

# **PISCINE CODING FACTORY**

Romuald GRIGNON

Coding Factory by ITESCIA  
CCI Paris Ile-de-France  
Cergy-Pontoise / Paris-Champerret

Année scolaire 2020-2021

# Contexte

- Pendant 1 semaine vous allez apprendre les bases de la programmation
- Une phase de prise de connaissances des notions :
  - Définition d'un algorithme
  - Compréhension des structures de stockage de données
  - Compréhension des différentes structures de contrôle d'exécution d'un programme
  - Logique de conception d'algorithmie
  - Implémentation d'algorithmes simples
- Une phase de mise en pratique ludique :
  - Programmation d'un algorithme de pilotage de vaisseau de course
  - Utilisation des notions abordées précédemment
  - Utilisation d'une interface existante (grâce à la documentation fournie)
- Environnement de développement [IntelliJ](#)
- Travail en petits groupes

# Contexte

- Pendant la première phase, nous étudierons les différents éléments d'un programme
- Pour chaque élément, nous débuterons par un **cours théorique** et des exemples en **live coding**, suivi d'une phase de mise en pratique avec des **exercices** types
- Le but de cette première phase est de connaître et savoir manipuler les différents éléments d'un programme informatique, ainsi que de commencer à acquérir une logique algorithmique (traduction d'un problème en successions d'instructions)
- Le but final de la deuxième phase sera de produire un **algorithme** qui pilotera un **vaisseau** dans une course. Chaque groupe s'organisera pour définir la stratégie qu'il entend adopter pour programmer son vaisseau
- Cette phase sera rythmée par des **compétitions** régulières où tous vos vaisseaux se mesureront les uns avec les autres et les différents classements permettront de comparer l'efficacité de vos algorithmes
- Vous serez **accompagnés** par plusieurs **formateurs** qui pourront vous aider sur les différents aspects (environnement de travail, algorithmie, théorie, implémentation)

# Planning de la semaine

- Le planning prévisionnel est le suivant. Une période de cours et exercices pendant les 2 premiers jours, suivi de 3 jours pendant lesquels vous pourrez faire évoluer le code de votre vaisseau
- A chaque fin de demi-journée, les codes de vos vaisseaux seront récupérés et des courses seront lancées afin d'établir les classements. Reportez vous à la diapositive « Evaluation des étudiants » pour plus d'informations.

LUNDI	MARDI	MERCREDI	JEUDI	VENDREDI
COURS + EXERCICES	COURS + EXERCICES	MISE EN PLACE DU PROJET		
		EVALUATION ?	EVALUATION	EVALUATION
COURS + EXERCICES	COURS + EXERCICES			
	PRESENTATION DU PROJET			
		EVALUATION	EVALUATION	FINALE

# **ENVIRONNEMENT DE TRAVAIL**

## **Logiciel : IntelliJ**

# Environnement de travail

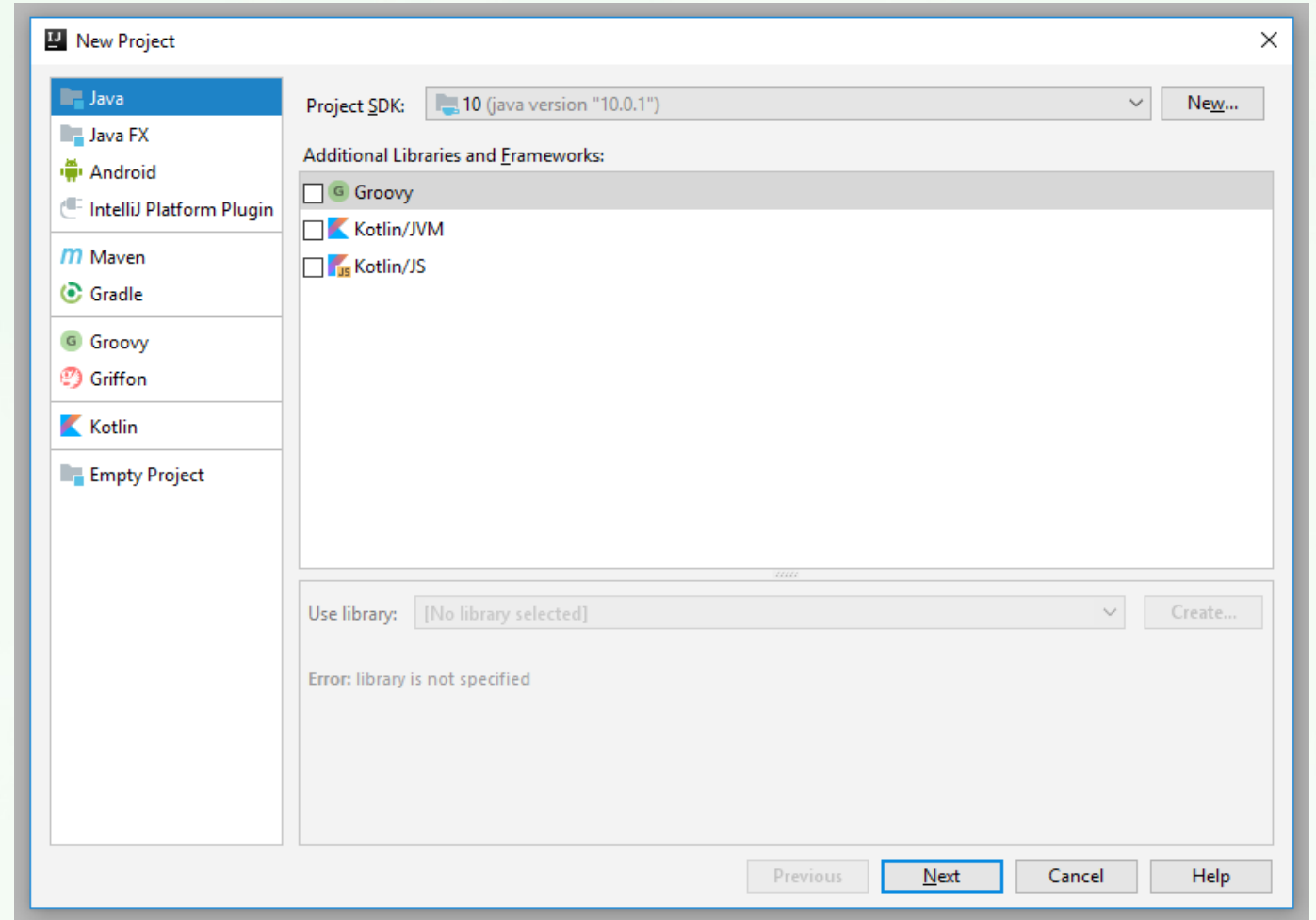
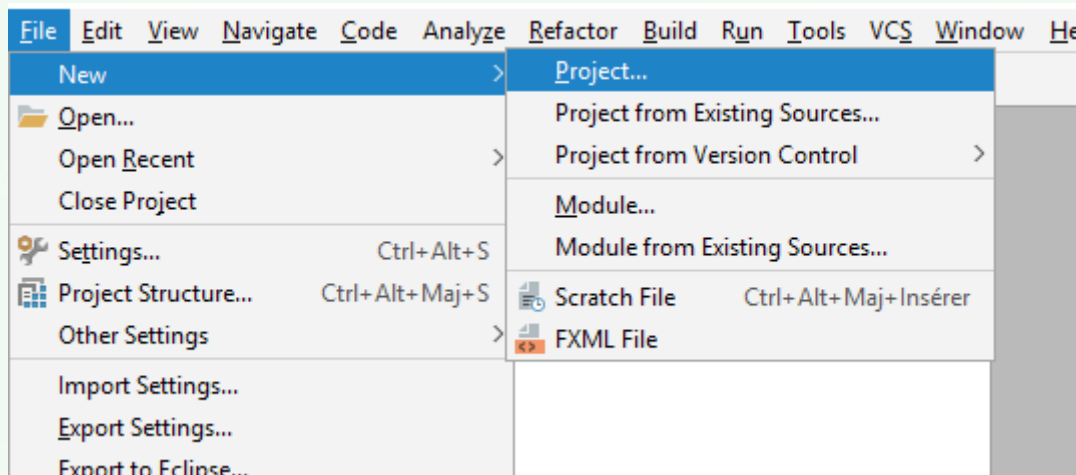
- Nous allons utiliser l'environnement de développement appelé **IntelliJ**.
- Cet environnement est multi-plateforme et gratuit. Même la licence professionnelle est accessible gratuitement au public de l'éducation (après inscription sur le site de l'éditeur JetBrains)
  - <http://www.jetbrains.com/>
- L'application que nous utiliserons pour la deuxième phase (course de vaisseaux) est codée en langage **JAVA**. Pour éviter de rajouter de la difficulté, nous **détournerons** l'utilisation de ce langage pour vous **simplifier** la prise en main et parvenir à une syntaxe plus proche du **langage C**
- La création du projet concernant les exercices de la première phase sera faite manuellement (voir détail des étapes dans les diapositives suivantes)
- Le projet de l'application de course de vaisseaux quant à lui est déjà disponible :
  - [https://github.com/Romuald78/codingdojo\\_studentplugin/](https://github.com/Romuald78/codingdojo_studentplugin/)

