

# Guide

Formation Java et WPILib

Étienne Beaulac  
Ultime FRC 5528

Dernière modification  
22 septembre 2018





# Table des matières

<b>Table des extraits de code</b>	<b>iv</b>
<b>Table des figures</b>	<b>v</b>
<b>Les valeurs FIRST</b>	<b>vi</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 ? . . . . .	3
<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
4.3	Les <i>else if</i> multiples . . . . .	25
<b>II</b>	<b>La librairie WPILib</b>	<b>26</b>
<b>5</b>	<b>La classe TimedRobot</b>	<b>27</b>
5.1	Création d'un projet . . . . .	27
5.2	Les méthodes Init et Periodic . . . . .	27
5.3	Faire avancer le robot . . . . .	29
5.3.1	La classe VictorSP . . . . .	30
5.3.2	La classe DifferentialDrive . . . . .	30
5.3.3	La classe Joystick . . . . .	30
<b>6</b>	<b>Les actionneurs</b>	<b>31</b>
6.1	Les moteurs . . . . .	31
6.2	La pneumatique . . . . .	31
6.2.1	Les solénoïdes simples . . . . .	32
6.2.2	Les solénoïdes doubles . . . . .	33
6.3	Les servomoteurs . . . . .	34
<b>7</b>	<b>Les capteurs</b>	<b>36</b>
7.1	Les interrupteurs de fin de course . . . . .	36
7.2	Les potentiomètres . . . . .	37
7.3	Les gyroscopes . . . . .	39
7.4	Les encodeurs . . . . .	42

7.5	ADIS16448_IMU . . . . .	44
7.6	Les autres capteurs . . . . .	44
7.6.1	Les accéléromètres . . . . .	44
7.6.2	Les capteurs ultrasons . . . . .	45
<b>8</b>	<b>Commandes et sous-systèmes</b>	<b>46</b>
8.1	Une nouvelle façon d'organiser nos idées . . . . .	46
8.2	Créer un projet à base de commandes . . . . .	47
8.3	Créer un sous-système . . . . .	47
8.3.1	Le constructeur . . . . .	48
8.3.2	Ajouter une commande par défaut . . . . .	49
8.3.3	Ajouter des méthodes à un sous-système . . . . .	50
8.4	Créer une commande . . . . .	50
8.4.1	Le constructeur . . . . .	50
8.4.2	Lier un bouton à une commande dans la classe OI . . . . .	50
8.5	Les groupes de commandes . . . . .	50
8.5.1	Le processus de création d'une commande . . . . .	51
<b>III</b>	<b>Git et les logiciels de gestion de versions</b>	<b>52</b>

## 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 . . . . .	29
6.1	Utilisation d'un solénoïde simple . . . . .	32
6.2	Utilisation d'un solénoïde double . . . . .	33
6.3	Utilisation d'un servomoteur . . . . .	34
7.1	Utilisation d'un interrupteur de fin de course (limit switch) . . . . .	37
7.2	Utilisation d'un potentiomètre . . . . .	38
7.3	Utilisation d'un gyro . . . . .	40
7.4	Aller en ligne droite avec un gyro . . . . .	41
7.5	Utilisation d'un encodeur . . . . .	43
8.1	Les attributs et le constructeur d'un sous-système . . . . .	48

# 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. . . . .	28
7.1	Les angles mesurés par un gyroscope à 3 axes (X, Y et Z) . . . . .	39
8.1	La hiérarchie d'un nouveau projet <i>Command robot</i> . . . . .	47

# Les valeurs *FIRST*

## **Découverte**

Nous explorons de nouvelles idées et habilités.

## **Innovation**

Nous sommes créatifs et déterminés à résoudre des problèmes.

## **Impact**

Nous nous servons de nos apprentissages pour améliorer notre monde.

## **Inclusion**

Nous nous respectons mutuellement et nous sommes ouverts à la diversité.

## **Travail d'équipe**

Nous sommes plus forts en travaillant ensemble.

## **Plaisir**

Nous apprécions et célébrons nos accomplissements.



# **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. 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*, ce qui fait de lui le fondateur de l'informatique moderne. Pendant la Seconde Guerre mondiale, il déchiffra le code de la machine Enigma, ce qui sauva la vie à plus de 14 millions de personnes. Sa carrière prit fin tragiquement en 1954 lorsqu'il se suicida à l'âge de 41 ans, après avoir été condamné pour indécence en raison de son homosexualité.

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

## 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*.

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 paquets ( <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 ...

