Guide

Formation Java et WPILib

Étienne Beaulac
Ultime FRC 5528

Dernière modification
26 septembre 2018

Table des matières

Table des extraits de code

Table des figures

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 [1]). Cependant, à la base, les ordinateurs ne comprennent que le binaire (Alan Turing, 1936 [2]). Pour se simplifier la vie, les informaticiens ont créé des langages intermédiaires qui font le pont entre nous et les ordinateurs (Grace Hopper, 1951 [3]). 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 Visual Studio Code

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 Visual Studio Code [1] avec l'extension WPILib fournie par FIRST.

Visual Studio Code est disponible gratuitement sur code.visualstudio.com/. Vous devrez également vous assurer d'avoir installé une version récente du JDK 8 (*Java Development Kit*). Les étapes d'installation sont également détaillées ici.

Visual Studio Code 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 intéresserons le plus :

Explorateur	Cette fenêtre regroupe les fichiers de votre projet, subdivisés en dossiers et paquetages (*packages*), jusqu'aux fichiers Java.
Fenêtre d'édition	Cette fenêtre affiche tous les fichiers que vous êtes en train d'éditer, vous permettant facilement de naviguer entre différents documents.
Structure	Cette section affiche la structure du fichier en cours d'édition.
Problèmes	Comme son nom l'indique, on y retrouve une liste de tous les avertissements et erreurs concernant votre projet. Chaque item précise la nature de l'erreur et où elle se trouve.
Débogage	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.

L'interface est personnalisable, à vous de l'adapter comme il vous plaira !

1. Pour plus d'informations concernant Visual Studio Code, consultez code.visualstudio.com/docs.

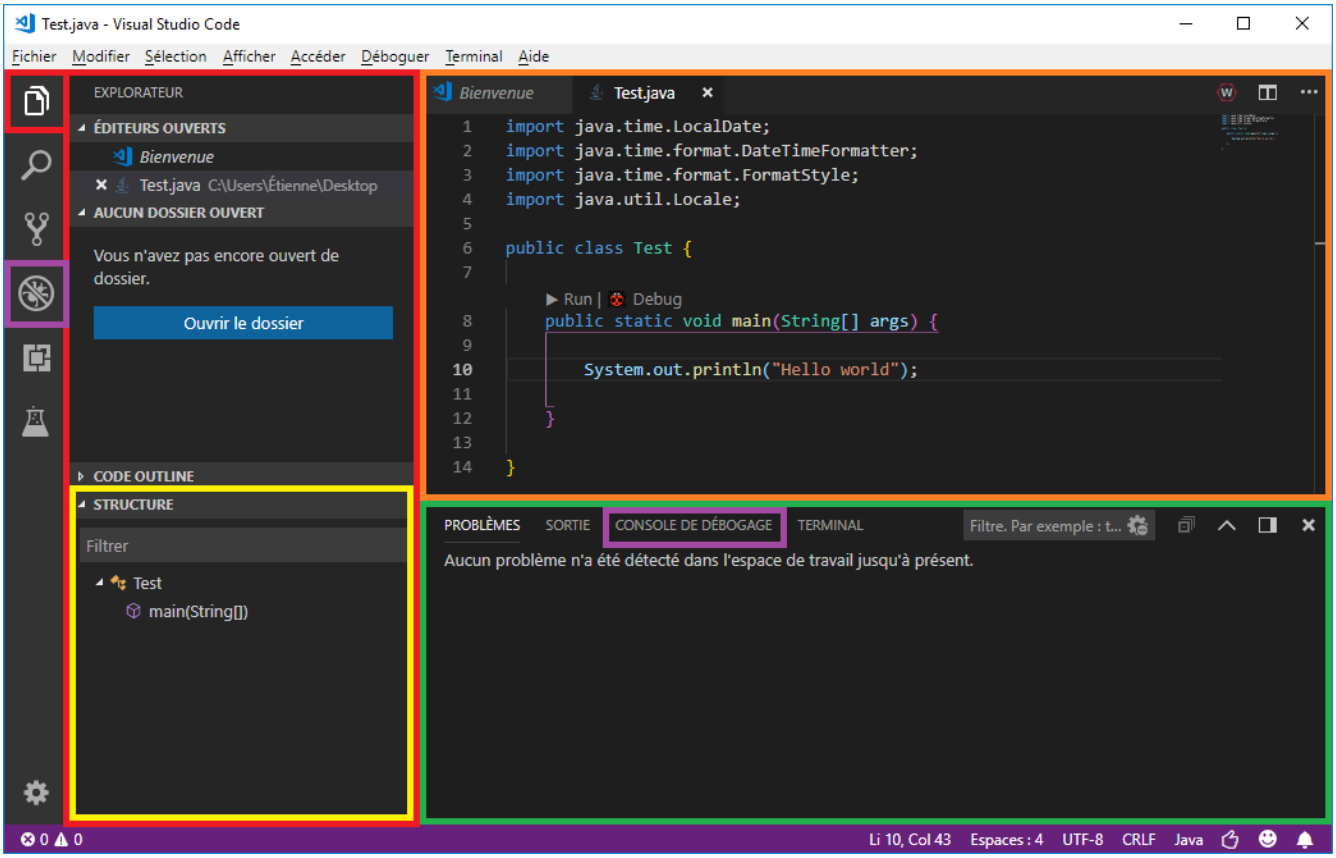
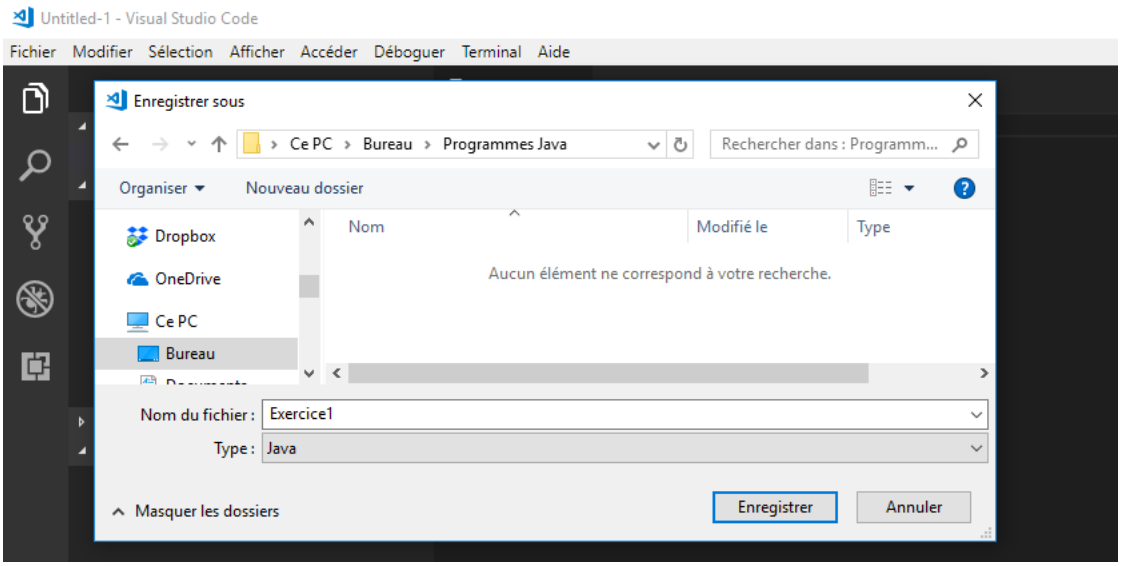


FIGURE 2.1 – L'interface principale de Visual Studio Code.

2.2 Création d'un programme Java

1. Créez un dossier vide qui contiendra votre projet.
2. Faites un clic droit sur votre dossier, puis sélectionnez `Open with Code` . Vous pouvez aussi choisir `Fichier > Ouvrir le dossier...` , directement dans VS Code.
3. Dans VS Code, créez un nouveau fichier avec `Fichier > Nouveau fichier` . Un fichier vide sera créé.

4. 

   Enregistrez votre fichier dans le même dossier avec Fichier > Enregistrer sous... , ou Ctrl + S . Donnez un nom à votre programme (sans espace débutant par une majuscule) et sélectionner Java comme type de fichier. Cliquez sur Enregister .