# **NOUVEAU PROJET IntelliJ**



# Nouveau projet IntelliJ

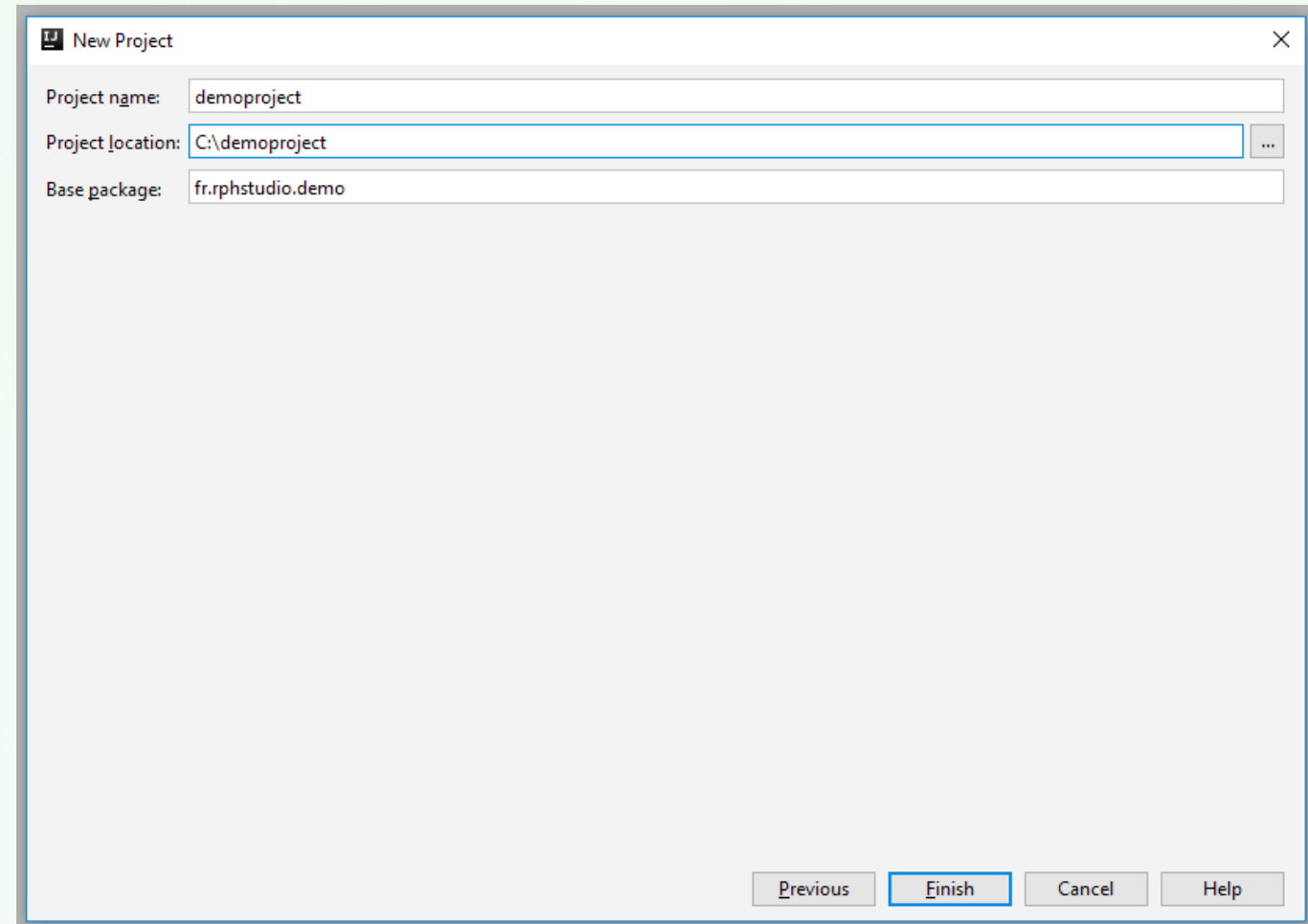
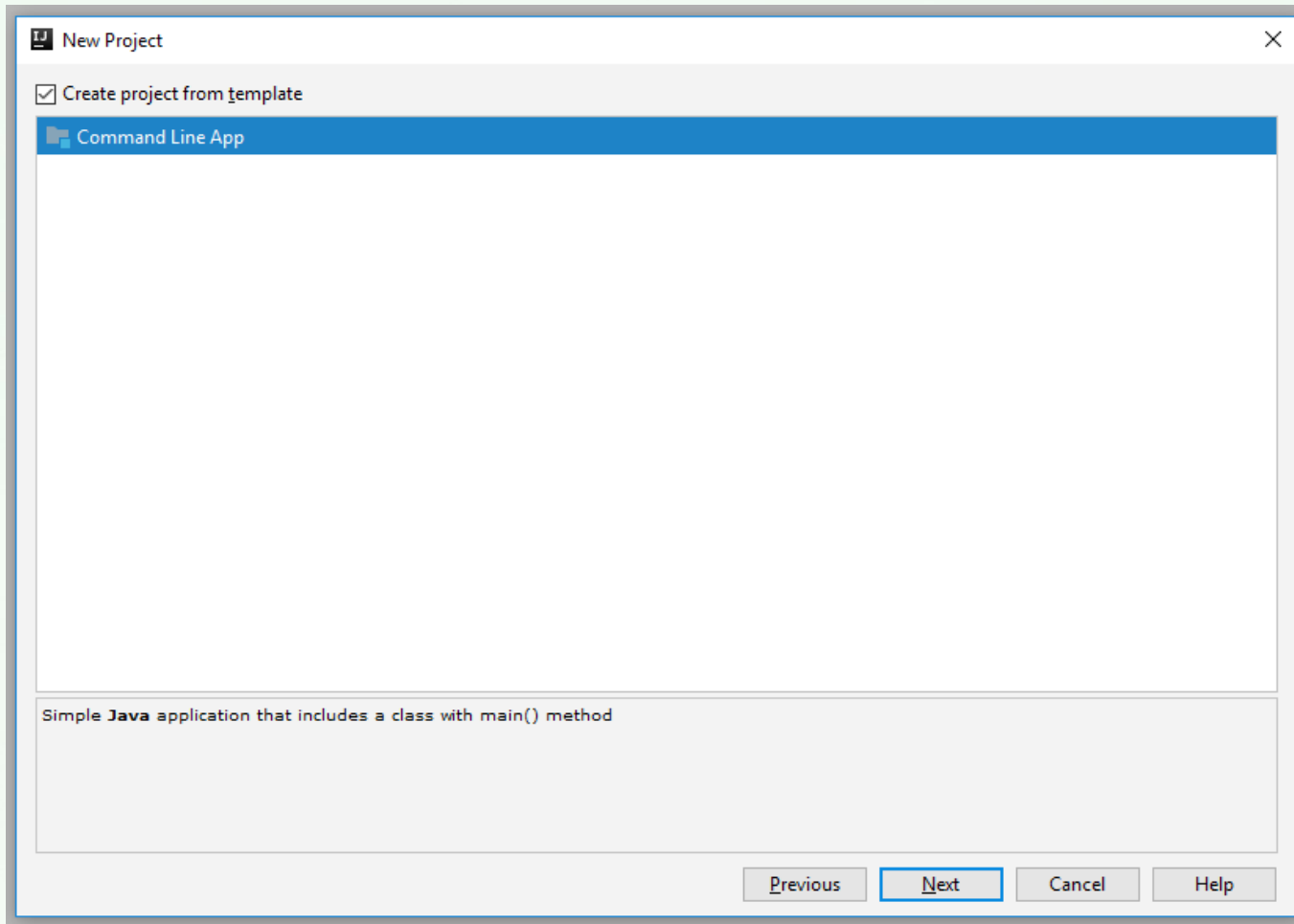
- Pour créer un nouveau projet d'application Java, choisissez '**File**' - '**New**' – '**Project**'
- Choisissez '**Java**' dans la liste de gauche puis cliquez sur '**next**'