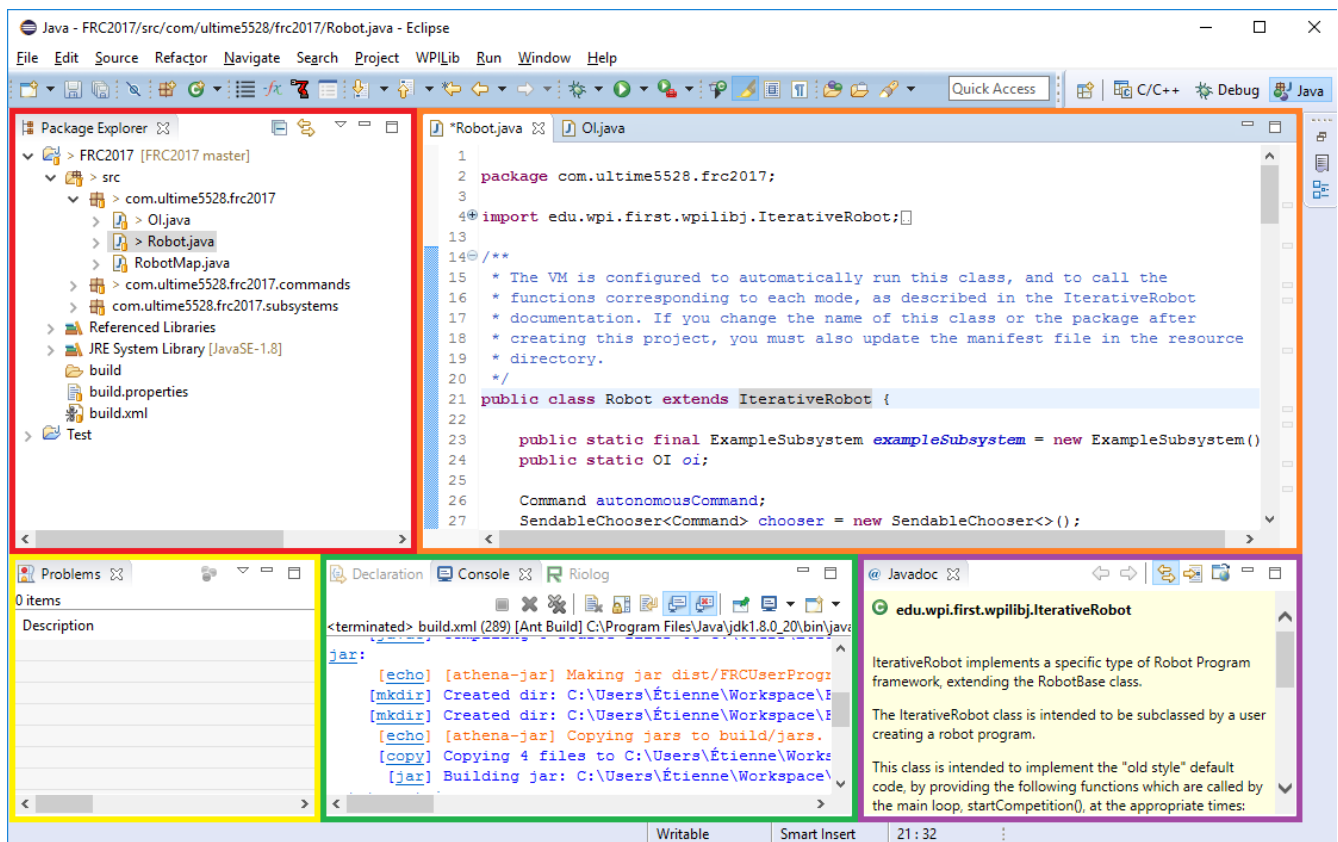


FIGURE 2.1 – L'interface principale de Eclipse.

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 }
```

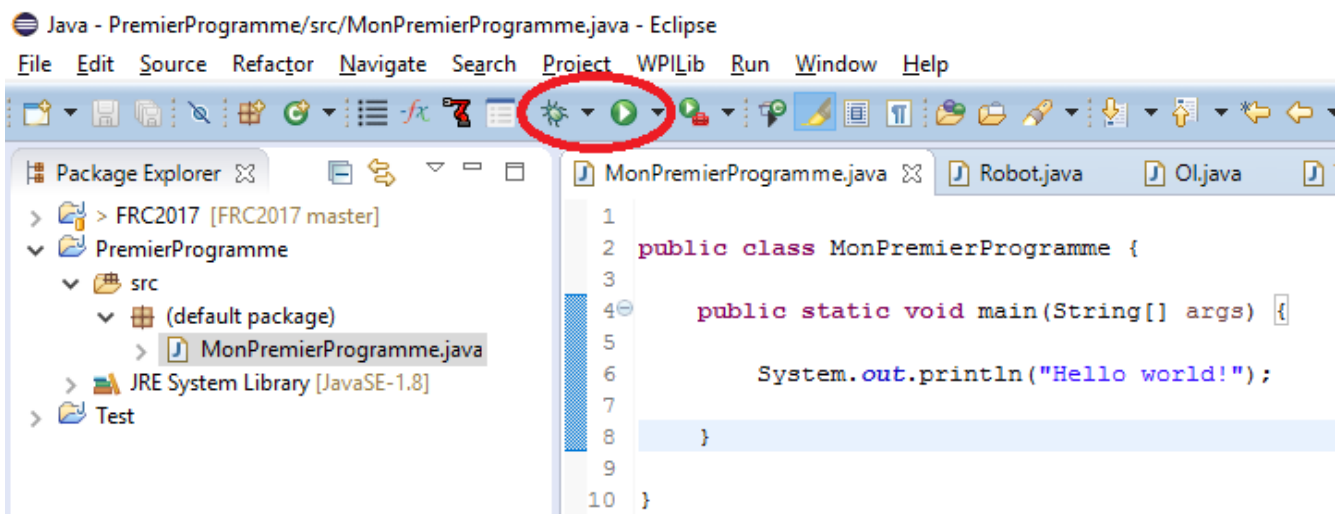




FIGURE 2.2 – Compiler, exécuter et déboguer un programme avec Eclipse.

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.

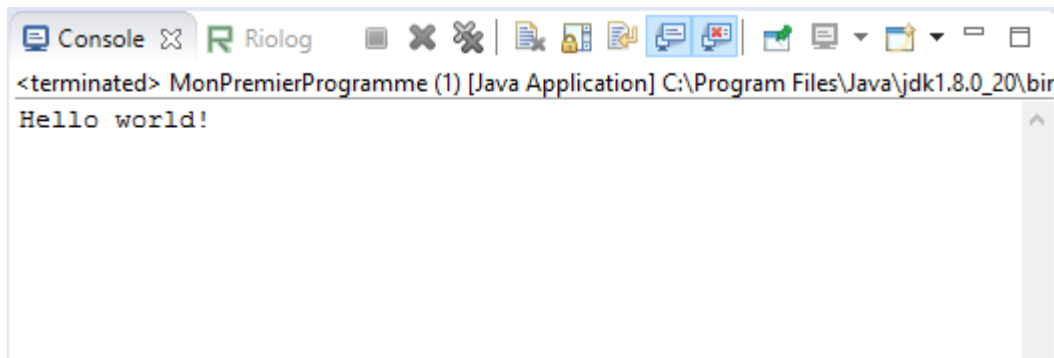


FIGURE 2.3 – Écriture dans la console.

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 { ... }
```

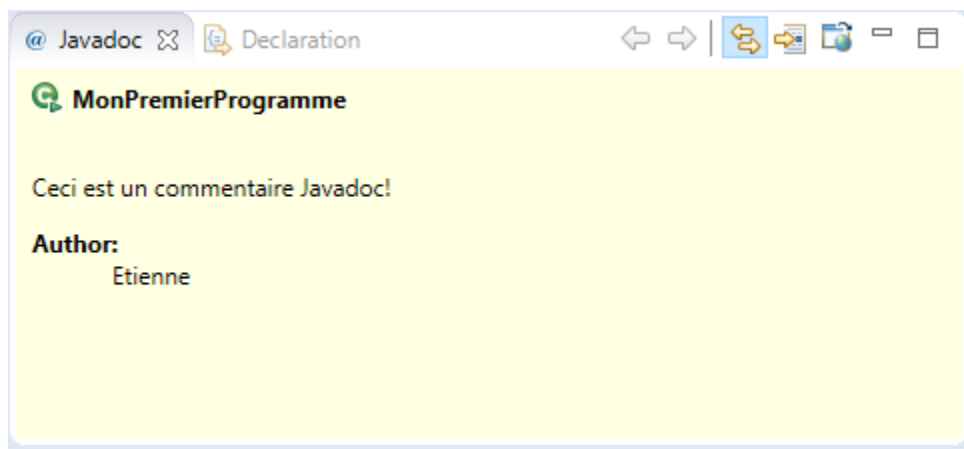


FIGURE 2.4 – Visualisation de la Javadoc dans Eclipse

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.