5. Complétez votre fichier avec l'exemple suivant.

CODE 2.1 — Programme de base

```
public class Exercice1 {

    public static void main(String[] args) {

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

    }

}
```

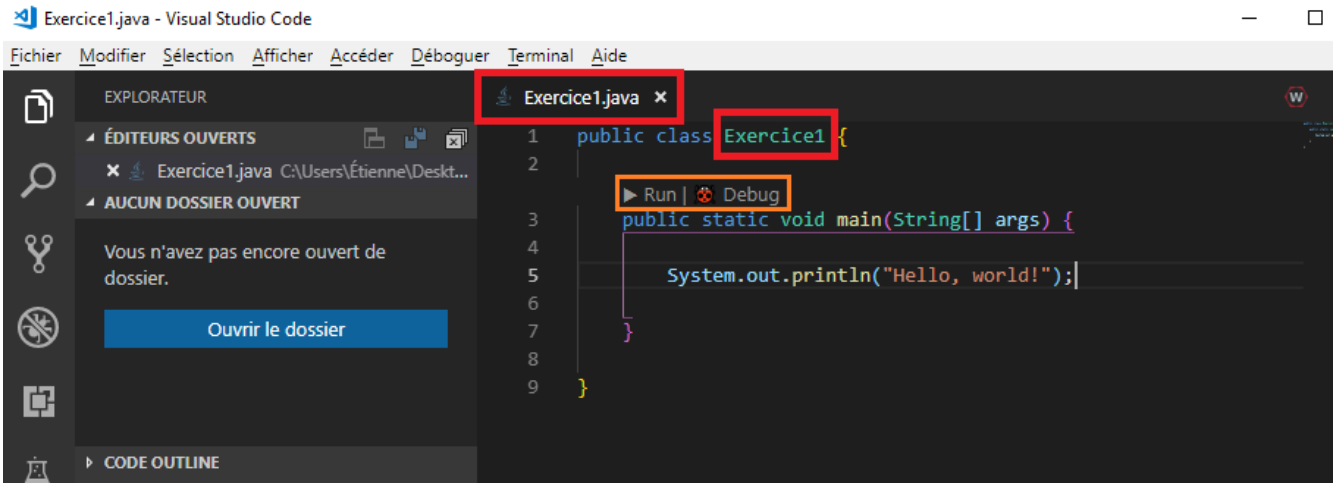


FIGURE 2.2 – Compiler, exécuter et déboguer un programme avec VS Code.

Attention ! Il est très important que le nom de votre classe (à côté du mot `class`) soit le même que le nom de votre fichier (par exemple, *Exercice1*).

En orange, le bouton Run vous permet de compiler et d'exécuter votre programme, tandis que Debug lance votre programme en mode débogage. En appuyant sur l'un ou l'autre, vous devriez voir du texte apparaître dans votre console.

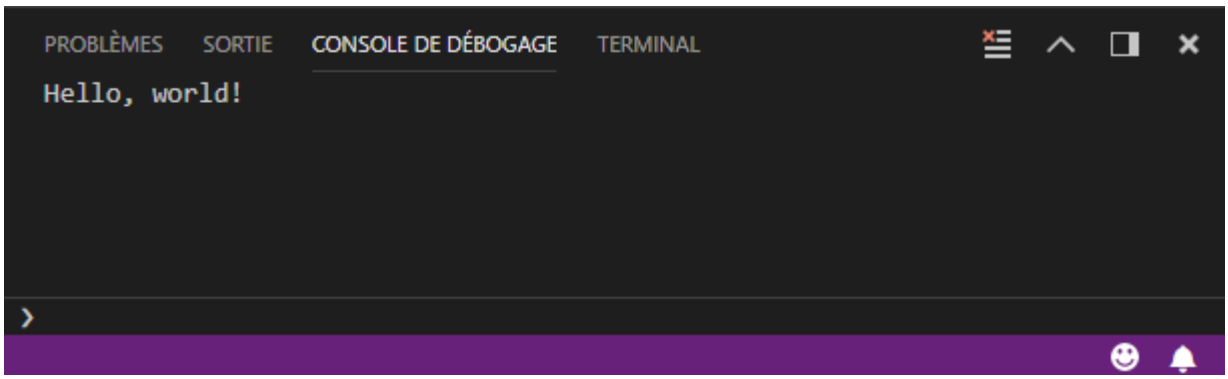


FIGURE 2.3 – Écriture dans la console.

Si la console n'est pas déjà affichée, il suffit de cliquer sur Afficher > Console de débogage , ou Ctrl + Shift + Y .

Il est également possible d'ouvrir automatiquement la console lorsque vous démarrez un programme. Cliquez sur Fichier > Préférences > Paramètres , ou Ctrl + , . Dans la barre de recherche, saisissez open console . À l'option *Contrôle le moment où la console de débogage interne doit s'ouvrir*, choisissez openOnSessionStart .

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. VS Code s'en occupe automatiquement la plupart du temps. Dans votre fichier, vous pouvez également faire Clic droit > Mettre en forme le document pour corriger l'espacement et l'indentation.

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

```java
/*
 * La classe suivante affiche un message
 * dans la console.
 */
public class MonPremierProgramme {

    /*  Fonction principale
        du programme.        */
    public static void main(String[] args) {

        //Début du programme

        System.out.println("Hello, world"); //Affichage du message

    }

}
```

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 :

```java
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 !

Pour afficher la Javadoc d'un élément, il suffit de laisser son curseur au-dessus de lui pendant quelques instants.

CODE 2.3 — Ajout de commentaires Javadoc

```
/**
 * Ceci est un commentaire Javadoc!
 * @author Etienne
 *
 */
public class MonPremierProgramme { ... }
```

FIGURE 2.4 – Visualisation de la Javadoc dans VS Code

La Javadoc possède plusieurs attributs spéciaux débutant par un arrobe `@` . 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.

2.8 Afficher les caractères internationaux dans VS Code

— Cliquez sur Fichier > Préférences > Paramètres , ou Ctrl + , .
— Dans la barre de recherche, saisissez vmargs , puis cliquez sur Modifier dans settings.json du premier résultat.
— Cliquez sur le crayon pour modifier la propriété sélectionnée, puis sur Remplacer dans les paramètres .
— À la fin de la chaîne de caractères sélectionnée, ajouter `-Dfile.encoding=UTF-8` .
— Sauvegardez avec Ctrl + S .
— Lorsque VS Code vous propose de redémarrer, sélectionner Restart now .

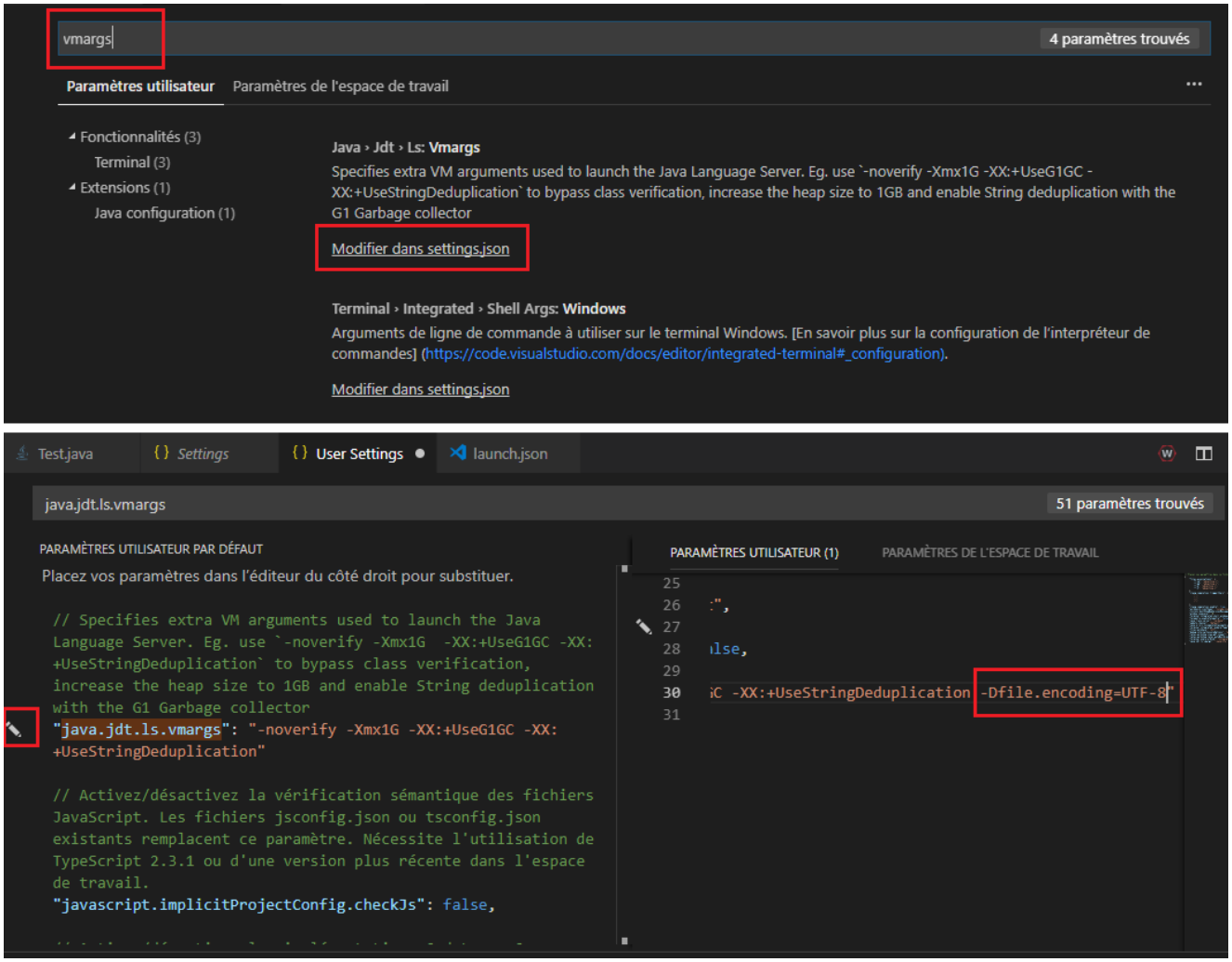


FIGURE 2.5 – Afficher des caractères internationaux dans la console avec VS Code.

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