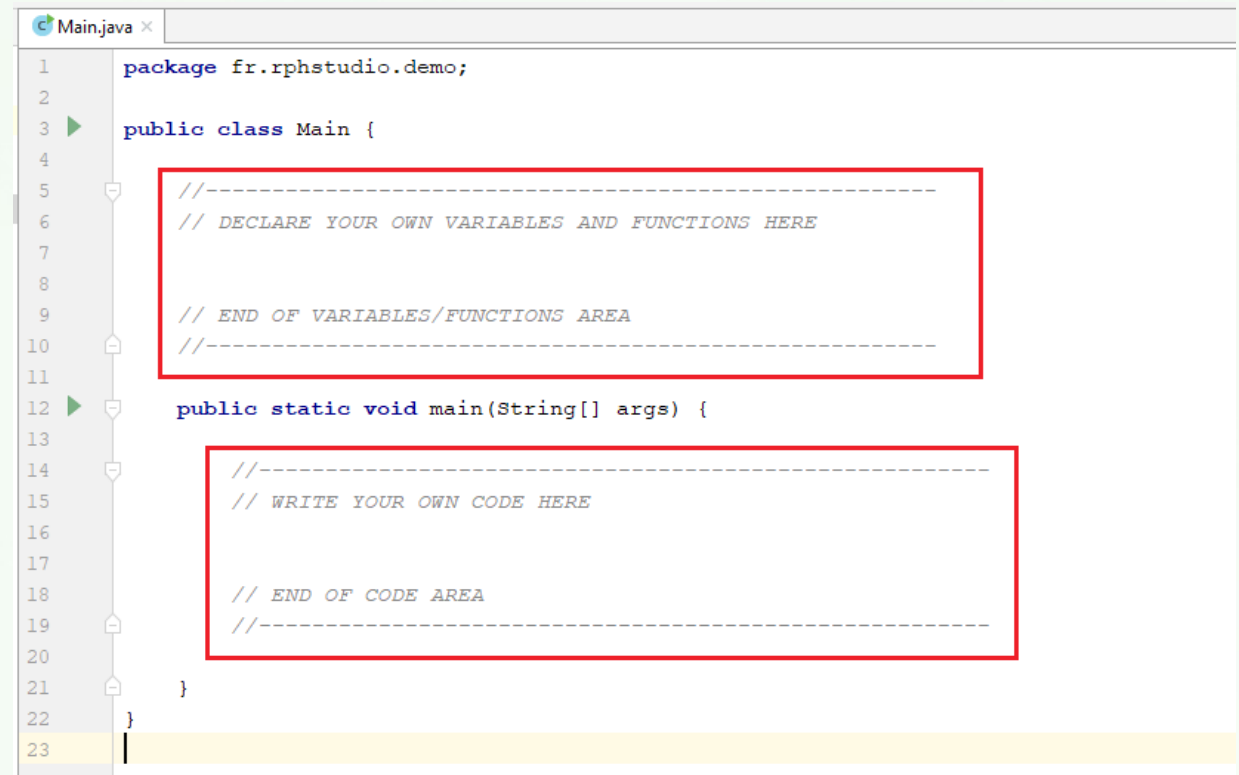
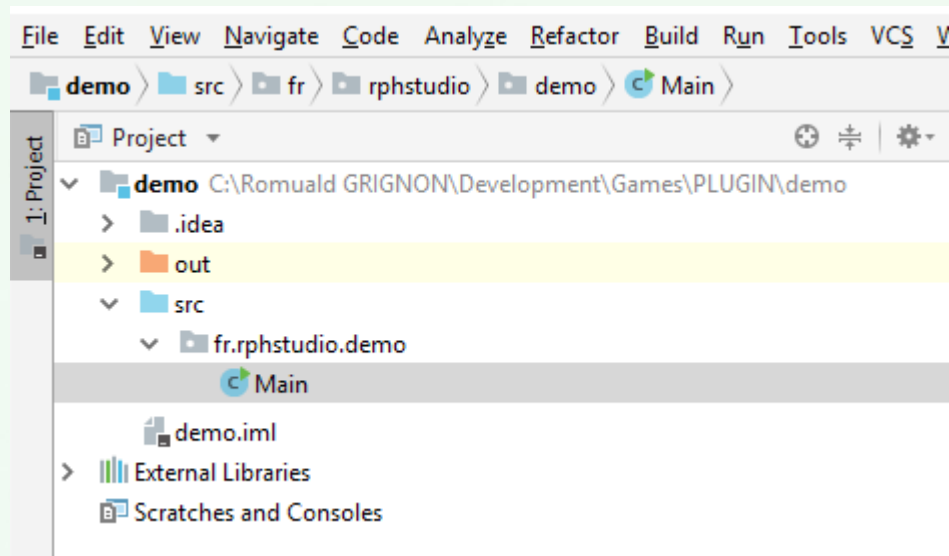
# Nouveau projet IntelliJ

- Dans la page suivante, sélectionnez le template '**Command Line App**'
- Sélectionner l'emplacement et le nom de votre projet



# Nouveau projet IntelliJ

- Votre projet est créé, et vous voyez apparaître un fichier '**Main**' dans le répertoire des sources : double-cliquez dessus pour l'ouvrir
- Vous pouvez alors le modifier comme indiqué sur l'image ci-dessous. Comme énoncé précédemment, nous allons employer une syntaxe proche du langage C. Il n'est pas question durant cette semaine de faire de la programmation orientée objet en Java.

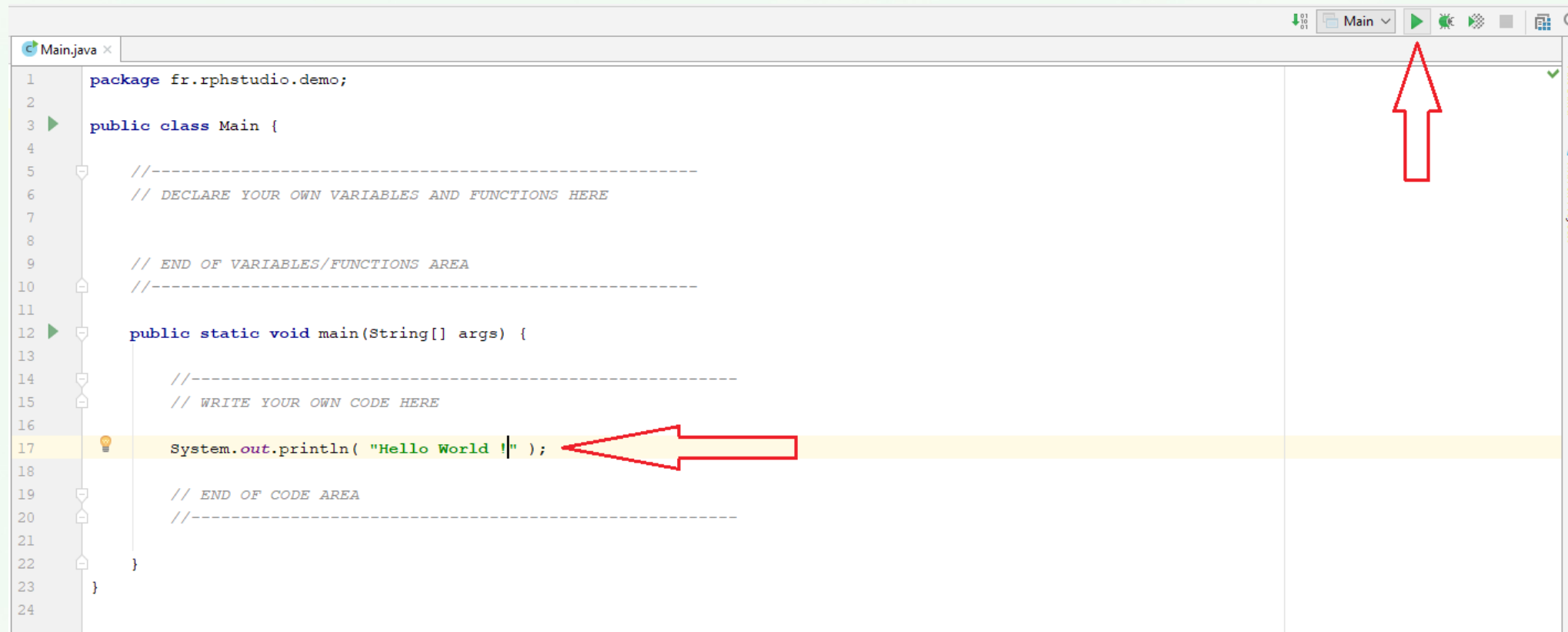


# Nouveau projet IntelliJ

- Dans votre fichier Main écrivez l'instruction suivante :