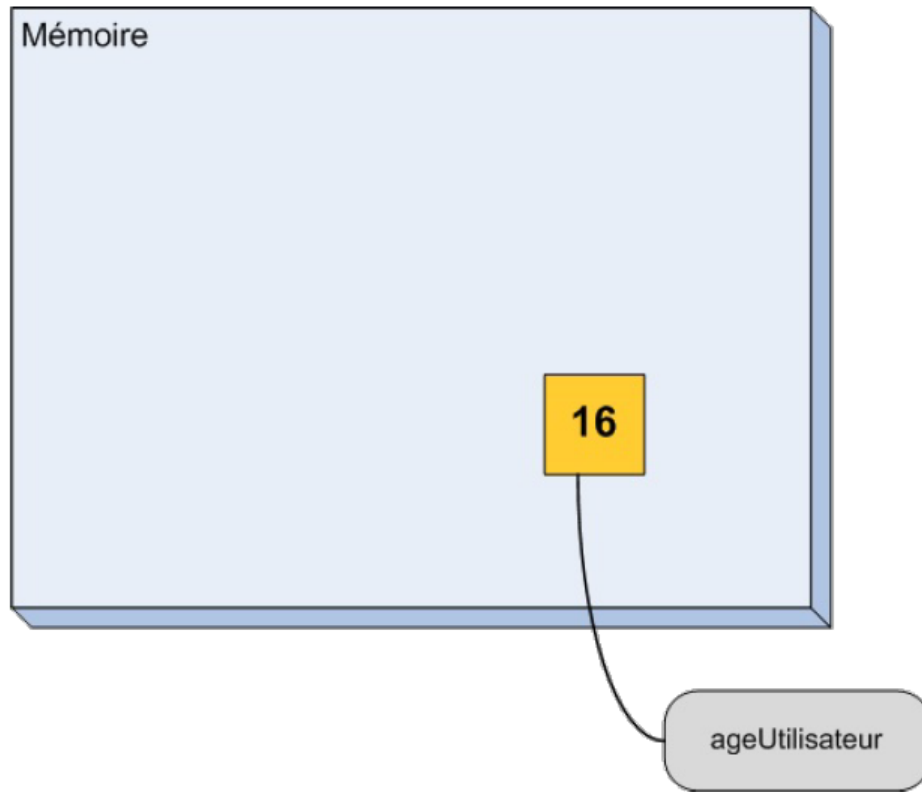


FIGURE 3.1 – Une variable contenant l'âge de l'utilisateur en mémoire.

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`, `vitesseGauche1` 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>
Décrémententation	<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 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.

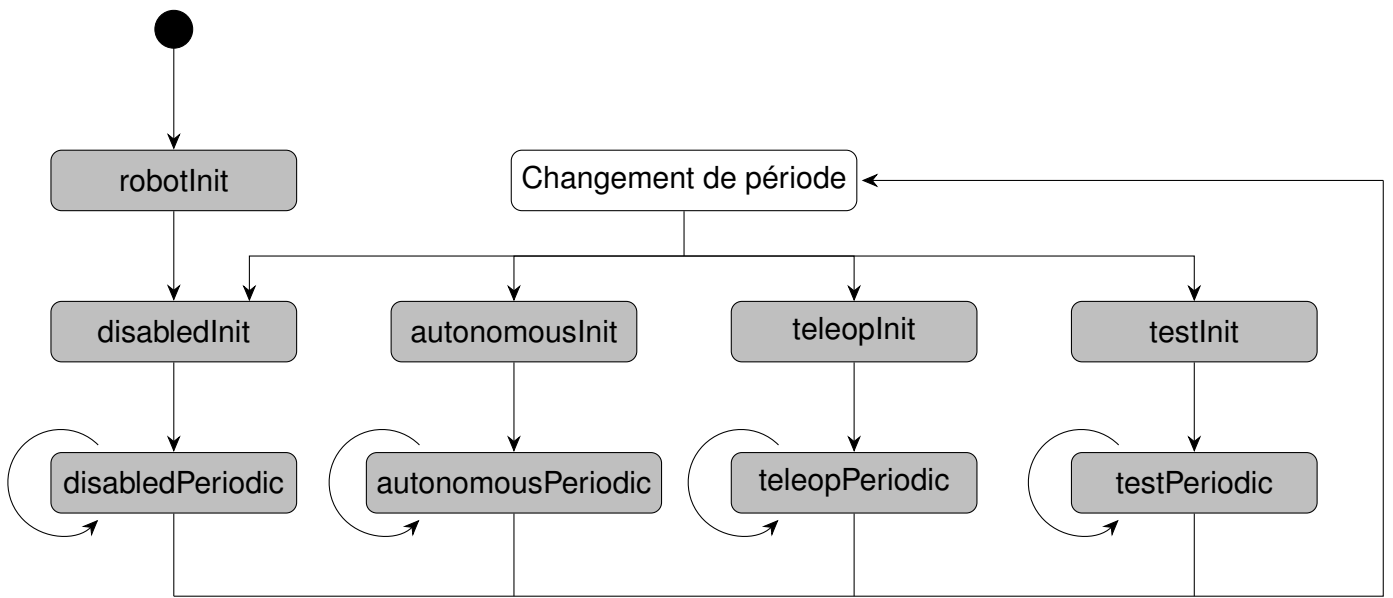


FIGURE 5.1 – Séquence d'exécution des méthodes de la classe Robot.

## 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.

# Chapitre 6

## Les actionneurs <sup>1</sup>

Plusieurs définissent un robot comme étant une machine qui possède un système logique, des capteurs et des actionneurs. Le système logique, c'est le programme exécuté par le RoboRIO. Les capteurs seront utilisés au prochain chapitre. Quant aux actionneurs (*actuators* en anglais), ce sont toute partie du robot qui crée un mouvement. Plusieurs choix d'actuateurs s'offrent à nous, selon nos besoins.

### 6.1 Les moteurs

Pour un exemple d'utilisation de moteurs (VictorSP, DifferentialDrive), voir le chapitre précédent.

### 6.2 La pneumatique

Physiquement, un système pneumatique est composé de plusieurs éléments :

- une bonbonne, pour contenir l'air comprimé ;
- un compresseur, pour comprimer de l'air dans la bonbonne ;
- un pressostat (*pressure switch* en anglais), pour mesurer la pression dans la bonbonne ;
- un PCM (module de contrôle pneumatique : *Pneumatic Control Module*), pour activer le compresseur et l'arrêter lorsque la pression est assez élevée ;
- des vérins (pistons, cylindres pneumatiques), pour effectuer les mouvements désirés ;
- des solénoïdes, pour laisser passer de l'air dans les vérins.