```java
/**
 * Affiche des noms dans la console.
 *
 * @author Etienne
 */
public class AffichageNom {

    public static void main(String[] args) {

        String nom = "Étienne"; // Nom de l'utilisateur

        System.out.println("Je m'appelle " + nom + "!"); // Affichage

        nom = "Alexandre"; // Nouvelle valeur

        System.out.println("Je m'appelle maintenant " + nom + "!"); //
        ↪ Affichage de la nouvelle valeur

    }

}
```

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 (⇐) dans la même instruction

```
type nomVariable = valeur;
```

Déclaration, puis affectation (⇐) 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`, `vitessexGauche1` 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.

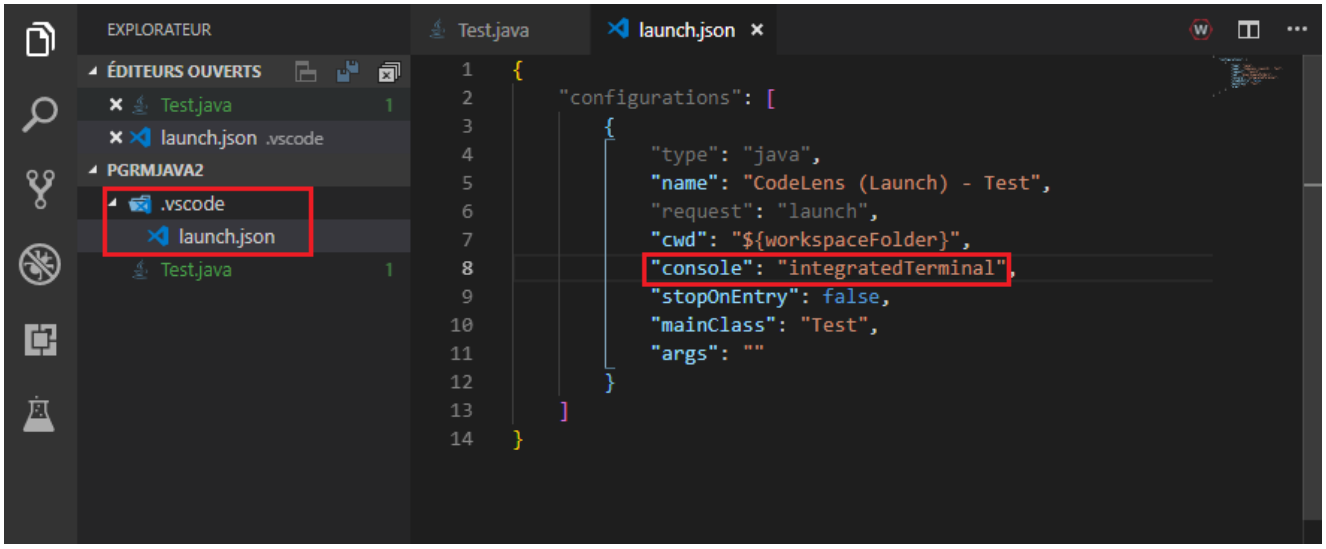


FIGURE 3.2 – Configuration nécessaire pour lire la console dans VS Code.

CODE 3.2 — Demander et afficher un nom

```java
import java.util.Scanner;

/**
 * Demande le nom de l'utilisateur, puis l'affiche.
 *
 * @author Etienne
 */
public class DemanderNom {

    public static void main(String[] args) {

        String nom;
        Scanner scanner = new Scanner(System.in);

        //Demander le nom
        System.out.print("Saisissez votre nom : ");
        nom = scanner.nextLine();

        //Affichage
        System.out.println("Votre nom est " + nom + "!");

    }

}
```

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
`int`	Un nombre entier.	`int ageUtilisateur = 20;`
`double`	Un nombre à virgules de précision double.	`double prix = 19.95;`
`boolean`	Une valeur booléenne (**true** ou **false**).	`boolean estOuvert = true;`

FIGURE 3.3 – 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

```
/**
 * Affiche des données de type primitif.
 *
 * @author Etienne
 */
public class AffichagePrimitif {

    public static void main(String[] args) {

        int age = 14, ageAmi = 13;
        double taille = 1.45;

        System.out.println("J'ai " + age + " ans!");
        System.out.println("Mon ami a " + ageAmi + " ans.");
        System.out.println("Je mesure " + taille + " m.");

    }

}
```

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

```
import java.util.Scanner;

/**
 * Demande l'âge et la taille de l'utilisateur,
 * puis l'affiche dans la console.
 *
 * @author Etienne
 */
public class AgeTaille {

    public static void main(String[] args) {

        int age;
        double taille;
        Scanner scanner = new Scanner(System.in);

        //Demande de l'âge
        System.out.print("Saisissez votre âge : ");
        age = scanner.nextInt();

        //Demande de la taille
        System.out.print("Saisissez votre taille : ");
        taille = scanner.nextDouble();

        //Affichage
        System.out.println("Vous avez " + age + " ans et mesurez " +
        ↪  taille + " m.");

    }

}
```

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 3.4 – Les opérateurs arithmétiques.

CODE 3.5 — Années avant la majorité

```java
import java.util.Scanner;

/**
 * Demande l'âge de l'utilisateur et affiche le nombre
 * d'années avant qu'il soit majeur.
 *
 * @author Etienne
 */
public class Majorite {

    public static void main(String[] args) {

        int age;
        final int AGE_MAJORITE = 18; // L'âge de la majorité est fixe
        Scanner scanner = new Scanner(System.in);

        //Demander l'âge
        System.out.print("Saisissez votre âge : ");
        age = scanner.nextInt();

        //Âge avant majorité
        System.out.println("Vous serez majeur dans " + (AGE_MAJORITE -
        ↪  age) + " ans.");

    }

}
```

CODE 3.6 — Liquidation d'un inventaire

```java
import java.util.Scanner;

/**
 * Gestion de la liquidation
 * d'un inventaire.
 *
 * @author Etienne
 */
public class Liquidation {

    public static void main(String[] args) {

        int joursRestants; // Nombre de jours avant la fermeture

        int nombreItems; // Nombre d'items restants en inventaire
        double prixItem; // Prix d'un item
        double prixTotal; // Prix d'un item avec les taxes
        double valeurInventaire; // Valeur de tous les items

        final double TAXES = 1.15; // Taxes fixes de 15%

        Scanner scanner = new Scanner(System.in);


        // Nombre de jours restants
        System.out.print("Nombre de jours avant la fermeture : ");
        joursRestants = scanner.nextInt();

        // Nombre d'items
        System.out.print("Nombre d'items restants : ");
        nombreItems = scanner.nextInt();

        // Prix d'un item
        System.out.print("Prix d'un item : ");
        prixItem = scanner.nextDouble();

        // Prix total d'un item
        prixTotal = prixItem * TAXES;

        // Valeur de l'inventaire
        valeurInventaire = prixItem * nombreItems;
```

```java
        // Affichage
        System.out.println("Le prix à payer avec taxes est de " +
        ↪ prixTotal + "$.");
        System.out.println("Votre inventaire vaut " + valeurInventaire +
        ↪ "$.");
        System.out.println("Pour tout liquider, vous devrez vendre
        ↪ environ " + (nombreItems / joursRestants) + " items par
        ↪ jour.");

    }
}
```

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...	`+=`	`age += 2;`	`age = age + 2;`
Retrait de...	`-=`	`vie -= dommages;`	`vie = vie - dommages;`
Multiplier par...	`*=`	`prix *= rabais;`	`prix = prix * rabais;`
Diviser par...	`/=`	`taille /= 3.28;`	`taille = taille / 3.28;`
Incrémentation	`++`	`compteur++;`	`compteur += 1;`
Décrémentation	`--`	`points--;`	`points -= 1;`

FIGURE 3.5 – 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

```java
import java.util.Scanner;

/**
 * Affiche la valeur absolue et le cube d'un nombre.
 *
 * @author Etienne
 *
 */
public class TestMath {

    public static void main(String[] args) {

        double nombre;
        Scanner scanner = new Scanner(System.in);

        //Obtention du nombre
        System.out.print("Saisissez un nombre : ");
        nombre = scanner.nextDouble();

        //Calculs
        System.out.println("\nLa valeur absolue du nombre est : " +
        ↪ Math.abs(nombre));
        System.out.println("Le cube du nombre est : " + Math.pow(nombre,
        ↪ 3));

    }

}
```

La méthode `pow()` prend deux paramètres : le premier est la base et le deuxième est l'exposant. Ainsi, `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 continu 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	`==`	`a == 2`
n'est pas égal	`!=`	`prix != 10`
est plus grand	`>`	`rabais > prix`
est plus grand ou égal	`>=`	`age >= AGE_MAJORITE`
est plus petit	`<`	`rotation < 25.1`
est plus petit ou égal	`<=`	`distance <= 120`
ou	`||`	`a == 10 || b == 5`
et	`&&`	`distance >= 10 && angle == 180`
n'est pas	`!`	`!estOuvert`

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

```java
import java.time.LocalDate;
import java.util.Scanner;
/**
 * Validation d'une année de naissance.
 *
 * @author Etienne
 */
public class ValidationNaissance {

    public static void main(String[] args) {

        int annee;
        final int ANNEE_MINIMALE = 1900;
        final int ANNEE_COURANTE = LocalDate.now().getYear();
        Scanner scanner = new Scanner(System.in);