`System.out.println( "Hello World !" );`

- Exécutez le programme en appuyant sur le bouton '**Play**' : vous devriez voir apparaître le message dans la console. Votre environnement est prêt



# **PRINCIPES DE BASE ALGORITHMIE - PROGRAMMATION**

# Compilation / Interprétation

- Dans les langages informatiques, il existe 2 grandes familles :
  - Les langages dit « **compilés** ». Ces langages nécessitent une opération de compilation qui va se charger de transformer les fichiers de code en fichiers exécutables par un processeur. C'est le cas du langage **C**, **C++** et du **Java** (même si pour ce dernier, l'exécutable n'est pas directement utilisable par le processeur)
  - Les langages dit « **interprétés** ». Ces langages ne nécessitent pas d'opération après l'écriture du fichier de code, on peut directement les lancer, l'interpréteur se chargeant d'analyser les instructions au fur et à mesure qu'il les exécute (Python, JavaScript et PHP par exemple)
- Avec des langages compilés, beaucoup de problèmes peuvent être vus pendant l'étape de compilation, ce qui empêche d'exécuter un code non fonctionnel
- Avec des langages interprétés, majoritairement, les problèmes sont rencontrés lors de l'exécution du code
- Nous allons utiliser un langage compilé pendant cette semaine de Coding Dojo. Pour que la compilation fonctionne, il faut respecter une certaine grammaire dans l'écriture des lignes de code. Cette syntaxe sera vue au fur et à mesure de la prise de connaissances des différents éléments du code

# Algorithme

- Un algorithme est une **succession d'opérations** déterministes (claires, précises, sans ambiguïté, rejouables) dont le nombre est fini (l'exécution peut être infinie). Ces opérations sont effectuées une par une dans un ordre précis
- Il peut prendre en **entrée** des données et fournit obligatoirement une **sortie** (résultat, affichage, modifications de données, stockage, ...)
- Pour les mêmes données d'entrée et les mêmes instructions, l'algorithme fournira toujours le même résultat : c'est le côté **déterministe** de l'algorithme (on mettra de côté la notion d'aléa que l'on peut introduire dans certains algorithmes)
- On peut faire l'analogie entre un algorithme et une recette de cuisine ou une partition de musique.
  - Dans ces 2 exemples nous avons des données d'entrée (ingrédients avec quantités, notes de musiques avec valeurs et durées)
  - nous avons un procédé (étapes de la recette avec modification/mélange des ingrédients, durées de cuissons, exécution du jeu de chaque note de musique en fonction de sa valeur, durée)
  - nous obtenons un résultat (plat préparé, musique jouée). Si les données d'entrée et le procédé appliqué dessus sont les mêmes, le résultat sera toujours le même également



# Algorithme

- Un algorithme répond à une **problématique** donnée (traiter des données, résoudre un calcul complexe, ...), par conséquent, la suite d'instructions décrite dans l'algorithme est censée apporter la réponse au problème initial
- Un algorithme est divisé globalement en **3 parties** :
  - **La déclaration et l'initialisation** : on prépare les données d'entrée du problème. C'est ici que la déclaration et l'initialisation des variables seront faites. On prépare également toutes les variables qui serviront pour les calculs intermédiaires et pour stocker le résultat
  - **Le traitement** : c'est le cœur de l'algorithme. Il a pour rôle de transformer les données d'entrée pour répondre à la problématique. Il va donc manipuler les variables créées en conséquence
  - **Le résultat** : c'est ici que l'on récupère le résultat de l'algorithme et qu'on le met à disposition de l'entité qui a exécuté cet algorithme.
- Un algorithme n'est pas un programme, c'est une succession d'opérations qui peuvent être décrites avec des mots, des phrases : c'est ce que l'on appelle le **langage algorithmique**. En utilisant le langage algorithmique, on garanti que n'importe qui peut lire et comprendre l'algorithme, et son implémentation pourra se faire quel que soit le langage utilisé



# LES VARIABLES

# Variables : définition

- Une variable est une **zone de stockage**
- Sert à **mémoriser** des informations (valeurs numériques, chaînes de caractères, ...)
- Est définie par 3 éléments :
  - **Type de la variable** : permet d'indiquer quelle type d'information est contenue dans la variable.  
Nous utiliserons principalement des données de type
    - Valeur numérique entière
    - Valeur numérique décimale
    - Booléen
    - Chaîne de caractères
  - **Nom de la variable** : nom arbitraire donné à notre variable pour que nous puissions la réutiliser plus facilement. Un mauvais nommage de variables peut amener à la confusion dans la programmation d'un algorithme. Attention les noms de variables sont sensibles à la casse (majuscules/minuscules)
  - **Valeur** : le contenu de cette zone de stockage qu'est la variable est une 'valeur', unique et du type décrit précédemment

# Variables : déclaration / affectation

- Déclarer une variable c'est créer une zone en mémoire pour y stocker sa valeur. En général, cette zone mémoire est allouée dans la mémoire vive de du système (RAM)
- En langage algorithmique pour déclarer une variable on écrira :

Déclaration de 'age' de type entier

Déclaration de 'width' de type flottant (ou décimal)

Déclaration de 'name' de type chaîne de caractères

Déclaration de 'isCorrect' de type booléen

- Affecter une valeur à une variable c'est écrire cette valeur dans la case mémoire réservée à cette variable
- En langage algorithmique, pour affecter une valeur à une variable on écrira :

age ← 21

width ← 5,37

name ← « Romuald »

isCorrect ← VRAI

# Variables : déclaration / affectation

- Pour créer une variable, nous implémenterons avec la syntaxe suivante :

```
int age ;           // on crée une variable numérique entière qui s'appellera 'age'  
float width ;       // on crée une variable numérique décimale qui s'appellera 'length'  
String name ;       // on crée une variable 'chaine de caractères' qui s'appellera 'name'  
boolean isCorrect ; // on crée une variable booléenne qui s'appellera 'isCorrect'
```

- Pour affecter une valeur dans une variable, nous utiliserons l'opérateur '=' :

```
age = 21 ;           // met la valeur '21' dans la case mémoire appelée 'age'  
width = 5.37f ;      // met la valeur '5.37' dans la case mémoire appelée 'width'  
name = "Romuald" ;   // met la valeur "Romuald" dans la case mémoire appelée 'name'  
isCorrect = true ;   // met la valeur 'true' dans la case mémoire appelée 'isCorrect'
```

- Notez le point-virgule à la fin de chaque ligne de code. C'est une règle importante pour que le compilateur puisse savoir où commencent et se terminent les instructions
- Notez également l'emploi du // pour mettre en commentaires le code qui suit sur la ligne. Les commentaires ne sont pas compilés

# Variables : initialisation / affichage

- La première affectation de valeur dans une variable est appelée initialisation
- Il est possible de déclarer une variable et de l'initialiser sur la même ligne :

```
int age = 37 ;
```

```
float width = 9.587f ;
```

```
String name = "coding dojo" ;
```

```
boolean isCorrect = false ;
```

- Pour afficher une valeur de variable sur la console, nous utilisons la syntaxe suivante :

```
System.out.println( age ) ;      // affiche '37'
```

```
System.out.println( width ) ;    // affiche '9.587'
```

```
System.out.println( name ) ;     // affiche 'coding dojo'
```

```
System.out.println( isCorrect ) ; // affiche 'false'
```

# Variables : opérateurs mathématiques

- Avec les variables de type numérique, il est possible d'effectuer des opérations mathématiques en utilisant les opérateurs suivants :
  - Addition ( + )
  - Soustraction ( - )
  - Multiplication ( \* )
  - Division ( / )
  - Reste de division (modulo) ( % )

- Exemples :

```
age = 37 + 5 ;           // la variable 'age' prend la valeur 42
```

```
width = 100.0f - 57.4f ; // la variable 'width' prend la valeur 42.6
```

```
age = 10 * 5 ;           // la variable 'age' prend la valeur 50
```

```
width = 200.0f / 11 ;    // la variable 'width' prend la valeur 18.1818 car elle est de type flottante
```

```
age = 200 / 11 ;         // la variable 'age' prend la valeur 18 car elle est de type entier
```

```
age = 200 % 11 ;        // la variable 'age' prend la valeur 2
```

# Variables : opérateurs mathématiques

- Les opérandes utilisées dans les calculs peuvent également être des variables :
  - `float note1 = 10.5f ;`
  - `float note2 = 14.7f ;`
  - `float somme1 = note1 + note2 ;`
- Il est possible d'effectuer plusieurs opérations sur la même ligne de code :
  - `float note3 = 9.3f ;`
  - `float somme2 = note1 + note2 + note3 ;`
- Les parenthèses permettent de forcer l'ordre des opérations :
  - `float moyenne = (note1 + note2 + note3) / 3 ;`