Au niveau de la programmation, les seuls éléments que nous devons gérer sont les solénoïdes. Ceux-ci agissent comme des « portes » qui laissent passer l'air (ou non) selon le signal qu'on leur envoie. Attention : les solénoïdes ne sont pas branchés au RoboRIO, mais bien au PCM.

---

1. Référence : <http://wpilib.screenstepslive.com/s/currentCS/m/java/c/88897>

## 6.2.1 Les solénoïdes simples

Les solénoïdes simples ne comportent qu'une seule valve. Ils peuvent être dans deux états :

- **false** : la valve est fermée : aucun air n'est envoyé ;
- **true** : la valve est ouverte : de l'air est envoyé.

Lorsqu'un objet `Solenoid` est instancié, le PCM active automatiquement le compresseur. Cependant, si le programme ne déclare aucun solénoïde, le compresseur ne s'activera tout simplement pas.

CODE 6.1 — Utilisation d'un solénoïde simple

```
1 public class UtilisationSolenoidSimple extends TimedRobot {
2
3     private Solenoid solenoid;
4     private Joystick joystick;
5
6     @Override
7     public void robotInit() {
8         // Branché sur le port 2 du PCM
9         solenoid = new Solenoid(2);
10        joystick = new Joystick(0);
11    }
12
13    @Override
14    public void teleopPeriodic() {
15
16        if(joystick.getRawButton(1)) {
17
18            solenoid.set(true);
19
20        } else {
21
22            solenoid.set(false);
23
24        }
25
26    }
27
28 }
```

Ce code simple ouvre le solénoïde lorsqu'on appuie sur le bouton 1 du Joystick.



## 6.2.2 Les solénoïdes doubles

Les solénoïdes doubles comportent deux valves. Ils peuvent donc être dans 3 états différents :

- `DoubleSolenoid.Value.kOff` : les deux valves sont fermées : aucun air n'est envoyé ;
- `DoubleSolenoid.Value.kForward` : la valve avant est ouverte : de l'air est envoyé dans l'avant du vérin.
- `DoubleSolenoid.Value.kReverse` : la valve arrière est ouverte : de l'air est envoyé dans l'arrière du vérin.

CODE 6.2 — Utilisation d'un solénoïde double

```
1 public class UtilisationSolenoidDouble extends TimedRobot {
2
3     private DoubleSolenoid solenoid;
4     private Joystick joystick;
5
6     @Override
7     public void robotInit() {
8         solenoid = new DoubleSolenoid(3, 4); // Ports 3 et 4 du PCM
9         joystick = new Joystick(0);
10    }
11
12    @Override
13    public void teleopPeriodic() {
14
15        if(joystick.getRawButton(1))
16            solenoid.set(DoubleSolenoid.Value.kForward);
17
18        else if(joystick.getRawButton(2))
19            solenoid.set(DoubleSolenoid.Value.kReverse);
20
21        else
22            solenoid.set(DoubleSolenoid.Value.kOff);
23
24    }
25
26 }
```

Ce type de solénoïde est particulièrement utile pour ouvrir **et** fermer rapidement des mécanismes. Par exemple, pour lancer une balle, la séquence suivante pourrait être exécutée :

1. le solénoïde double est mis en mode `kForward` : le vérin s'allonge et éjecte la balle
2. une seconde plus tard, le solénoïde est mis en mode `kReverse` : le vérin se rétracte, prêt à recevoir une autre balle ;
3. une demi-seconde plus tard, le solénoïde est mis en mode `kOff` : le solénoïde ferme ses

valves, se qui permet au compresseur de remplir la bonbonne.

Attention ! Il est important de toujours fermer les solénoïdes. Sinon, l'air continue de s'échapper et la bonbonne perd de l'air inutilement.

Dans l'exemple précédent, le constructeur d'un `DoubleSolenoid` reçoit en paramètres deux entiers, car il est branché à deux ports du PCM. Cela permet de contrôler indépendamment les deux valves du solénoïde.

## 6.3 Les servomoteurs

Jusqu'à présent, les moteurs que nous avons abordés se contrôlent en terme de puissance : lorsque le programme envoie 0.5 à un `VictorSP`, celui-ci fonctionnera à 50 % de sa capacité. Cependant, il peut être intéressant de contrôler la **position absolue** d'un moteur. C'est pour répondre à ce problème que les servomoteurs ont été créés. Par exemple, on peut fixer une caméra sur un servomoteur pour la faire tourner précisément de 180 degrés et voir derrière le robot.

### CODE 6.3 — Utilisation d'un servomoteur

```
1 public class UtilisationServo extends TimedRobot {
2
3     private Servo servo;
4     private Joystick joystick;
5
6     @Override
7     public void robotInit() {
8         servo = Servo(8); // Branché sur le port PWM 8 du RoboRIO
9         joystick = new Joystick(0);
10    }
11
12    @Override
13    public void teleopPeriodic() {
14
15        if(joystick.getRawButton(1))
16            servo.setAngle(0.0);
17
18        else if(joystick.getRawButton(3))
19            solenoid.setAngle(90.0);
20
21    }
22
23 }
```

Ce programme met le servomoteur à son angle minimum (0.0) au début de la période téléopérée,

puis permet de le faire tourner à 3 positions différentes : 0°, 45° et 90°.

La méthode `setAngle(double)` reçoit en paramètre un nombre entre 0.0 et 180.0, car elle est configurée pour fonctionner avec le servomoteur Hitec HS-322HD, qui a une portée de 180°. Il se peut donc que les valeurs données ne soient pas tout à fait exactes lors de l'utilisation d'autres modèles de servos. Il suffit de faire quelques tests pour déterminer les bons angles.

# Chapitre 7

## Les capteurs <sup>1</sup>

Les capteurs, contrairement aux actionneurs, ne créent aucun mouvement. Ils fournissent au robot des informations sur son environnement et ses déplacements.

### 7.1 Les interrupteurs de fin de course

Un interrupteur de fin de course (communément appelé *limit switch*) est le plus simple des capteurs mis à notre disposition. Ce capteur agit comme un bouton : il nous dit s'il est appuyé (`true`) ou non (`false`). Par exemple, on s'en sert pour déterminer si un bras motorisé a atteint sa position maximale, ou encore pour détecter lorsqu'un ballon a été attrapé.

---

1. <http://wpilib.screenstepslive.com/s/currentCS/m/java/c/88895>

### CODE 7.1 — Utilisation d'un interrupteur de fin de course (limit switch)

```
1 public class UtilisationSwitch extends TimedRobot {
2
3     private DigitalInput limitSwitch;
4     private VictorSP ramasseBallons;
5
6     @Override
7     public void robotInit() {
8         limitSwitch = DigitalInput(2); // Port digital 2
9         ramasseBallons = new VictorSP(1); // Port PWM 1
10    }
11
12    @Override
13    public void autonomousPeriodic() {
14
15        if(limitSwitch.get())
16            ramasseBallons.set(0.0);
17
18        else
19            ramasseBallons.set(1.0).
20
21    }
22
23 }
```

Le programme précédent comporte un mode autonome qui met le moteur à zéro lorsque l'interrupteur retourne une valeur `true`, c'est-à-dire lorsque l'interrupteur est appuyé. Cela veut donc dire qu'il y a un ballon sur l'interrupteur. Sinon, on active le moteur jusqu'à obtenir un ballon.