        //Obtention de l'année
        System.out.print("Saisissez une année de naissance : ");
        annee = scanner.nextInt();


        //Validation
        if(annee >= ANNEE_MINIMALE && annee <= ANNEE_COURANTE) {
            System.out.println("Année valide.");
        }
        else {
            System.out.println("L'année " + annee + " est invalide.");
            System.out.println("Vous devez recommencer!");
        }

    }

}
```

4.3 Les *else if* multiples

Le *if else* standard permet de gérer deux possibilités en même temps. Dans certains, on voudrait pouvoir gérer plusieurs cas exclusifs entre eux.

Il est possible d'ajouter un *if* immédiatement après un *else*. Il n'y a donc plus de limite au nombre de

possibilités !

CODE 4.2 — MessageAge.java

```java
/**
 * Affiche un message selon l'âge de l'utilisateur.
 */
public class MessageAge {

    public static void main(String[] args) {

        final int AGE_PRIMAIRE = 5;
        final int AGE_SECONDAIRE = 12;
        final int AGE_MAJORITE = 18;

        int age;

        Scanner scanner = new Scanner(System.in);

        // Obtention de l'âge
        System.out.print("Saisissez votre âge : ");
        age = scanner.nextInt();

        // Message selon l'âge
        if(age < 0) {
            System.out.println("Âge invalide!");
        }
        else if(age < AGE_PRIMAIRE) {
            System.out.println("Pas encore à l'école!");
        }
        else if(age < AGE_SECONDAIRE) {
            System.out.println("Au primaire!");
        }
        else if(age < AGE_MAJORITE) {
            System.out.println("Au secondaire!");
        }
        else {
            System.out.println("Vous êtes majeur!");
        }
    }
}
```

Comme les *else if* sont imbriqués, **un seul cas** sera exécuté. Les conditions des *if* sont vérifiés dans le même ordre qu'ils ont été écrits.

Par exemple, dans l'exemple précédent, supposons que l'âge est de 10 ans. Si on les vérifie indivi-

duellement, les deux dernières conditions sont vraies (`age < AGE_SECONDAIRE` et `age < AGE_MAJORITE`). Cependant, la première condition vraie est `age < AGE_SECONDAIRE`, c'est donc ce *if* qui sera exécuté.

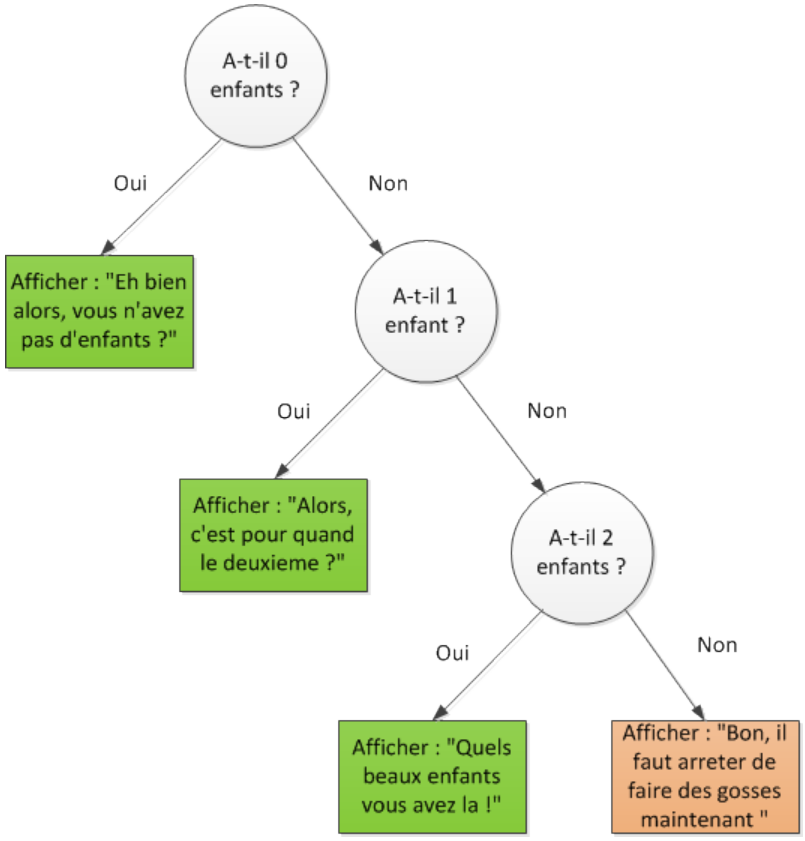


FIGURE 4.2 – La structure de *else if* multiples.

Chapitre 5

Les méthodes

Manuel de référence : p. 69 à 77

5.1 L'appel de méthodes

Jusqu'à présent, vous avez utilisé quelques méthodes prédéfinies. Certaines reçoivent des paramètres entre leurs parenthèses, d'autres non. Certaines retournent une valeur, d'autres non.

En Java, une **méthode** est une instruction qui réalise une série d'opérations définies ailleurs. 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 `println()`, `nextLine()` et toutes les méthodes de classe `Math` (voir 3.7).

Utiliser une méthode

Utiliser une méthode sur un objet (instancié avec `new`)

```
objet.nomMethode(param1, param2, ...);
typeRetour valeurRetournee = objet.nomMethode(param1, param2, ...);
```

Utiliser une méthode sur une classe (méthode statique)

```
NomClasse.nomMethode(param1, param2, ...);
typeRetour valeurRetournee = NomClasse.nomMethode(param1, param2, ...);
```

5.2 Créer une méthode

CODE 5.1 — Programme sans méthode

```
public class Bonjour {

    public static void main(String[] args) {

        System.out.println("Bonjour Étienne!");
        System.out.println("Bonjour Camille!");
        System.out.println("Bonjour Alexandre!");
        System.out.println("Bonjour Chaymae!");

    }

}
```

Vous aurez sûrement remarqué qu'il vous arrive de réécrire la même opération plusieurs fois. Au lieu de copier-coller ces instructions, vous pouvez déplacer ce bout de code dans une méthode externe, puis l'utiliser comme bon vous semble.

CODE 5.2 — Programme avec méthode

```
public class Bonjour {

    public static void main(String[] args) {

        direBonjour("Étienne");
        direBonjour("Camille");
        direBonjour("Alexandre");
        direBonjour("Chaymae");

    }

    public static void direBonjour(String nom) {
        System.out.println("Bonjour " + nom + "!");
    }

}
```

Ainsi, si vous décidez de remplacer `"Bonjour"` par `"Bonne journée"`, il vous faudra le remplacer seulement une fois, et non quatre.

Déclarer une méthode

```
public [static] typeRetour nomMéthode (param1, param2, ...) {
   instruction1;
   ... // Autres instructions ou calculs
   return valeur; // Facultatif, dépend du type de retour
}
```

Remarquez que la méthode `direBonjour` ne retourne aucune valeur. C'est pourquoi son type de retour est `void`. C'est aussi le cas de la méthode `println()`.

Des méthodes peuvent, par exemple, effectuer un calcul et retourner un `double` ou `int`. D'autres peuvent vérifier une condition et retourner un `boolean`.

Les méthodes, comme le `main`, doivent être à l'intérieur d'une classe. Faites attention à bien placer vos accolades.

```
// Converti une durée en secondes
public static int nombreDeSecondes(int heures, int minutes, int secondes) {

    int duree = 0;

    duree += 60 * 60 * heures;
    duree += 60 * minutes;
    duree += secondes;

    return duree;

}
```

```
// Vérifie si une personne est majeure
public static int nombreDeSecondes(int age) {

    return (age >= 18);