# Variables : exercices

- Manipulez des variables
  - Déclarez une variable de chaque type (entier, décimal, chaîne de caractères, booléen)
  - Initialisez vos variables avec des valeurs arbitraires
  - Affichez les valeurs de vos variables pour vérifier que votre initialisation est correcte
  - Utilisez chacun des opérateurs arithmétiques vus précédemment (+, -, \*, /, %) pour modifier les valeurs des variables
  - Utilisez chacun des opérateurs arithmétiques pour combiner les valeurs de 2 variables et stockez le résultat dans une troisième variable
- Opérations décimales et entières
  - Initialisez une variable décimale et une autre entière avec une valeur impaire
  - Divisez ces deux valeurs par 2
  - Affichez le résultat et analysez-le
- Notez l'utilisation du caractère 'f' qui suit les valeurs décimales dans votre code

# **LES BRANCHEMENTS CONDITIONNELS**

# Branchements conditionnels

- Dans un algorithme, il arrive parfois où l'on souhaite traiter des données différemment en fonctions de conditions particulières
- Le but ici est de **vérifier** si une **condition** est vraie ou fausse, et en fonction du résultat de cette comparaison, effectuer un traitement ou un autre
- En langage algorithmique, on a le schéma suivant :

SI <condition>

ALORS

<traitement A>

SINON

<traitement B>

FIN SI

- La condition est une valeur booléenne, qui vaut donc **VRAI** ou **FAUX**. Ici on exécutera le traitement A si la condition est vraie, ou le traitement B si la condition est fausse
- Le bloc «SINON» n'est pas obligatoire dans le cas où le traitement B est vide

# Opérateurs de comparaison

- Pour réaliser un branchement conditionnel, il nous faut une condition, qui très souvent est le résultat d'une comparaison entre variables
- Pour comparer les valeurs de 2 variables, il faut en général qu'elles soient du même type. Il reste toutefois possible, suivant les langages de comparer des types différents
- Les opérateurs de comparaison qui existent sont les suivants :

A est égal à B ( == )

A est supérieure strictement à B ( > )

A est inférieur strictement à B ( < )

A est supérieure ou égal à B ( >= )

A est inférieur ou égal à B ( <= )

A est différent de B ( != )

- Tous les opérateurs renvoient un résultat de type booléen : on récupère donc une valeur qui vaut VRAI si la condition est remplie, FAUX sinon

# Branchements conditionnels

- On souhaite multiplier la valeur d'un entier par 10 quand il est supérieur ou égal à 5, sinon on lui ajoutera 1. On affichera la nouvelle valeur de cet entier :
- En langage algorithmique on obtiendra :

déclaration d'une variable A de type entier

$A \leftarrow$  valeur arbitraire

SI A est supérieur ou égal à 5

ALORS

$A \leftarrow A * 10$

SINON

$A \leftarrow A + 1$

FIN SI

Afficher A

# Branchements conditionnels

- L'implémentation se fait grâce aux instructions if... else... :

// Initialisation de la variable A

int A;

A = 12 ;

// Traitement

if( A >= 5 ) {

    A = A \* 10 ;

}

else {

    A = A + 1 ;

}

// Affichage du résultat

System.out.println( A ) ;

- Notez l'utilisation des accolades pour encadrer les blocs de traitement

# Branchements conditionnels

- On souhaite afficher la mention d'un bachelier qui vient de recevoir sa note. On va tester successivement la note obtenue :

Déclaration variable N de type décimal

N ← valeur arbitraire

SI N inférieur strictement à 10 ALORS

Affiche « rattrapage »

SINON SI N inférieur strictement à 12 ALORS

Affiche «pas de mention »

SINON SI N inférieur strictement à 14 ALORS

Affiche « assez bien »

SINON SI N inférieur strictement à 16 ALORS

Affiche « bien »

SINON SI N inférieur strictement à 18 ALORS

Affiche « très bien »

SINON SI N inférieur ou égal à 20 ALORS

Affiche « excellent »

SINON

Affiche « Tricheur »



# Opérateurs logiques

- Il est possible de grouper plusieurs conditions en une seule grâce aux opérateurs logiques que sont le ET logique et le OU logique
- Un ET logique de deux opérandes booléennes renvoie lui aussi un booléen en fonction du tableau de correspondance suivant :
  - A est faux, B est faux  $\rightarrow$  A ET B est faux
  - A est vrai, B est faux  $\rightarrow$  A ET B est faux
  - A est faux, B est vrai  $\rightarrow$  A ET B est faux
  - A est vrai, B est vrai  $\rightarrow$  A ET B est vrai
- Un OU logique de deux opérandes booléennes renvoie lui aussi un booléen en fonction du tableau de correspondance suivant :
  - A est faux, B est faux  $\rightarrow$  A OU B est faux
  - A est vrai, B est faux  $\rightarrow$  A OU B est vrai
  - A est faux, B est vrai  $\rightarrow$  A OU B est vrai
  - A est vrai, B est vrai  $\rightarrow$  A OU B est vrai

# Opérateurs logiques

- L'implémentation des opérateurs logiques ET et OU :

```
boolean A = false ;
```

```
boolean B = true ;
```

```
// teste si A et B sont vraies toutes les deux
```

```
if( A && B ) {
```

```
...
```

```
// teste si au moins une des 2 variables A,B est vraie
```

```
If( A || B ) {
```

```
...
```

- Comme les opérateurs logiques renvoient un booléen, on peut stocker ce résultat dans une variable de type booléen également :

```
boolean C = A && B ;
```

```
if( C ) {
```

```
...
```

# Branchements conditionnels : exercices

- Initialisez 2 variables entières et déterminez quelle est la valeur la plus petite des 2
  - Refaites l'exercice avec 3 variables
- Initialisez 2 variables entières et déterminez le signe du produit sans faire le calcul
- Initialisez une variable entière comme l'âge d'une personne et affichez un message si cette personne est majeure ou mineure
- Faîtes l'exercice qui affiche la mention d'un bachelier en fonction de sa note
- Initialisez une variable et affichez un message si la valeur est paire ou impaire
  - En utilisant l'opérateur modulo (%) et en utilisant les propriétés de division d'une variable entière
- Déterminez si la valeur d'une année est bissextile ou non
  - Il faut que l'année soit divisible par 4, sauf une année qui est divisible par 100 et pas par 400
- Déterminez si une date est correcte
  - 3 variables entières qui représentent les jour, mois et année d'une date
  - Utilisez le code précédent pour déterminer si au mois de février il y a 28 ou 29 jours

# LES BOUCLES

# Boucles

- Dans un algorithme, pour des raisons d'optimisation d'écriture, on peut vouloir effectuer plusieurs fois le même bloc de traitement. On utilisera pour cela une structure de boucle :
- Il existe globalement 2 types de structures de boucles :
  - La boucle POUR qui est utilisée quand on souhaite faire varier une variable sur une plage de valeurs finie et connue (le nombre d'itérations est alors connu).

La boucle POUR nécessite une variable qui contiendra une valeur qui changera à chaque itération, une valeur de départ, une valeur de fin, une valeur d'incrément
  - La boucle TANT QUE qui est utilisée quand on souhaite boucler tant qu'une condition est maintenue. Ici Le nombre d'itérations n'est pas forcément connu  
La boucle TANT QUE ne nécessite qu'une condition (valeur booléenne)

# Boucle POUR

- On souhaite effectuer la multiplication de deux variables entières en n'utilisant que des instructions de type addition :

Déclaration de 4 variables A, B, C et I de type entier

$A \leftarrow$  valeur arbitraire

$B \leftarrow$  valeur arbitraire

$C \leftarrow 0$

POUR I variant de 1 à B inclus par pas de 1

FAIRE

$C \leftarrow C + A$

FIN POUR

Afficher C

# Boucle POUR

- L'implémentation du code correspondant :

```
int A = 6 ;  
int B = 12 ;  
int C = 0 ;  
int I;  
for( I=1 ; I<=B ; I=I+1 ) {  
    C = C + A ;  
}  
System.out.println( C ) ;
```

- Notez le format de l'instruction for ( <initialisation> ; <condition> ; <itération> )
  - Le bloc d'instructions <initialisation> est exécuté avant de rentrer dans la boucle la première fois
  - Le bloc d'instructions <condition> retourne un booléen qui est testé à chaque itération pour savoir si on continue de boucler ou si l'on sort de la boucle. Si la condition est vraie, on exécute le bloc d'instructions entre les accolades, sinon on saute tout ce bout de code et on continue l'exécution
  - Le bloc d'instructions <itération> est exécuté après le bloc d'instructions dans la boucle à la fin de chaque itération