Ce type de capteur se branche dans un des ports digitaux du RoboRIO, et non dans un port PWM.

Attention ! La valeur d'une *limit switch* peut être inversée. Il faut alors l'inverser à nouveau avec un «!»: `!switch.get()`.

## 7.2 Les potentiomètres

Les potentiomètres sont des capteurs qui mesurent la position absolue (en rotation ou en translation) de composantes du robot. On parle de **position absolue** parce que les potentiomètres **ne se réinitialisent pas** lorsque l'on éteint le robot. Par contre, les potentiomètres ont une portée limitée. Ils ne sont donc pas adaptés pour mesurer la rotation des roues de la base pilotable, par exemple. Cependant, on pourrait les utiliser pour déterminer la hauteur d'un élévateur ou l'angle d'un bras motorisé.

## CODE 7.2 — Utilisation d'un potentiomètre

```
1 public class UtilisationPotentiometre extends TimedRobot {
2
3     private final static double HAUTEUR_MIN = 0.2;
4     private final static double HAUTEUR_MAX = 1.8;
5
6     private AnalogPotentiometer pot;
7     private VictorSP elevateur;
8     private Joystick joystick;
9
10    @Override
11    public void robotInit() {
12        pot = new AnalogPotentiometer(0); // Port analogue 0
13        elevateur = new VictorSP(0); // Port PWM 0
14        joystick = new Joystick(0); // Premier joystick branché
15    }
16
17    @Override
18    public void teleopPeriodic() {
19
20        double hauteur = pot.get();
21        double puissance = 0.0;
22
23        if(hauteur > HAUTEUR_MIN && hauteur < HAUTEUR_MAX) {
24
25            if(joystick.getRawButton(1))
26                puissance = 0.5;
27
28            else if(joystick.getRawButton(2))
29                puissance = -0.5;
30
31        }
32
33        elevateur.set(puissance);
34
35    }
36
37 }
```

Dans cet exemple, le robot comporte un élévateur et un potentiomètre. Si la valeur du potentiomètre (la hauteur) est comprise entre les bornes 0.2 et 1.8, on permet au joystick de faire monter ou descendre l'élévateur. Autrement, la valeur envoyée au moteur (la variable puissance) reste à zéro, ce qui arrête le moteur.

## 7.3 Les gyroscopes

Les gyroscopes mesurent la rotation (en degrés) effectuée par le robot. Certains sont analogues et ne mesurent la rotation qu'autour d'un seul axe. Il est donc important de les fixer perpendiculairement au sol.

```
AnalogGyro gyro = new AnalogGyro(0); // Port analogue 0
double angle = gyro.getAngle(); // Retourne la rotation angulaire (en °)
```

Les gyroscopes mesurent un déplacement **relatif** : ils se réinitialisent à zéro lorsque l'on éteint le robot ou lorsqu'on appelle leur méthode `reset()`. Leur sensibilité peut également être réglée avec la méthode `setSensitivity(double)`<sup>2</sup>.

Certains gyros ne sont pas analogues : c'est le cas du modèle ADXRS450\_Gyro, qui se branche dans le port SPI du RoboRIO.

```
ADXRS450_Gyro gyro = new ADXRS450_Gyro(); // Port SPI
double angle = gyro.getAngle(); // Retourne la rotation angulaire (en °)
```

De plus, certains gyros plus complexes mesurent la rotation autour de 3 axes différents.

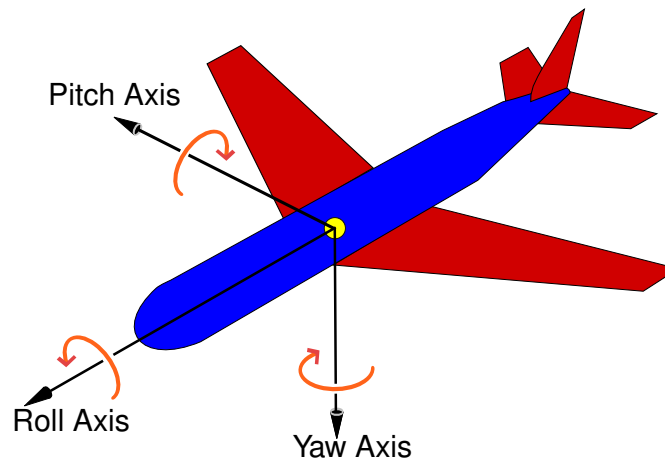


FIGURE 7.1 – Les angles mesurés par un gyroscope à 3 axes (X, Y et Z)<sup>3</sup>

Habituellement, l'angle qui nous intéresse est celui représenté par l'axe yaw. Il faut donc faire quelques tests pour déterminer s'il s'agit de l'axe des X, des Y ou des Z de notre gyro.

Un des ces capteurs est l'ADIS16448\_IMU. Voir 7.5.

2. La sensibilité est indiquée dans les spécifications du gyro : <http://wpilib.screenstepslive.com/s/currentCS/m/java/1/599713-gyros-measuring-rotation-and-controlling-robot-driving-direction>

3. By Yaw\_Axis.svg : Auawisederivative work : Jrvz (Yaw\_Axis.svg) [CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0>)], via Wikimedia Commons.

### CODE 7.3 — Utilisation d'un gyro

```
1 public class UtilisationGyro extends TimedRobot {
2
3     private ADXRS450_Gyro gyro;
4     private VictorSP moteurGauche, moteurDroit;
5     private DifferentialDrive drive;
6
7     @Override
8     public void robotInit() {
9
10        gyro = new ADXRS450_Gyro(); // Port SPI
11        gyro.calibrate();
12
13        moteurGauche = new VictorSP(0); // Port PWM 0
14        moteurDroit = new VictorSP(1); // Port PWM 1
15        drive = new DifferentialDrive(moteurGauche, moteurDroit);
16
17    }
18
19    @Override
20    public void autonomousInit() {
21        gyro.reset();
22    }
23
24    @Override
25    public void autonomousPeriodic() {
26
27        if(gyro.getAngle() <= 90.0)
28            drive.arcadeDrive(0.0, 0.75); // avancer 0.0, tourner 0.75
29
30        else
31            drive.setMotors(0.0, 0.0);
32
33    }
34
35 }
```

Dans le programme précédent, le mode autonome fait effectuer au robot une rotation (puissance à 75 %) jusqu'à ce que le robot atteigne 90°. Il est important d'appeler la méthode `calibrate()` lorsque le robot est immobile pour bien calibrer le gyro. De plus, on appelle la méthode `reset()` dans `autonomousInit` pour s'assurer que l'angle mesuré est à zéro au début du mode autonome, car il est possible que le robot ait été déplacé entre le moment où il a été allumé et le début du match.