}
```

Deuxième partie

La librairie WPILib

Chapitre 6

La classe `TimedRobot`

6.1 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`.

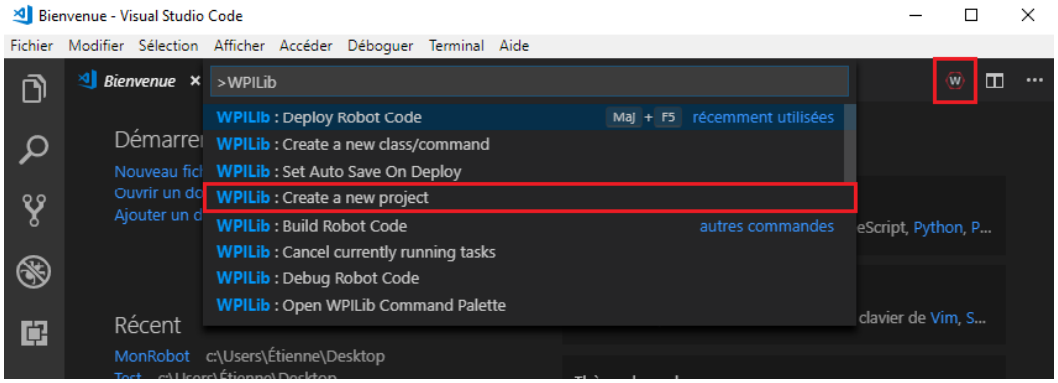


FIGURE 6.1 – Le menu principal de l'extension WPILib.

1. Dans VS Code, cliquez sur le bouton de l'extension WPILib.
2. Sélectionnez WPILib : Create a new project .
3. Comme type de projet, sélectionnez Template .
4. Comme langage, sélectionnez Java .
5. Comme projet de base, sélectionnez Timed Robot .
6. Sélectionnez le dossier de sauvegarde de votre projet.
7. Saisissez le nom de votre projet (sans espace et débutant par une majuscule).
8. Saisissez le numéro de l'équipe.
9. Cliquez sur Generate project .

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

6.2 Les méthodes `Init` et `Periodic`

La classe `Robot` que VS Code a générée pour vous contient déjà quelques méthodes prédéfinies. Chacune d'entre elles a un rôle bien précis.

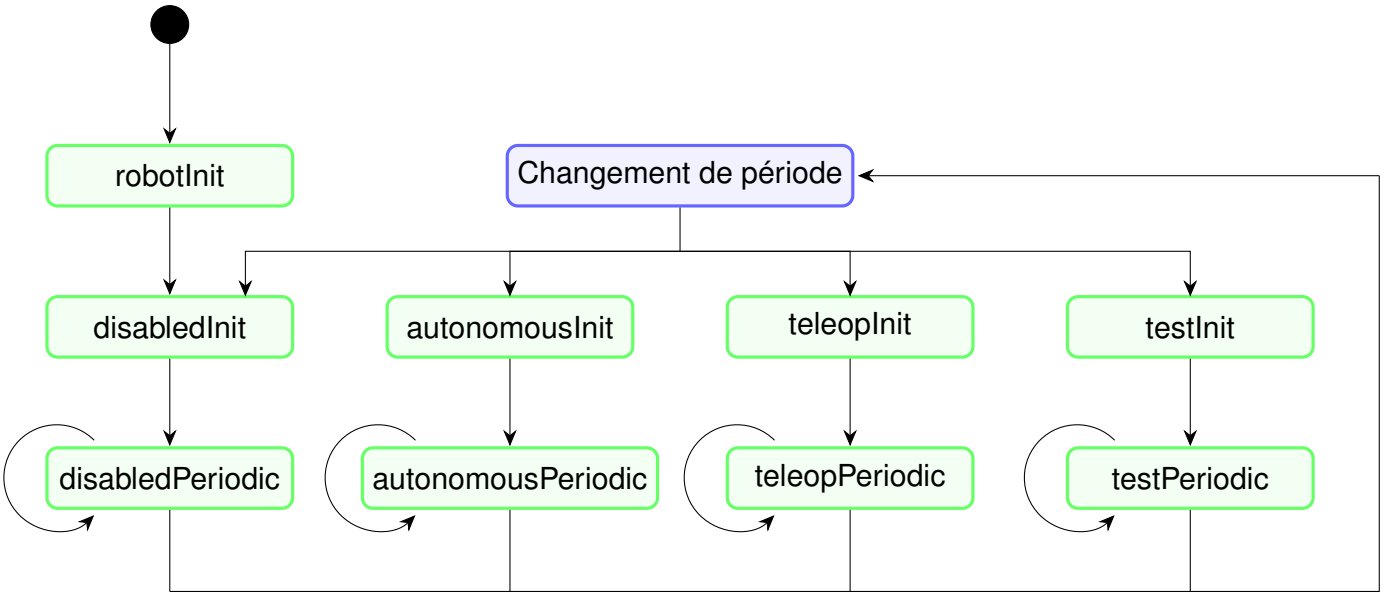


FIGURE 6.2 – Séquence d'éxécution des méthodes de la classe Robot.

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.

6.3 Faire avancer le robot

CODE 6.1 — Utilisation de DifferentialDrive avec Joystick

```
public class MonPremierRobot extends TimedRobot {

    private VictorSP moteurGauche, moteurDroit;
    private DifferentialDrive basePilotable;
    private Joystick joystick;

    @Override
    public void robotInit() {
        moteurGauche = new VictorSP(0);
        moteurDroit = new VictorSP(1);
        basePilotable = new DifferentialDrive(moteurGauche,
        ↪  moteurDroit);
        joystick = new Joystick(0);
    }

    @Override
    public void autonomousPeriodic() {
        moteurGauche.set(0.5);
        moteurDroit.set(-0.5);
    }


    @Override
    public void teleopPeriodic() {
        basePilotable.arcadeDrive(-1 * joystick.getY(),
        ↪  joystick.getX());
    }

}
```

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 contrôleur le robot naturellement.

6.3.1 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.

6.3.2 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).

6.3.3 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 Joytick 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 7

Les actionneurs [1]

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.

7.1 Les moteurs

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

7.2 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 envoit. 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

7.2.1 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 7.1 — Utilisation d'un solénoïde simple

```
public class UtilisationSolenoideSimple extends TimedRobot {

    private Solenoid solenoid;
    private Joystick joystick;

    @Override
    public void robotInit() {
        // Branché sur le port 2 du PCM
        solenoid = new Solenoid(2);
        joystick = new Joystick(0);
    }

    @Override
    public void teleopPeriodic() {

        if(joystick.getRawButton(1)) {

            solenoid.set(true);

        } else {

            solenoid.set(false);

        }

    }

}
```

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

7.2.2 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 7.2 — Utilisation d'un solénoïde double

```
public class UtilisationSolenoideDouble extends TimedRobot {

    private DoubleSolenoid solenoid;
    private Joystick joystick;

    @Override
    public void robotInit() {
        solenoid = new DoubleSolenoid(3, 4); // Ports 3 et 4 du PCM
        joystick = new Joystick(0);
    }

    @Override
    public void teleopPeriodic() {

        if(joystick.getRawButton(1))
            solenoid.set(DoubleSolenoid.Value.kForward);

        else if(joystick.getRawButton(2))
            solenoid.set(DoubleSolenoid.Value.kReverse);

        else
            solenoid.set(DoubleSolenoid.Value.kOff);

    }

}
```

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 ballel
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 indépendemment les deux valves du solénoïde.

7.3 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 7.3 — Utilisation d'un servomoteur

```
public class UtilisationServo extends TimedRobot {

    private Servo servo;
    private Joystick joystick;

    @Override
    public void robotInit() {
        servo = Servo(8); // Branché sur le port PWM 8 du RoboRIO
        joystick = new Joystick(0);
    }

    @Override
    public void teleopPeriodic() {

        if(joystick.getRawButton(1))
            servo.setAngle(0.0);

        else if(joystick.getRawButton(3))
            solenoid.setAngle(90.0);

    }

}
```

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 8

Les capteurs [1]

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

8.1 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 8.1 — Utilisation d'un interrupteur de fin de course (limit switch)

```
public class UtilisationSwitch extends TimedRobot {

    private DigitalInput limitSwitch;
    private VictorSP ramasseBallons;

    @Override
    public void robotInit() {
        limitSwitch = DigitalInput(2); // Port digital 2
        ramasseBallons = new VictorSP(1); // Port PWM 1
    }

    @Override
    public void autonomousPeriodic() {

        if(limitSwitch.get())
            ramasseBallons.set(0.0);

        else
            ramasseBallons.set(1.0).

    }

}
```

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()`.

8.2 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 8.2 — Utilisation d'un potentiomètre

```
public class UtilisationPotentiometre extends TimedRobot {

    private final static double HAUTEUR_MIN = 0.2;
    private final static double HAUTEUR_MAX = 1.8;

    private AnalogPotentiometer pot;
    private VictorSP elevateur;
    private Joystick joystick;

    @Override
    public void robotInit() {
        pot = new AnalogPotentiometer(0); // Port analogue 0
        elevateur = new VictorSP(0); // Port PWM 0
        joystick = new Joystick(0); // Premier joystick branché
    }

    @Override
    public void teleopPeriodic() {

        double hauteur = pot.get();
        double puissance = 0.0;

        if(hauteur > HAUTEUR_MIN && hauteur < HAUTEUR_MAX) {

            if(joystick.getRawButton(1))
                puissance = 0.5;

            else if(joystick.getRawButton(2))
                puissance = -0.5;

        }

        elevateur.set(puissance);

    }

}
```

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.

8.3 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)` [2].

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.

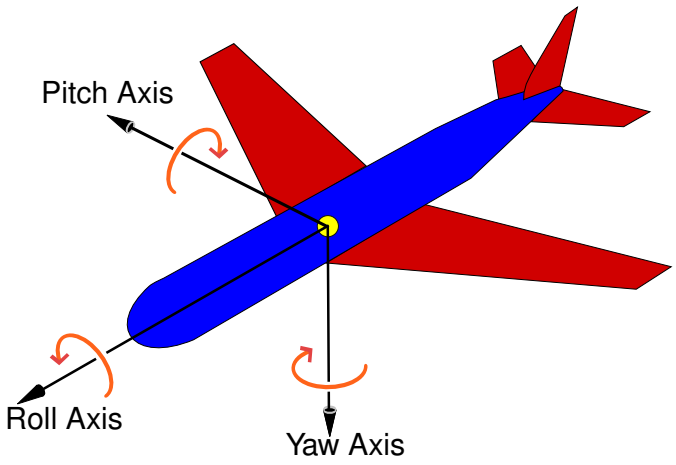


FIGURE 8.1 – Les angles mesurés par un gyroscope à 3 axes (X, Y et Z) [3]

Habituellement, l'angle qui nous intéresse et 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 8.5.

2. La sensibilité est indiquée dans les spécifications du gyro : http://wpilib.screenstepslive.com/s/currentCS/m/java/l/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 8.3 — Utilisation d'un gyro