# Boucle TANT QUE

- On souhaite effectuer le calcul du factoriel d'un nombre entier:

Déclaration de 2 variables N, R de type entier

$N \leftarrow$  valeur arbitraire

$R \leftarrow N$

TANT QUE  $N > 1$

FAIRE

$N \leftarrow N - 1$

$R \leftarrow R * N$

FIN TANT QUE

Afficher R

# Boucle TANT QUE

- L'implémentation du code correspondant :

```
int N = 12 ;  
int R = N ;  
while ( N > 1 ) {  
    N = N - 1 ;  
    R = R * N ;  
}  
System.out.println( R ) ;
```

- Notez le format de l'instruction while ( <condition> )
  - On va exécuter le bloc de code entre accolades si la condition est vraie.
  - Une fois le bloc de code terminé d'exécuter, on teste à nouveau la condition, si elle est toujours vraie, on exécute à nouveau le bloc de code
  - On continue ainsi tant que la condition est vraie
- Attention à la conception de votre algorithme et prévoyez bien vos **conditions de sortie** de boucle, sinon votre programme **bouclera à l'infini**

# Les boucles : exercices

- Affichez tous les nombres entre 0 et 100 inclus, par ordre croissant
- Affichez tous les nombres entre 100 et 0 inclus, par ordre décroissant
- Affichez tous les nombres multiples de 3 entre 0 et 100 par ordre croissant
  - En utilisant une valeur d'incrément spécifique pour votre boucle
  - En utilisant une valeur d'incrément de '1' et en utilisant un branchement et l'opérateur modulo
- Calculez  $a^b$  (a puissance b) en utilisant des multiplications successives
- Affichez la table de multiplication de la valeur d'une variable
- Calculez la somme des entiers de 1 à N, et stoppez votre programme si cette somme dépasse la valeur M
- Affichez le miroir d'un nombre (ex : pour une valeur de 1597, affichez 7951)
- Testez une suite de Syracuse qui démarre par N :
  - 1 élément de la suite dépend du précédent :  $[U_{n+1} = U_n / 2 \text{ si } U_n \text{ pair}] \text{ et } [U_{n+1} = 3 * U_n + 1 \text{ si } U_n \text{ impair}]$
  - Bonus : trouvez la valeur maximale de la suite

# LES FONCTIONS

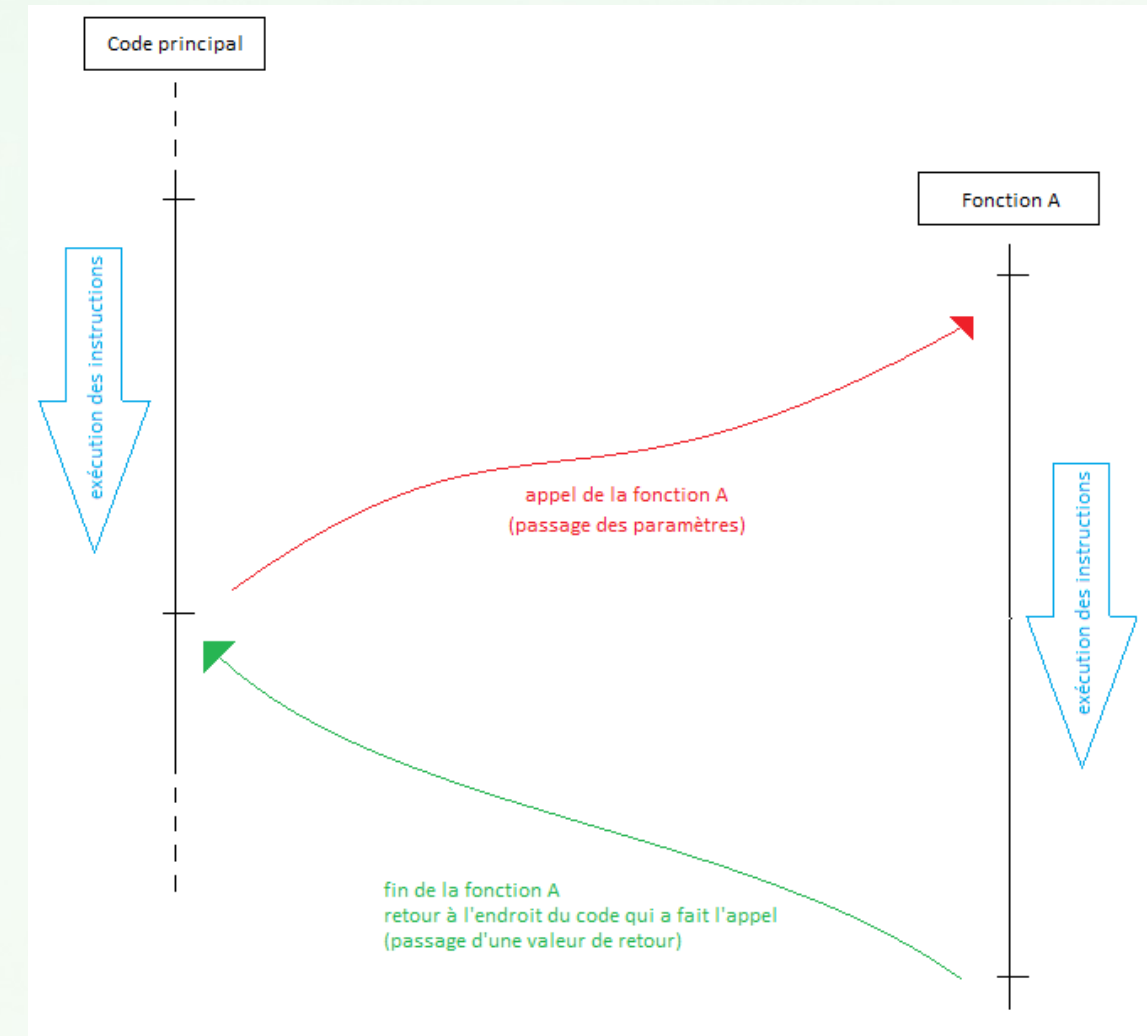
# Fonctions

- Une fonction est un **ensemble d'instructions** regroupées au sein d'un seul et même bloc de code
- Une fonction sert à exécuter un traitement particulier (en fonction des besoins de l'application) plusieurs fois sans avoir à recopier le code : on dit qu'on « **appelle** » la fonction pour qu'elle exécute le code voulu
- Une fonction sert donc à **factoriser** du code. En programmation, on tend à ne jamais avoir deux blocs de code qui font la même chose à deux endroits différents dans l'application : on essaye de factoriser ce code dans une fonction et on appellera la fonction aux différents endroits nécessaires
- Une fonction est déterminée par 3 éléments :
  - Son **nom**, arbitraire au même titre que les noms des variables (sensible à la casse)
  - Ses **paramètres d'entrée**, optionnels, forment une liste de variables à utiliser par la fonction
  - Sa **valeur de retour**, optionnelle également, est une valeur renvoyée à la fonction appelante

# Fonctions

- Schéma chronologique d'un appel de fonction :

- Le code principal appelle la fonction A  
(il passe 0, 1 ou plusieurs valeurs en paramètres)
- Le code principal suspend son exécution
- La fonction A démarre son exécution  
(elle utilise éventuellement les paramètres reçus)
- La fonction A termine son exécution  
(elle renvoie 0 ou 1 valeur de retour)
- Le code principal reçoit la valeur de retour
- Le code principal reprend son exécution



- Un appel de fonction est donc **bloquant**

- Le code « appelant » reste en attente de la fin de l'exécution du code « appelé »

# Fonctions

- En langage algorithmique comme en implémentation, on utilisera le schéma suivant :  
`<typeValeurDeRetour> <nomDeFonction> ( [<type1 nom1>[,<type2 nom2>[...]]] )`
- Ce schéma reprend les 3 blocs décrits précédemment
- Si il y a plusieurs paramètres d'entrée on utilisera la virgule pour les séparer
- Chaque paramètre sera décrit par son type (un paramètre est une variable) et par son nom arbitraire
- Pour la valeur de retour on indique seulement le type de données
- Exemples d'implémentation :

`int multiply (int a, int b) { ... } // prend deux entiers en entrée et retourne un entier`

`float power (float x, float y) { ... } // prend deux réels en entrée et retourne un réel`

