petit	<code>&lt;</code>	<code>rotation &lt; 25.1</code>
est plus petit ou égal	<code>&lt;=</code>	<code>distance &lt;= 120</code>
ou	<code>  </code>	<code>a == 10    b == 5</code>
et	<code>&amp;&amp;</code>	<code>distance &gt;= 10 &amp;&amp; angle == 180</code>
n'est pas	<code>!</code>	<code>!estOuvert</code>

FIGURE 4.1 – Les opérateurs de comparaison et les opérateurs logiques.

Les six premiers sont appelés les **opérateurs de comparaison**. On les utilise pour comparer des nombres ensemble. Les trois derniers sont appelés les **opérateurs logiques**. Ils permettent de modifier d'autres conditions ou booléens. Avec le **ou**, la nouvelle condition est vraie si la première ou la deuxième condition est vraie. Quant au **et**, il faut que les deux conditions soient vraies pour que la nouvelle condition soit vraie. Finalement, le dernier opérateur permet d'inverser une valeur booléenne.

#### CODE 4.1 — Validation d'une année de naissance

```
1  import java.util.Calendar;
2  import java.util.Scanner;
3  /**
4   * Validation d'une année de naissance.
5   *
6   * @author Etienne
7   */
8  public class ValidationNaissance {
9
10     public static void main(String[] args) {
11
12         int annee;
13         final int ANNEE_MINIMALE = 1900;
14         final int ANNEE_COURANTE =
15             ↪ Calendar.getInstance().get(Calendar.YEAR); //2017
16         Scanner scanner = new Scanner(System.in);
17
18         //Obtention de l'année
19         System.out.print("Saisissez une année de naissance : ");
20         annee = scanner.nextInt();
21
22
23         //Validation
24         if(annee >= ANNEE_MINIMALE && annee <= ANNEE_COURANTE) {
25             System.out.println("Année valide.");
26         }
27         else {
28             System.out.println("L'année " + annee + " est invalide.");
29             System.out.println("Vous devez recommencer!");
30         }
31
32     }
33
34 }
```

### 4.3 Les *else if* multiples

## **Deuxième partie**

### **La programmation orientée objet**



# **Troisième partie**

## **La librairie WPILib**

# Chapitre 5

## La classe `TimedRobot`

### 5.1 Création d'un projet

Jusqu'à présent, le coeur de vos programmes se trouvait dans la méthode `main`. Cependant, le comportement d'un robot est plus complexe qu'une simple méthode. Vos prochains programmes auront comme base la classe `Robot`, héritant de `TimedRobot`.

1. Dans Eclipse, débutez la création d'un nouveau projet avec `File > New > Other...`, puis, dans le dossier `WPILib Robot Java Project`, sélectionnez `Robot Java Project`.
2. Donnez un nom à votre projet (débutant avec une lettre majuscule).
3. Sélectionnez `Command-Based Robot` ou `Timed Robot`, selon le paradigme que vous souhaitez utiliser. Cliquez sur `Finish`.

Vous vous retrouverez ainsi avec un fichier `Robot.java` contenant le squelette d'un programme pour la FRC.

### 5.2 Les méthodes `Init` et `Periodic`

La classe qu'Eclipse a générée pour vous contient déjà quelques méthodes prédéfinies. Chacune d'entre elles a un rôle bien précis.

Chaque état dans lequel peut être le robot possède sa méthode `Init` et `Periodic`. Par exemple, lorsque le robot entre en période autonome, le contenu de la méthode `autonomousInit` est appelé une fois. Ensuite, la méthode `autonomousPeriodic` est appelée en boucle (périodiquement, soit environ aux 20 millisecondes) tant que le robot reste dans cet état.

La seule exception est la méthode `robotInit` : elle s'exécute une fois, au tout début, lorsque le programme démarre. Habituellement, on y initialise les différentes composantes du robot, ses sous-systèmes, etc.