```java
public class UtilisationGyro extends TimedRobot {

    private ADXRS450_Gyro gyro;
    private VictorSP moteurGauche, moteurDroit;
    private DifferentialDrive drive;

    @Override
    public void robotInit() {

        gyro = new ADXRS450_Gyro(); // Port SPI
        gyro.calibrate();

        moteurGauche = new VictorSP(0); // Port PWM 0
        moteurDroit = new VictorSP(1); // Port PWM 1
        drive = new DifferentialDrive(moteurGauche, moteurDroit);

    }

    @Override
    public void autonomousInit() {
        gyro.reset();
    }

    @Override
    public void autonomousPeriodic() {

        if(gyro.getAngle() <= 90.0)
            drive.arcadeDrive(0.0, 0.75); // avancer 0.0, tourner 0.75

        else
            drive.setMotors(0.0, 0.0);

    }

}
```

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 8.4 — Aller en ligne droite avec un gyro

```java
public class LigneDroiteGyro extends TimedRobot {

    // Constante de correction
    private static final double K_ANGLE = -0.1;

    private ADXRS450_Gyro gyro;
    private VictorSP moteurGauche, moteurDroit;
    private DifferentialDrive drive;

    @Override
    public void robotInit() {

        gyro = new ADXRS450_Gyro(); // Port SPI
        gyro.calibrate();

        moteurGauche = new VictorSP(0); // Port PWM 0
        moteurDroit = new VictorSP(1); // Port PWM 1
        drive = new DifferentialDrive(moteurGauche, moteurDroit);

    }

    @Override
    public void autonomousInit() {
        gyro.reset();
    }

    @Override
    public void autonomousPeriodic() {
        drive.arcadeDrive(0.5, K_ANGLE * gyro.getAngle());
    }

}
```

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 envoit 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.

8.4 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 8.5 — Utilisation d'un encodeur

```
public class UtilisationEncodeur extends TimedRobot {

    private Encoder encoder;
    private VictorSP moteurGauche, moteurDroit;
    private DifferentialDrive drive;

    @Override
    public void robotInit() {

        encoder = Encoder(2, 3); // Ports digitaux 2 et 3
        encoder.setDistancePerPulse(0.0002262);

        moteurGauche = new VictorSP(0); // Port PWM 0
        moteurDroit = new VictorSP(1); // Port PWM 1
        drive = new DifferentialDrive(moteurGauche, moteurDroit);

    }

    @Override
    public void autonomousInit() {
        encoder.reset();
    }

    @Override
    public void autonomousPeriodic() {

        if(encoder.getDistance() <= 2.0)
            drive.arcadeDrive(0.5, 0.0); // avancer 50%, tourner 0.0

        else
            drive.setMotors(0.0, 0.0);

    }

}
```

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.

8.5 ADIS16448_IMU

Ce capteur est bien plus qu'un gyro, c'est un IMU (*Inertial Measurement Unit*). À lui seul, il peut mesure la rotation, l'accélération, le champ magnétique, la température et la pression barométrique. 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();
```

8.6 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();
```

8.7 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 multiple le voltage mesuré par une constante (différente pour chaque modèle) afin de convertir les volts en centimètres.

8.8 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. Pour en apprendre davantage, n'hésitez pas à consulter la Javadoc de WPILib et son code source.

Chapitre 9

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*).

9.1 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.

9.2 Créer un projet à base de commandes

Référez-vous à la section 6.1. À l'option Select a project base , choisissez Command robot .

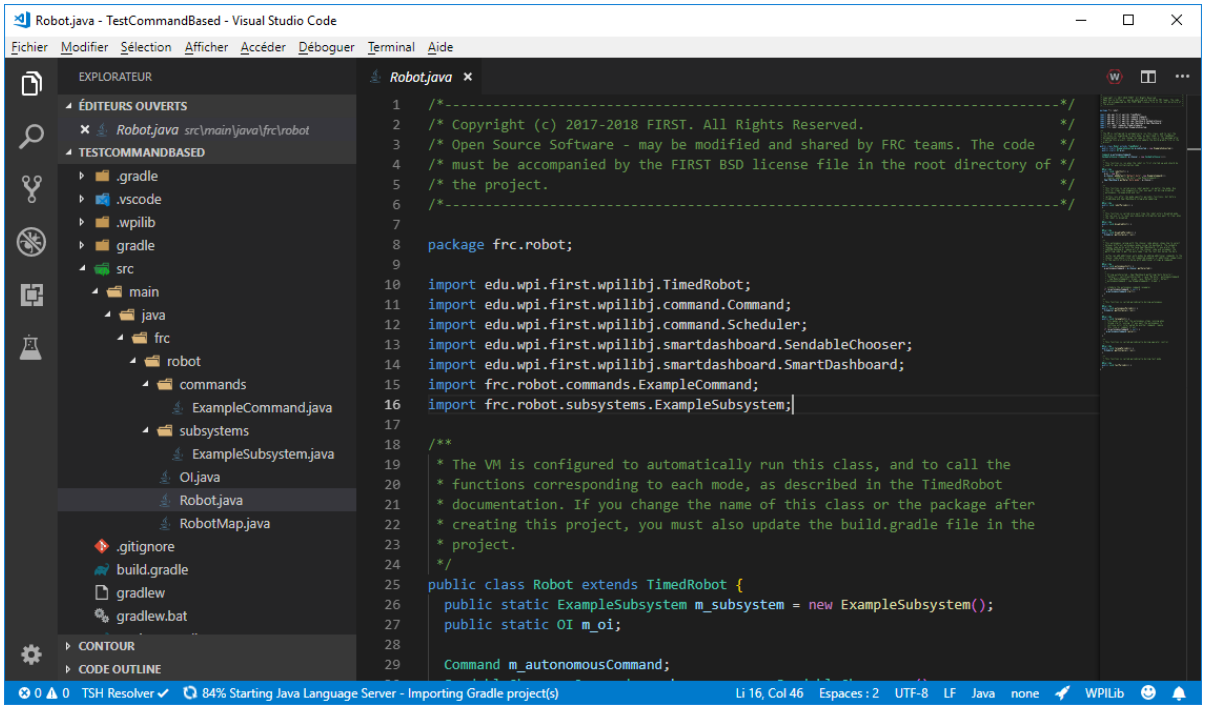


FIGURE 9.1 – La hiérarchie d'un nouveau projet *Command robot*.

9.3 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 `Intake`, dont le rôle est de prendre et lancer un ballon.

CODE 9.1 — Les attributs et le constructeur d'un sous-système

```java
/**
 * Sous-système de l'intake.
 * Son rôle est de prendre et relâcher un ballon.
 * Il comprend un moteur, un encodeur (pour calculer la vitesse de
 * rotation du moteur) et une limit switch, pour savoir
 * lorsqu'un ballon a été attrapé.
 */
public class Intake extends Subsystem {

  private VictorSP moteur;
  private Encoder encoder;
  private DigitalInput limitSwitch;

  public Intake() {

    moteur = new VictorSP(3);
    addChild("Moteur", moteur);

    encoder = new Encoder(0, 1);
    addChild("Encoder", encoder);

    limitSwitch = new DigitalInput(2);
    addChild("Limit switch", limitSwitch);

  }

  @Override
  public void initDefaultCommand() {
    // Set the default command for a subsystem here.
    // setDefaultCommand(new MySpecialCommand());
  }

  public void prendreBallon() {
    moteur.set(0.5);
  }

  public void lancerBallon() {
    moteur.set(-0.5);
  }

  public void arreterMoteur() {
    moteur.set(0.0);
```

```
    }

    public boolean aUnBallon() {
      return limitSwitch.get();
    }

    public double getVitesse() {
      return encoder.getRate();
    }

  }
```

9.3.1 Le constructeur

Après avoir déclaré les trois attributs dans le haut de notre classe, nous les instancions (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.

```
// Le constructeur d'un sous-système
public Intake() {
    // ...
}
```

Dans le constructeur, 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 la structure de notre robot plus claire lors du débogage.

```
// Le constructeur d'un sous-système
public Intake() {

    moteur = new VictorSP(3);
    addChild("Moteur", moteur);
    // ...
}
```

9.3.2 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`.

```
public class BasePilotable extends Subsystem {
  // ...
  @Override
  public void initDefaultCommand() {
    setDefaultCommand(new Piloter());
  }
}
```

9.3.3 Ajouter des méthodes à un sous-système

Un sous-système n'est pas complet tant qu'on ne lui ajoute pas de méthodes. Les méthodes d'un sous-système sont habituellement publiques et ont la forme suivante.

La déclaration d'une méthode dans un sous-système

```
public typeRetour nomMéthode () {
   instruction1 ;
   ... // Autres instructions ou calculs
   return valeur; // Facultatif, dépend du type de retour
}
```

Un sous-système a habituellement deux types de méthodes : celles qui effectuent des actions, et celles qui retournent une information.

La majorité des méthodes qui effectuent des actions ne retournent pas d'information. Leur type de retour est donc `void`, pour indiquer qu'elles ne retournent rien.

Par exemple, on veut que notre sous-système ait comme fonctionnalité de prendre un ballon. On lui ajoute la méthode `prendreBallon()`. Pour prendre un ballon, on doit activer son moteur. Cette méthode effectue une action pure : elle n'a pas a retourner d'information. C'est pourquoi son type de retour est `void`.