#### CODE 7.4 — Aller en ligne droite avec un gyro

```
1 public class LigneDroiteGyro extends TimedRobot {
2
3     // Constante de correction
4     private static final double K_ANGLE = -0.1;
5
6     private ADXRS450_Gyro gyro;
7     private VictorSP moteurGauche, moteurDroit;
8     private DifferentialDrive drive;
9
10    @Override
11    public void robotInit() {
12
13        gyro = new ADXRS450_Gyro(); // Port SPI
14        gyro.calibrate();
15
16        moteurGauche = new VictorSP(0); // Port PWM 0
17        moteurDroit = new VictorSP(1); // Port PWM 1
18        drive = new DifferentialDrive(moteurGauche, moteurDroit);
19    }
20
21
22    @Override
23    public void autonomousInit() {
24        gyro.reset();
25    }
26
27    @Override
28    public void autonomousPeriodic() {
29        drive.arcadeDrive(0.5, K_ANGLE * gyro.getAngle());
30    }
31
32 }
```

Ce programme fait avancer le robot en ligne droite pendant tout le mode autonome. On pourrait penser que l'instruction `drive.arcadeDrive(0.5, 0.0)` (50 % de puissance vers l'avant, et 0 % de rotation) est suffisante pour faire avancer le robot. Cependant, le robot va dévier dès que les forces de chaque moteur ne sont pas tout à fait les mêmes ou que le poids du robot n'est pas réparti parfaitement.

Pour corriger la rotation, on envoie comme paramètre une valeur qui dépend de l'angle mesuré par le gyro.

Angle du gyro	Valeur de rotation envoyée
-3°	0.3
-2°	0.2
-1°	0.1
0°	0
1°	-0.1
2°	-0.2
etc.	

Si l'angle est de zéro (le robot est droit), alors  $K\_ANGLE \times 0 = -0.1 \times 0 = 0$ . Si l'angle est de 1°, alors la valeur -0.1 sera envoyé, etc. La constante K\_ANGLE fait varier la rapidité de la correction : il faut l'ajuster selon la réaction de notre robot.

## 7.4 Les encodeurs

Les encodeurs sont des capteurs qui mesurent la rotation **relative** de composantes du robot. Contrairement aux potentiomètres, les encodeurs se réinitialisent lorsque l'on éteint le robot. Par contre, ils n'ont aucune limite de rotation : c'est pourquoi on les utilise, par exemple, pour mesurer la distance parcourue (nombre de tours) par les roues de la base pilotable.

## CODE 7.5 — Utilisation d'un encodeur

```
1 public class UtilisationEncodeur extends TimedRobot {
2
3     private Encoder encoder;
4     private VictorSP moteurGauche, moteurDroit;
5     private DifferentialDrive drive;
6
7     @Override
8     public void robotInit() {
9
10        encoder = Encoder(2, 3); // Ports digitaux 2 et 3
11        encoder.setDistancePerPulse(0.0002262);
12
13        moteurGauche = new VictorSP(0); // Port PWM 0
14        moteurDroit = new VictorSP(1); // Port PWM 1
15        drive = new DifferentialDrive(moteurGauche, moteurDroit);
16
17    }
18
19    @Override
20    public void autonomousInit() {
21        encoder.reset();
22    }
23
24    @Override
25    public void autonomousPeriodic() {
26
27        if(encoder.getDistance() <= 2.0)
28            drive.arcadeDrive(0.5, 0.0); // avancer 50%, tourner 0.0
29
30        else
31            drive.setMotors(0.0, 0.0);
32
33    }
34
35 }
```

Dans ce programme autonome, le robot avance de 2 mètres, puis s'immobilise. Comme avec le gyro, on appelle la méthode `reset()` dans `autonomousInit()` pour réinitialiser l'encodeur.

En réalité, un encodeur ne calcule pas de distance ; il envoie au RoboRIO un signal (un *pulse*) à chaque fois qu'il tourne de quelques degrés. Pour transformer ces *pulses* en distance réelle (par exemple, en mètres), on utilise la méthode `setDistancePerPulse(double)`. La valeur donnée en paramètre doit être calculée selon les spécifications de l'encodeur et le diamètre de nos roues. C'est

pourquoi la méthode `getDistance()` retourne une distance en mètres.

## 7.5 ADIS16448\_IMU

Ce capteur est bien plus qu'un gyro : c'est un IMU (*Inertial Measurement Unit*), c'est-à-dire qu'il mesure différentes. Cette classe n'est pas incluse dans WPILib : elle peut être [télécharger ici](#).

```
ADIS16448_IMU adis = new ADIS16448_IMU();  
// Il faut tester pour savoir lesquels correspondent à X, Y et Z.  
double angleX = adis.getRoll();  
double angleY = adis.getPitch();  
double angleZ = adis.getYaw();  
  
double accelX = adis.getAccelX();  
double accelY = adis.getAccelY();  
double accelZ = adis.getAccelZ();
```

Ce capteur peut également mesurer le champ magnétique dans les trois axes, la pression barométrique et la température.

## 7.6 Les autres capteurs

Les capteurs utilisés en FRC sont multiples : de nouveaux modèles sont créés chaque année et il est de notre responsabilité de s'informer de ce qui nous est offert. La documentation de WPILib est également l'endroit tout désigné pour en apprendre davantage. En voici quelques-uns que nous utilisons plus rarement.

### 7.6.1 Les accéléromètres

Un accéléromètre, comme son nom l'indique, mesure l'accélération. Comme les gyros, il la mesure habituellement dans trois différents axes (X, Y et Z). Par exemple, on peut l'utiliser pour détecter si le robot fonce dans un obstacle (décélération vers l'arrière). Le RoboRIO est équipé d'un accéléromètre de base : on y accède avec la classe `BuiltInAccelerometer`.

```
BuiltInAccelerometer accel = new BuiltInAccelerometer();  
  
double accelX = accel.getX();  
double accelY = accel.getY();  
double accelZ = accel.getZ();
```

## 7.6.2 Les capteurs ultrasons

Un capteur ultrason mesure la distance entre lui et un obstacle. Ils ont une portée limitée. Certains sont branchés à deux ports digitaux (DIO).

```
Ultrasonic ultra = new Ultrasonic(8, 9); // Ports DIO 8 et 9
ultra.setAutomaticMode(true); // Démarre l'envoi d'ultrasons
double dist = ultra.getRangeInches();
```

D'autres modèles sont analogues : ils retournent donc un voltage.

```
AnalogInput ultra = new AnalogInput(3); // Port analogue 3
double distance = VOLT_VERS_CM * ultra.getAverageVoltage();
```

Dans cet exemple, on multiplie le voltage mesuré par une constante (différente pour chaque modèle) afin de convertir les volts en centimètres.