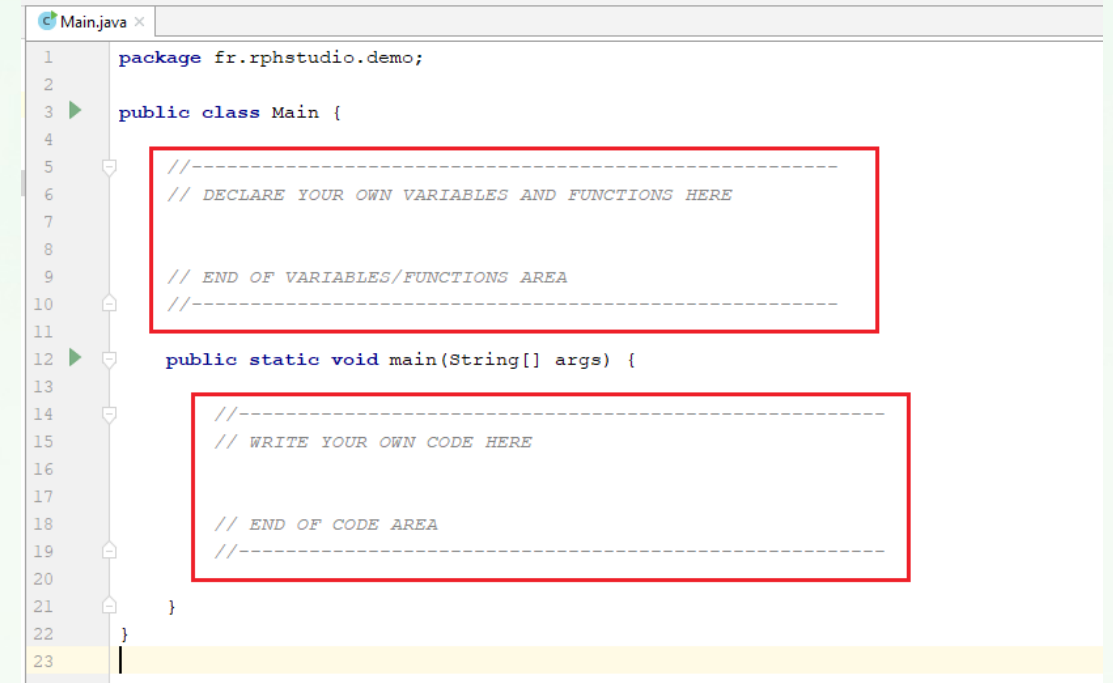
`float random () { ... } // ne prend pas de paramètre et retourne un réel`

`void displayInfo () { ... } // ne prend pas de paramètre et ne retourne pas de valeur`



# Fonctions

- Vous mettrez le code de vos fonctions dans l'emplacement encadré en rouge en haut :



```
1 package fr.rphstudio.demo;
2
3 public class Main {
4
5     //-----
6     // DECLARE YOUR OWN VARIABLES AND FUNCTIONS HERE
7
8     // END OF VARIABLES/FUNCTIONS AREA
9     //-----
10
11
12     public static void main(String[] args) {
13
14         //-----
15         // WRITE YOUR OWN CODE HERE
16
17         // END OF CODE AREA
18         //-----
19
20     }
21 }
22
23
```

- Pour des raisons techniques dû à l'environnement que nous détournons pour les besoins du cours, il faudra rajouter le mot-clé « static » devant la déclaration de votre fonction, mais ne vous formalisez pas sur ce point
- Faîtes appel aux formateurs afin de régler ce léger point de détail

# Fonctions

- Contenu d'une fonction :

```
int multiply ( int a, int b ) {  
    // on déclare deux variables locales supplémentaires, c et i,  
    int c = 0 ;  
    int i ;  
    // on boucle pour réaliser le calcul  
    for ( i=1 ; i<=b ; i=i+1 ) {  
        c = c + a ;  
    }  
    // on stoppe l'exécution de la fonction et on renvoie une valeur au code « appelant »  
    return c ;  
}
```

# Fonctions

- Appel de la fonction multiply depuis le code principal

...

```
int A = 4 ;           // déclare une variable entière A et on affecte la valeur 4
int B = 7 ;           // déclare une variable entière B et on affecte la valeur 7
int resultat = 0;      // déclare une variable entière resultat et on affecte la valeur 0
System.out.println( resultat ) ; // affiche la valeur de la variable resultat : ici 0
resultat = multiply( 6, 12 ) ; // affecte la valeur de retour de multiply à la variable resultat
System.out.println( resultat ) ; // affiche la valeur de la variable resultat : ici 72
resultat = multiply( A, B ) ; // affecte la valeur de retour de multiply à la variable resultat
System.out.println( resultat ) ; // affiche la valeur de la variable resultat : ici 28
```

...

# Fonctions

- Autre fonction

```
int maximum( int a, int b ) {  
    // on déclare une variable locale c pour stocker la plus grande valeur entre a et b  
    // pour le moment on y stocke la valeur de a  
    int c = a ;  
    // on teste si la valeur de b est strictement supérieure à celle de a : alors on stocke b dans c  
    if( b > a ) {  
        c = b ;  
    }  
    // on retourne la valeur de c (qui contient donc la plus grande valeur entre a et b)  
    return c ;  
}
```

# Fonctions

- Appel de la fonction maximum depuis le code principal

...

```
int A = 8 ;           // déclare une variable entière A et on affecte la valeur 8
int B = 12 ;          // déclare une variable entière B et on affecte la valeur 12
int resultat = 0;     // déclare une variable entière resultat et on affecte la valeur 0
System.out.println( resultat ) ; // affiche la valeur de la variable resultat : ici 0
resultat = maximum( 21, 3 ) ; // affecte la valeur de retour de maximum à la variable resultat
System.out.println( resultat ) ; // affiche la valeur de la variable resultat : ici 21
resultat = maximum( A, B ) ; // affecte la valeur de retour de maximum à la variable resultat
System.out.println( resultat ) ; // affiche la valeur de la variable resultat : ici 12
```

...

# Les fonctions : exercices

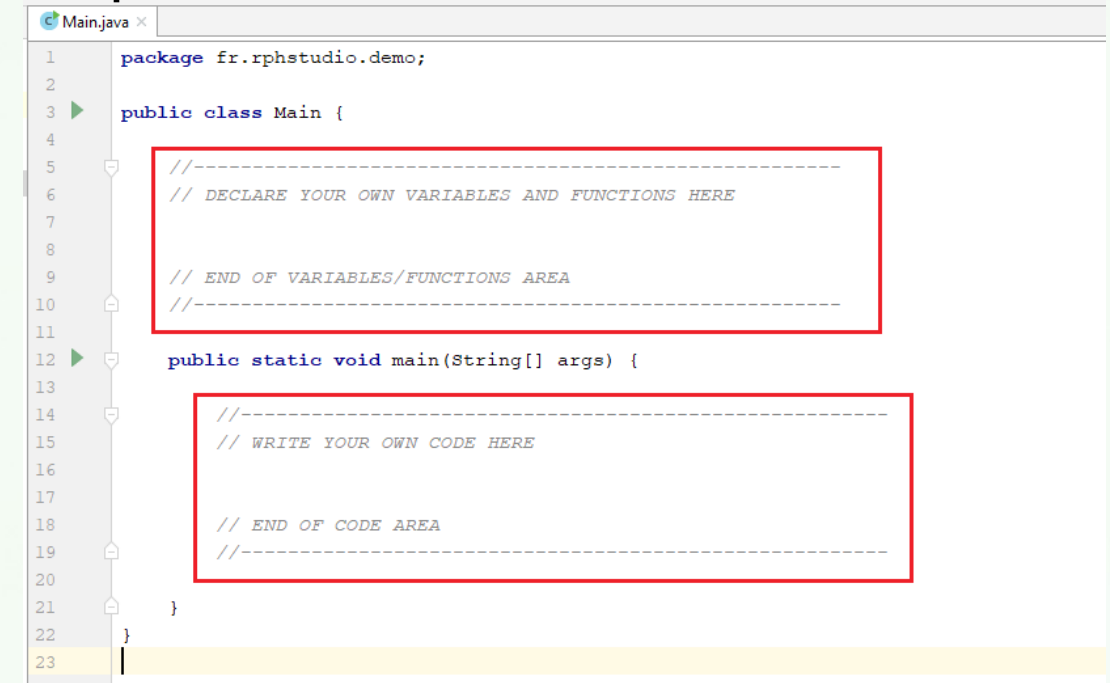
- Fonction « min » : 2 entiers en paramètres, 1 entier en sortie
- Fonction « abs » : 1 entier en paramètre, 1 entier en sortie
- Fonction « isCorrectDate » : 3 entiers en paramètres, 1 booléen en sortie
- Fonction « isLeapYear » : 1 entier en paramètre, 1 booléen en sortie
- Fonction « sum » : 1 entier N en paramètre, 1 entier en sortie (somme des N entiers)
- Fonction « power » : 2 entiers en paramètres, 1 entier en sortie
- Fonction « displayMultTable » : 1 entier en paramètre, affichage de la table
- Fonction « integerMirror » : 1 entier en paramètre, 1 entier en sortie (miroir d'un entier)