```
public void prendreBallon() {
  moteur.set(0.5);
}
```

On veut également que notre sous-système puisse nous dire lorsqu'il a un ballon ou non. Cette méthode n'effectuera pas d'action : son rôle est uniquement de nous donner une information. Comme l'information qui nous intéresse est de type booléen, ce sera le type de retour de notre méthode.

```
public boolean aUnBallon() {
  return limitSwitch.get();
}
```

Pour savoir si l'intake a un ballon, il suffit de retourner l'état de notre `limit switch`.

9.4 Créer une commande

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

Votre classe sera créée avec plusieurs méthodes vides. Voici un exemple utilisant le sous-système que nous avons créé précédemment.

CODE 9.2 — Création d'une commande

```
public class LancerBallon extends Command {
  public LancerBallon() {
    // Use requires() here to declare subsystem dependencies
    requires(Robot.intake);
  }

  // Called just before this Command runs the first time
  @Override
  protected void initialize() {
  }

  // Called repeatedly when this Command is scheduled to run
  @Override
  protected void execute() {
    Robot.intake.lancerBallon();
  }

  // Make this return true when this Command no longer needs to run
   ↪  execute()
  @Override
```

```
    protected boolean isFinished() {
      return Robot.intake.aUnBallon();
    }

    // Called once after isFinished returns true
    @Override
    protected void end() {
      Robot.intake.arreterMoteur();
    }

    // Called when another command which requires one or more of the same
    // subsystems is scheduled to run
    @Override
    protected void interrupted() {
      end();
    }
  }
```

9.4.1 Le constructeur

Dans le constructeur, on doit spécifier les sous-systèmes que la commande utilise. Pour ce faire, on appelle la méthode `requires` pour chaque sous-système utilisé.

```
public LancerBallon() {
    requires(Robot.intake);
    // Autres sous-systèmes, si nécessaire
    // requires(Robot.basePilotable);
    // ...
}
```

9.4.2 Cycle de vie d'une commande

Le cycle de vie d'une commande est entièrement géré par le `Scheduler`. C'est lui qui décide quand les commandes sont démarrées et arrêtées.

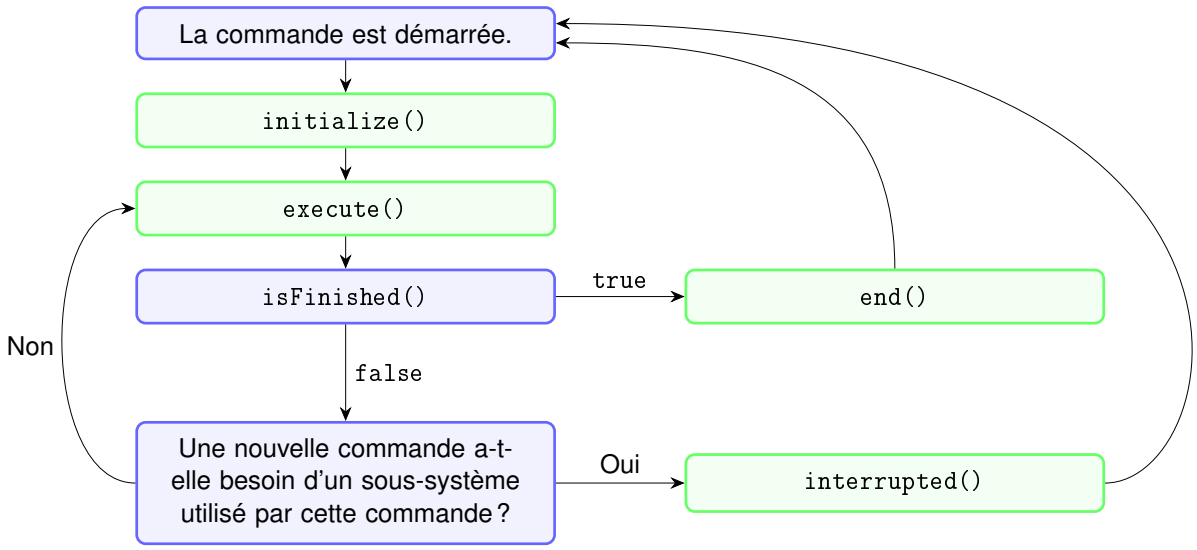


FIGURE 9.2 – Le cycle de vie d'une commande.

9.4.3 La méthode `initialize()`

La méthode `initialize()` est appelé une seule fois, lorsque la commande est démarrée. C'est le bon endroit pour remettre à zéro des capteurs comme des encodeurs ou un gyro.

```
public void initialize() {
    // On réinitialise le gyro au début de la commande
    Robot.basePilotable.resetGyro();
}
```

9.4.4 La méthode `execute()`

La méthode `execute()` est le coeur de la commande. Elle sera exécutée en boucle, tant que la commande est exécutée.

```
public void execute() {
    // Tant que la commande s'exécute, on active le moteur pour prendre un ballon
    Robot.intake.prendreBallon();
}
```

9.4.5 La méthode `isFinished()`

Contrairement aux autres, la méthode `isFinished()` retourne un booléen. Le système s'en sert pour déterminer si une commande est terminée. Si la méthode retourne `true`, alors la commande

s'arrêtera par elle-même. Sinon, elle continue, et `execute()` sera appelée une nouvelle fois. Dans notre exemple, on veut que la commande s'arrête dès que l'intake a attrapé un ballon.

```
@Override
public boolean isFinished() {
  return Robot.intake.aUnBallon();
}
```

Dans certains cas, on veut que la commande ne se termine jamais par elle-même. C'est le cas de la commande de pilotage. Dans ce cas, on retourne toujours `false`.

```
@Override
public boolean isFinished() {
  return false;
}
```

Si on veut que la commande se termine instantanément, il suffit de retourner `true`.

9.4.6 La méthode `end()`

La méthode `end()` est appelée lorsque la commande s'est terminée par elle-même. Par exemple, on peut donc y arrêter les moteurs qui étaient en cours d'utilisation.

```
@Override
public void end() {
  Robot.intake.arreterMoteur();
}
```

9.4.7 La méthode `interrupted()`

La méthode `interrupted()` est appelée lorsque la commande est interrompue par une autre commande et qu'elle doit s'arrêter abruptement. Elle est souvent similaire à `end()`. On peut donc la réutiliser.

```
@Override
public void interrupted() {
  end();
}
```

9.4.8 Ajouter un délai avec `setTimeout(double)` et `isTimedOut()`

Il est souvent utile de spécifier un délai à une commande. Par exemple, en mode autonome, on pourrait avoir besoin d'une commande qui fait avancer le robot en ligne droite pendant 3 secondes.

CODE 9.3 — Commande avec délai

```
public class AvancerLigneDroite extends Command {

  public AvancerLigneDroite() {
    requires(Robot.basePilotable);
    setTimeout(3);
  }

  @Override
  protected void initialize() {
  }

  @Override
  protected void execute() {
    Robot.basePilotable.avancer();
  }

  @Override
  protected boolean isFinished() {
    return isTimedOut();
  }

  @Override
  protected void end() {
    Robot.basePilotable.arreterMoteurs();
  }

  @Override
  protected void interrupted() {
    end();
  }
}
```

On spécifie le délai dans le constructeur avec la méthode `setTimeout(double)`. Le nombre passé en paramètres correspond au délai, en secondes. La commande se terminera donc après 3 secondes.

```
setTimeout(3.0);
```

Il faut également vérifier si le délai a été dépassé dans la méthode `isFinished()`.

```
@Override
protected boolean isFinished() {
  return isTimedOut();
}
```

On peut combiner un délai à une autre condition. Dans l'exemple suivant, la commande se termine si on dépasse le délai ou si on attrape un ballon.

```
@Override
protected boolean isFinished() {
  return isTimedOut() || Robot.intake.aUnBallon();
}
```

9.4.9 Lier un bouton à une commande dans la classe `OI`

Par convention, les boutons sont liés aux commandes dans la classe `OI` (l'interface opérateur, *operator interface* en anglais).

CODE 9.4 — L'interface opérateur

```
public class OI {

  private Joystick joystick;
  private JoystickButton button1;
  private JoystickButton button2;

  public OI() {

    joystick = new Joystick(0);

    button1 = new JoystickButton(joystick, 1);
    button1.whileHeld(new PrendreBallon());

    button2 = new JoystickButton(joystick, 2);
    button2.whenPressed(new LancerBallon());

  }

}
```

Dans un premier temps, on déclare les joysticks et les boutons comme attributs privés au haut de

notre classe. Ensuite, dans le constructeur, on les intancie. Le constructeur d'un `JoystickButton` reçoit en paramètres le `Joystick` auquel il est lié et son numéro.

Pour démarrer une commande au moment où un bouton est appuyé, on utilise la méthode `whenPressed`.