# Chapitre 8

## Commandes et sous-systèmes

Jusqu'à présent, tous vos programmes étaient contenus dans une seule classe : la classe `Robot`. Bien que ce soit adéquat pour des projets simples, un vrai robot est beaucoup plus complexe. Il serait d'ailleurs assez difficile de travailler en équipe sur un seul fichier. C'est pourquoi nous aborderons la création de commandes (*command based programming*).

### 8.1 Une nouvelle façon d'organiser nos idées

Cette manière de programmer est basée deux concepts principaux : les commandes et les sous-systèmes.

Un **sous-système** est un regroupement logique de différentes composantes du robot. Par exemple :

- les deux moteurs qui font rouler le robot peuvent former le sous-système `BasePilotable` ;
- le moteur relié au shooter et l'encodeur qui mesure sa vitesse peuvent former le sous-système `Shooter` ;
- les moteurs qui servent à prendre un ballon et la *limit switch* qui détecte si le robot en possède un peuvent former le sous-système `Intake`.

Chaque composante du robot doit être contenu par un seul sous-système. C'est le rôle du programmeur de bien les répartir. Cette tâche délicate peut avoir de graves conséquences sur le programme : nous aborderons pourquoi plus bas.

Une **commande** est une action que peut effectuer le robot à l'aide d'un ou plusieurs sous-systèmes. Par exemple :

- `AvancerLigneDroite` est une commande qui utilise le sous-système `BasePilotable` ;
- `LancerBalles` est une commande qui utilise le sous-système `Shooter` ;
- `PrendreBallon` est une commande qui utilise le sous-système `Intake` ;
- `AvancerEtLancerBalles` est une commande qui utilise les sous-systèmes `BasePilotable` et `Shooter`.

Chaque commande peut, par exemple, être lié à un bouton, qui activera la commande lorsqu'il est appuyé.

Attention ! Un sous-système peut être utilisé par une seule commande à la fois, ce qui est tout à fait logique. Par exemple, on ne veut pas que les commandes PrendreBallon et LancerBallon puissent s'exécuter en même temps, car elles utilisent toutes les deux le sous-système Intake. Il s'agit de la principale contrainte qu'on doit garder en tête lorsque l'on modélise les sous-systèmes de notre robot.

## 8.2 Créer un projet à base de commandes

Référez-vous à la section 5.1. À l'option `Select a project base`, choisissez `Command robot`.

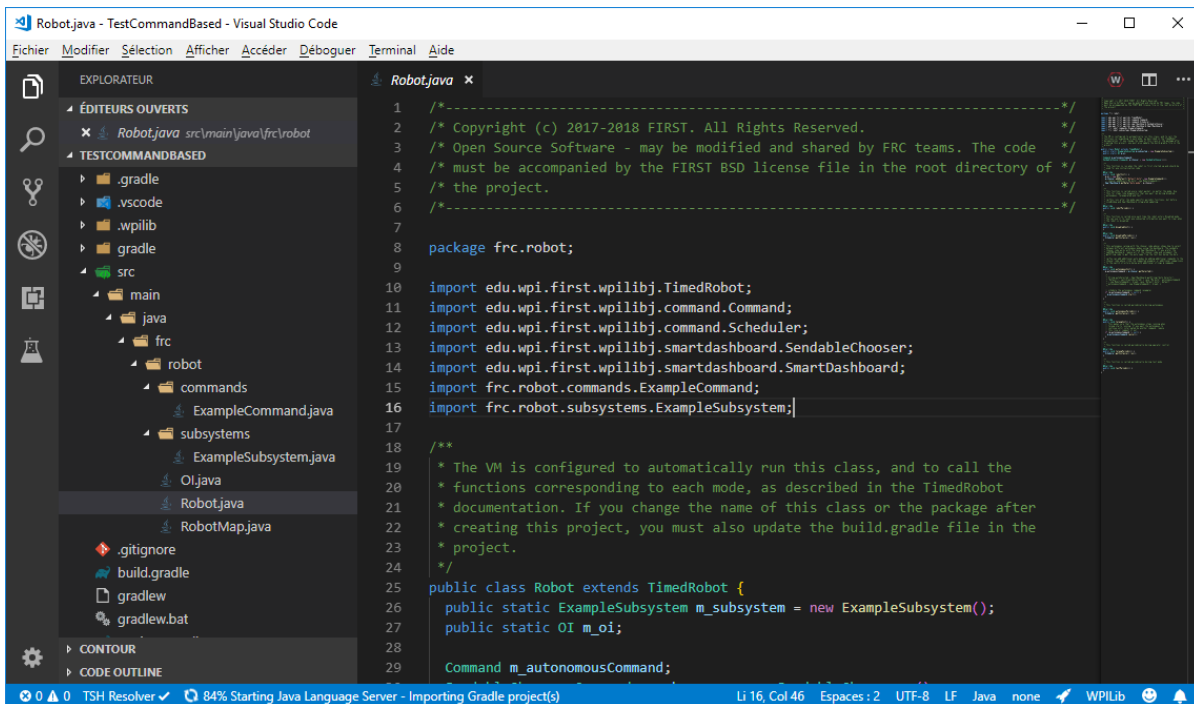


FIGURE 8.1 – La hiérarchie d'un nouveau projet *Command robot*.

## 8.3 Créer un sous-système

1. Dans le dossier `src`, cliquez à droite sur le dossier `subsystems`, puis sélectionnez `Create a new class/command`.
2. Dans le menu affiché, sélectionnez `Subsystem`.
3. Saisissez le nom de votre sous-système. Il doit commencer par une majuscule et être sans espace.

Comme on le faisait dans la classe Robot, on doit ajouter les composantes de notre sous-système dans le haut de la classe. Voici un exemple avec un sous-système nommé BasePilotable.