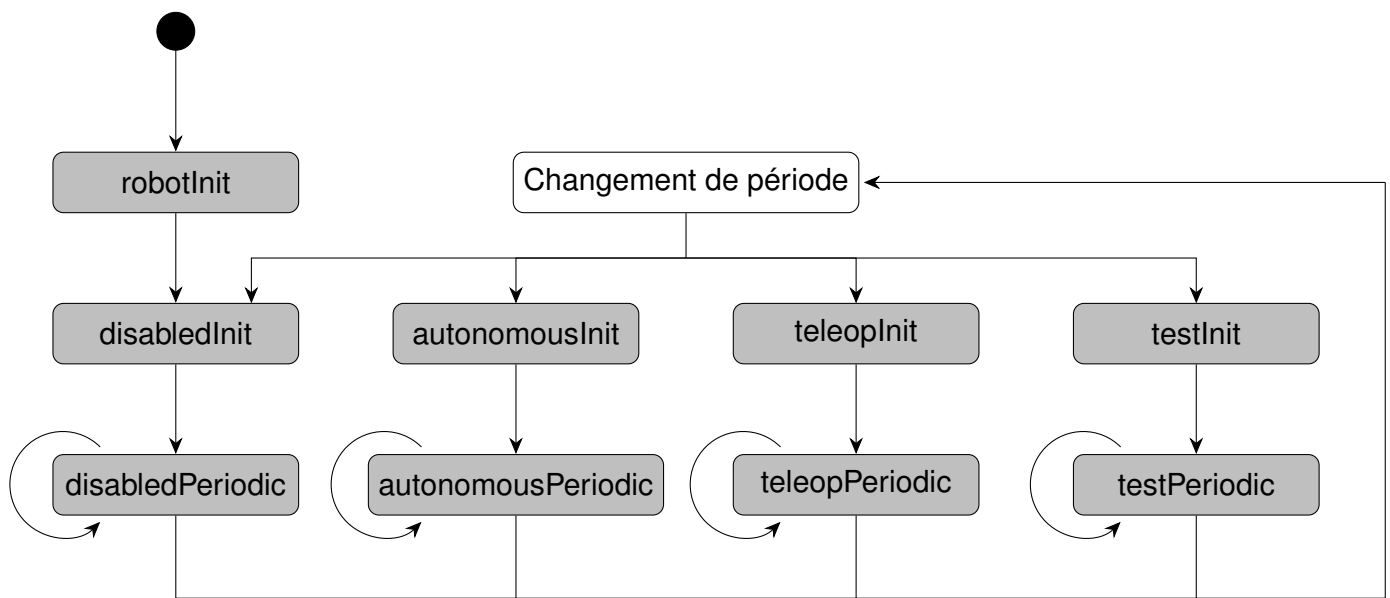


FIGURE 5.1 – Séquence d'exécution des méthodes de la classe Robot.

## 5.3 Faire avancer le robot

CODE 5.1 — Utilisation de DifferentialDrive avec Joystick

```
1 public class MonPremierRobot extends TimedRobot {
2
3     private VictorSP moteurGauche, moteurDroit;
4     private DifferentialDrive basePilotable;
5     private Joystick joystick;
6
7     @Override
8     public void robotInit() {
9         moteurGauche = new VictorSP(0);
10        moteurDroit = new VictorSP(1);
11        basePilotable = new DifferentialDrive(moteurGauche,
12        ↪      moteurDroit);
13        joystick = new Joystick(0);
14    }
15
16    @Override
17    public void autonomousPeriodic() {
18        moteurGauche.set(0.5);
19        moteurDroit.set(-0.5);
20    }
21
22    @Override
23    public void teleopPeriodic() {
24        basePilotable.arcadeDrive(-1 * joystick.getY(),
25        ↪      joystick.getX());
26    }
27 }
```

Dans un premier temps, on déclare comme attributs privés de classe les composantes de notre robot : deux contrôleurs moteur VictorSP, une DifferentialDrive et un Joystick. On les instancie dans la méthode `robotInit`.

Ensuite, en période autonome, on met le moteur gauche à 50 % de sa puissance vers à l'avant, et le moteur droit, à 50 % vers l'arrière. Le robot tournera donc en rond pendant 15 secondes.

Finalement, en période téléopérée, on utilise la méthode `arcadeDrive` avec le Joystick pour contrôler le robot naturellement.

### 5.3.1 La classe `VictorSP`

Cette classe sert à déclarer des contrôleurs moteur de type `VictorSP`. Des classes existent pour chaque modèle de contrôleur moteur : `Talon`, `Spark`, `Jaguar`, etc.

Son constructeur reçoit en paramètre un nombre entier (voir lignes 9 et 10). Il s'agit du port PWM du RoboRIO où est branché le contrôleur.

Cette classe possède la méthode `set(double)`. Elle reçoit un nombre compris entre -1.0 et 1.0, spécifiant la puissance à laquelle le moteur doit aller.

### 5.3.2 La classe `DifferentialDrive`

Cette classe fournit plusieurs méthodes utiles pour contrôler la base pilotable. Bien qu'il serait possible de calculer manuellement les valeurs à envoyer à chaque moteur, il est plus pratique (et rapide) d'utiliser cette classe. Elle est configurée pour les bases pilotables de type « tank ». Les classes `MecanumDrive` et `KilloughDrive` (robot à trois roues) sont également disponibles.

Le constructeur reçoit comme arguments les deux contrôleurs moteur : celui de gauche et celui de droite. La `DifferentialDrive` doit les avoir en référence pour contrôler la base pilotable.

Sa méthode `arcadeDrive(double, double)` reçoit deux nombres en paramètre : `forward` et `rotation`. Le paramètre `forward` (entre -1.0 et 1.0) représente la vitesse d'avancée en ligne droite. Le paramètre `rotation` (entre -1.0 et 1.0) représente la vitesse de rotation, où les valeurs positives vont vers la droite (sens horaire).

### 5.3.3 La classe `Joystick`

On utilise évidemment cette classe pour lire les valeurs d'un Joystick. Son constructeur reçoit en paramètre l'ordre de branchement du Joystick dans la `DriverStation` (à partir de 0). Cet ordre est très important lorsque le robot est contrôlé par plus d'une manette en même temps.

On accède aux valeurs des axes (de -1.0 à 1.0) avec la méthode `getRawAxis(int)`, où l'entier en paramètre est le numéro de l'axe (à partir de 0). Des raccourcis existent pour les axes principaux : `getX()` et `getY()`. Cependant, l'axe des Y est inversé selon notre logique : il renvoie -1.0 lorsqu'on pousse le Joystick vers l'avant, et 1.0 lorsqu'on le tire vers soi. On multiplie donc cette valeur par -1 avant de l'envoyer comme paramètre `forward` à `arcadeDrive` (voir ligne 24).

L'état des boutons est donné par la méthode `getRawButton(int)`, où l'entier en paramètre est le numéro du bouton (à partir de 1). La méthode renvoie le booléen `true` si le bouton est appuyé, et `false` dans le cas contraire.