```
button2.whenPressed(new LancerBallon());
```

Dans certains cas, on veut qu'une commande soit exécutée tant qu'on maintient le bouton appuyé. On utilise alors la méthode `whileHeld`.

```
button1.whileHeld(new PrendreBallon());
```

De plus, il est possible de démarrer une commande en appuyant sur un bouton, puis de l'annuler en réappuyant sur le même bouton. Le bouton agit alors comme un interrupteur. La méthode `toggleWhenPressed` a été créé pour répondre à ce besoin.

```
button1.toggleWhenPressed(new PrendreBallon());
```

9.5 Créer un groupe de commandes

Les commandes simples peuvent être groupées ensemble pour former des séquences plus complexes. On les appelle des **groupes de commandes** (*CommandGroup*).

1. Dans le dossier src , cliquez à droite sur le dossier commands , puis sélectionnez Create a new class/command .
2. Dans le menu affiché, sélectionnez Command Group .
3. Saisissez le nom de votre groupe de commandes. Il doit commencer par une majuscule et être sans espace.

CODE 9.5 — Groupe de commandes avec addSequential

```
public class LancerBallonHaut {

    public LancerBallonHaut() {

        addSequential(new LeverElevateur());
        addSequential(new LancerBallon());
        addSequential(new DescendreElevateur());

    }

}
```

Dans un groupe de commandes, il suffit d'ajouter dans le bon ordre les commandes qu'on souhaite démarrer. La méthode `addSequential` ajoute les commandes `séquentiellement`, c'est-à-dire que l'exécution du groupe est « bloqué » jusqu'à ce que la sous-commande soit terminée. Dans l'exemple ci-dessus, le robot lève l'élévateur, il lance le ballon, puis il redescend l'élévateur. Ces actions sont réalisées l'une à la suite de l'autre.

CODE 9.6 — Groupe de commandes avec addParallel

```
public class Autonome1 {

    public Autonome1() {

        addParallel(new Avancer1metre());
        addParallel(new LeverElevateur());

    }

}
```

Contrairement à `addSequential`, la méthode `addParallel` ne bloque pas l'exécution du groupe. Les commandes s'exécutent donc **parallèlement** (en même temps). Dans cet exemple, on lève l'élévateur en même temps de faire avancer la base pilotable. Cela nous permet d'économiser du temps, ce qui est essentiel en mode autonome ! Nécessairement, il faut que les commandes ajoutées en parallèle n'utilisent pas les mêmes sous-systèmes.

CODE 9.7 — Groupe de commandes avec WaitForChildren et WaitCommand

```
public class Autonome2 {

    public Autonome2() {

        addParallel(new Avancer1metre());
        addParallel(new LeverElevateur());

        addSequential(new WaitForChildren());
        addSequential(new WaitCommand(2));

        addSequential(new LancerBallon());

        addParallel(new DescendreElevateur());
        addParallel(new Reculer1metre());

    }

}
```

WPILib nous fournit deux commandes très utiles : `WaitForChildren` et `WaitCommand`. `WaitForChildren` bloque l'exécution jusqu'à ce que toutes les commandes démarrées jusqu'à présent soient terminées. Dans l'exemple, on attend donc que la base pilotable ait terminé d'avancer et que l'élévateur se soit levé. Ensuite, `WaitCommand` fait attendre le robot pendant 2 secondes avant de lancer le ballon. La commande se termine en faisant reculer le robot en même temps que de faire descendre l'élévateur.

9.6 Le processus de création d'une commande

Contrairement à ce qui a été présenté jusqu'à présent, il est rare que l'on sait exactement les méthodes dont un sous-système aura besoin avant de créer une commande.

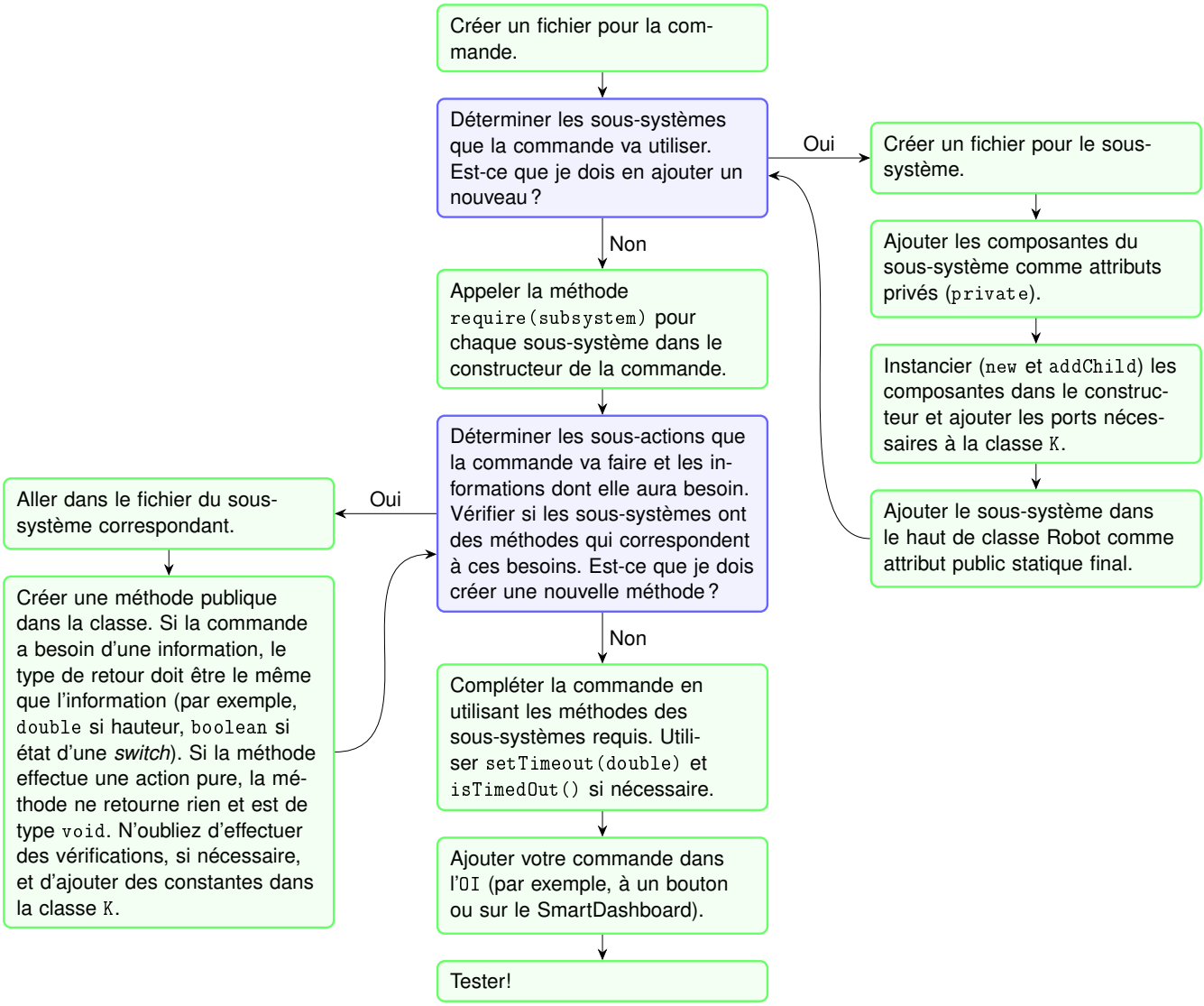


FIGURE 9.3 – Le processus de création d'une commande.

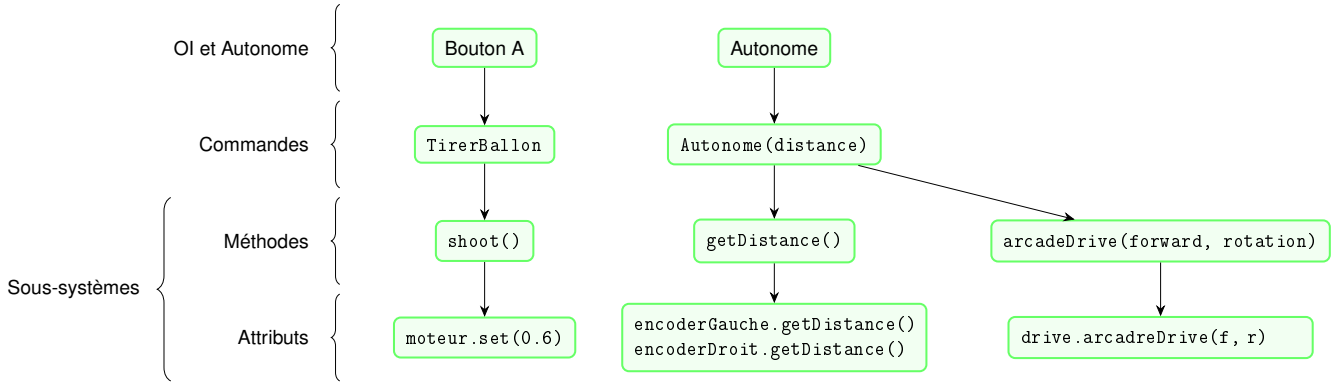


FIGURE 9.4 – La séquence d'appels dans un programme WPILib.

Liens utiles