## CODE 8.1 — Les attributs et le constructeur d'un sous-système

```
1  /**
2   * Sous-système de l'intake.
3   * Son rôle est de prendre et relâcher un ballon.
4   * Il comprend un moteur, un encodeur (pour calculer la vitesse de
5   * rotation du moteur) et une limit switch, pour savoir
6   * lorsqu'un ballon a été attrapé.
7   */
8  public class Intake extends Subsystem {
9
10     private VictorSP moteur;
11     private Encoder encoder;
12     private DigitalInput limitSwitch;
13
14     public Intake() {
15
16         super("Intake");
17
18         moteur = new VictorSP(3);
19         addChild("Moteur", moteur);
20
21         encoder = new Encoder(0, 1);
22         addChild("Encoder", encoder);
23
24         limitSwitch = new DigitalInput(2);
25         addChild("Limit switch", limitSwitch);
26
27     }
28
29     @Override
30     public void initDefaultCommand() {
31         // Set the default command for a subsystem here.
32         // setDefaultCommand(new MySpecialCommand());
33     }
34 }
```

### 8.3.1 Le constructeur

Après avoir déclaré les trois attributs dans le haut de notre classe, nous lesinstancions (avec `new`) dans ce qu'on appelle le **constructeur** de notre classe. Il s'agit d'une méthode publique, sans type de retour, et dont le nom est le même que la classe. Comme son nom l'indique, le constructeur est appelé lorsque notre intake sera construit dans la mémoire du RoboRIO. C'est un peu l'équivalent



de `robotInit`, mais pour un sous-système.

Dans le constructeur, la première chose à faire est d'appeler la méthode `super(String)` (le super constructeur). Cette méthode reçoit une chaîne de caractères représentant le nom de notre sous-système. Ce `String` est en quelque sorte une étiquette qui est attribué à notre sous-système pour mieux le repérer lorsque nous aurons à faire du débogage.

Par la suite, on instancie les attributs de notre sous-système. Remarquez que pour chaque composante, on appelle également la méthode `addChild`. Son rôle est d'associer une étiquette à notre composante et de spécifier que c'est un « enfant » (*child*) de notre sous-système. Cela rendra également la structure de notre robot plus claire lors du débogage.

### 8.3.2 Ajouter une commande par défaut

Lorsqu'aucune commande en cours d'exécution n'utilise un sous-système, celui-ci peut démarrer une commande automatiquement. On appelle cette commande la **commande par défaut du sous-système**. Par exemple, la commande `Piloter` est habituellement la commande par défaut du sous-système `BasePilotable`. Cependant, plusieurs sous-systèmes n'en ont pas besoin.

Pour ajouter une commande par défaut à un sous-système, il suffit d'utiliser `setDefaultCommand` dans la méthode `initDefaultCommand` (voir l'exemple [8.3](#)).

### 8.3.3 Ajouter des méthodes à un sous-système

## 8.4 Créer une commande

### 8.4.1 Le constructeur

La méthode `initialise()`

La méthode `execute()`

La méthode `isFinished()`

La méthode `end()`

La méthode `interrupted()`

Ajouter un délai avec `setTimeout(double)`

### 8.4.2 Lier un bouton à une commande dans la classe `OI`

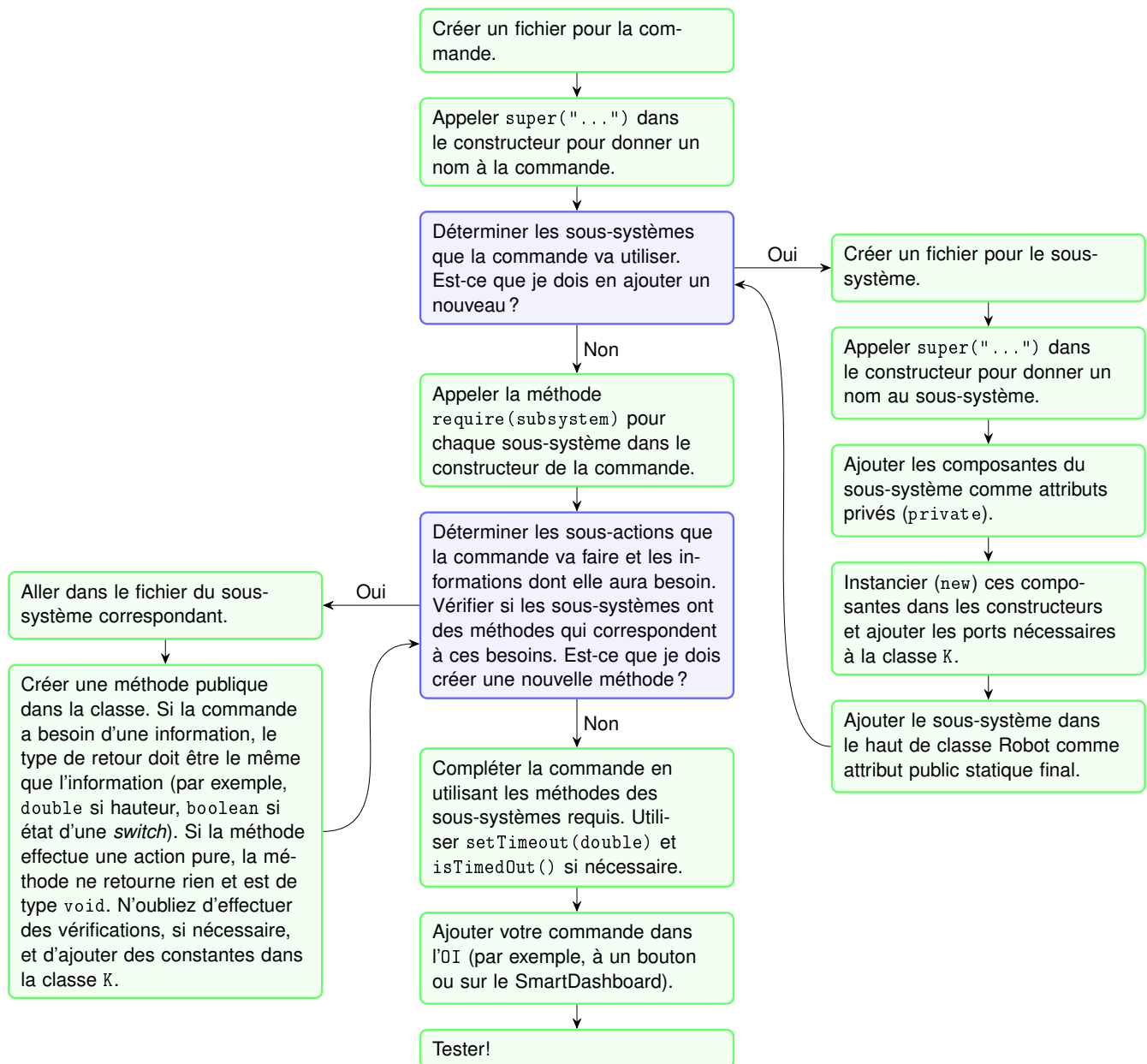
## 8.5 Les groupes de commandes

`addsequential`, `addparallel`

`Waitcommand`, `WaitForchildren`

Créer des sous-groupes de commandes

## 8.5.1 Le processus de création d'une commande



## **Troisième partie**

# **Git et les logiciels de gestion de versions**