# LES VARIABLES GLOBALES



# Variables globales

- Toutes nos variables ont une durée de vie limitée à la fonction dans laquelle elle sont déclarées. Il est toutefois possible de créer des variables qui persistent pendant toute la durée de votre programme et **accessibles** depuis **n'importe quelle fonction** : ce sont des variables « **globales** »
- Vous les déclarerez dans la zone encadrée rouge du haut
- Nommez les différemment de vos variables locales pour éviter toute confusion



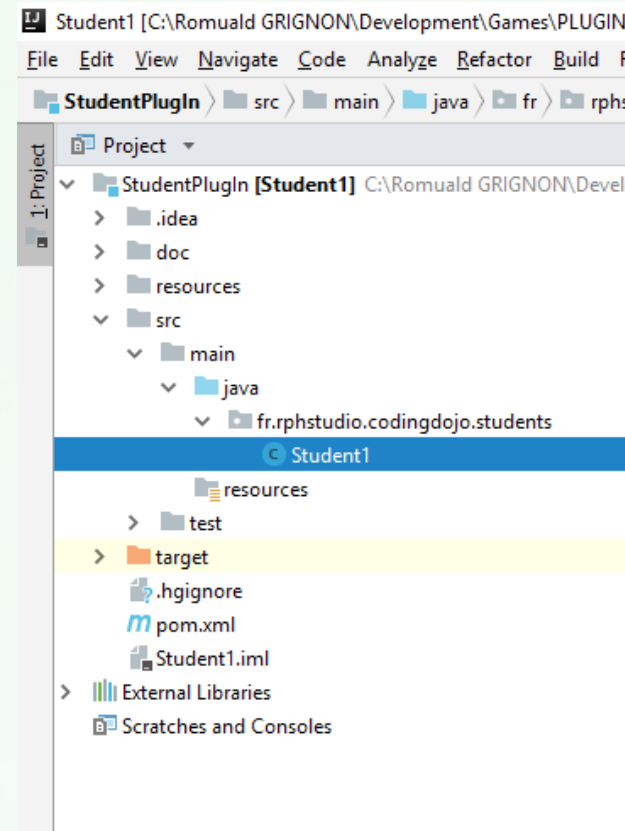
```
1 package fr.rphstudio.demo;
2
3 public class Main {
4
5     //-----
6     // DECLARE YOUR OWN VARIABLES AND FUNCTIONS HERE
7
8
9     // END OF VARIABLES/FUNCTIONS AREA
10    //-----
11
12    public static void main(String[] args) {
13
14        //-----
15        // WRITE YOUR OWN CODE HERE
16
17
18        // END OF CODE AREA
19        //-----
20
21    }
22 }
23
```

- Utilisez le mot-clé static devant votre déclaration tout comme pour les fonctions

# **COURSE DE VAISSEAUX : préparation**

# Préparation de l'environnement

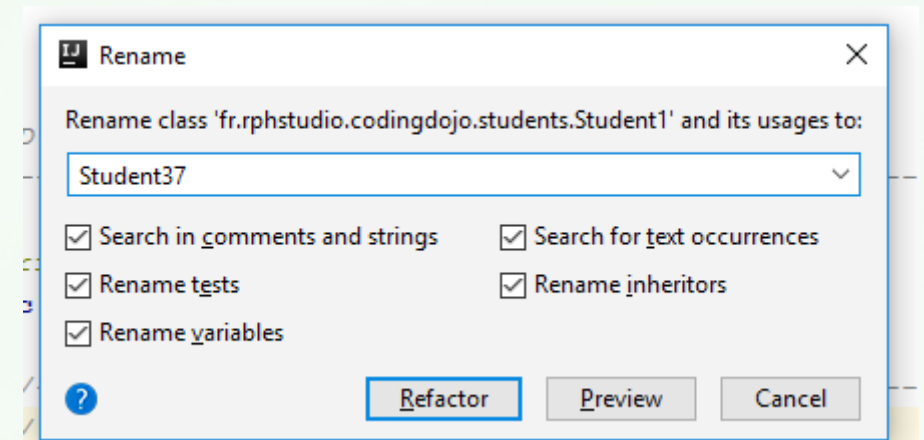
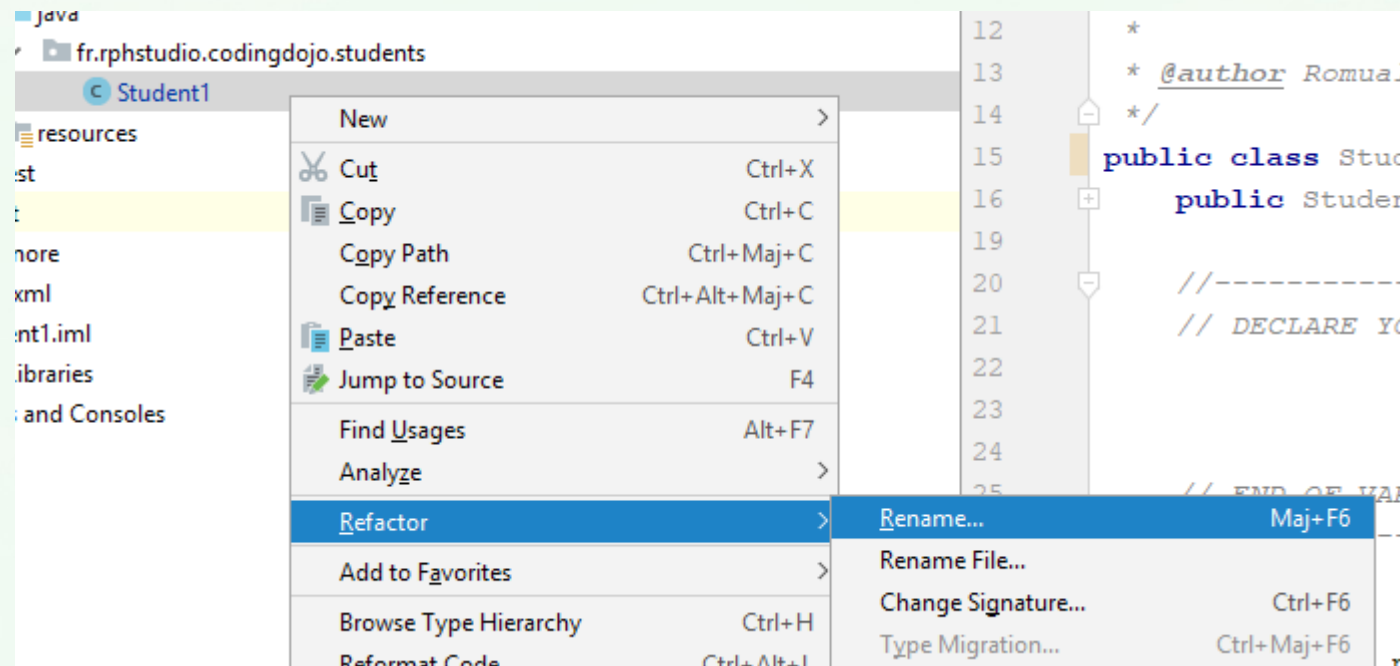
- Une fois le code du projet récupéré à :
  - [https://github.com/Romuald78/codingdojo\\_studentplugin/](https://github.com/Romuald78/codingdojo_studentplugin/)
- Ouvrez le projet sous IntelliJ. Pour cela, ouvrez le fichier pom.xml à la racine du dossier, et indiquez que vous voulez l'ouvrir en tant que « projet »
- Dépliez le répertoire des sources jusqu'à trouver le fichier '**Student1**'
- Double-cliquez dessus pour l'ouvrir
- Vous devriez voir apparaître le code d'origine qui est fourni



```
15 public class Student1 extends PodPlugIn {
16     public Student1(Pod p) { super(p); }
17
18     //-----
19     // DECLARE YOUR OWN VARIABLES AND FUNCTIONS HERE
20
21
22
23
24
25     // END OF VARIABLES/FUNCTIONS AREA
26     //-----
27
28     @Override
29     public void process(int delta)
30     {
31         //-----
32         // WRITE YOUR OWN CODE HERE
33
34         setPlayerName("Student 1");
35         selectShip( shapeNumber: 1);
36         setPlayerColor( r: 255, g: 255, b: 255, a: 255);
37
38         moveToNextCheckPoint( speed: 0.5f);
39
40         // END OF CODE AREA
41         //-----
42     }
```

# Préparation de l'environnement

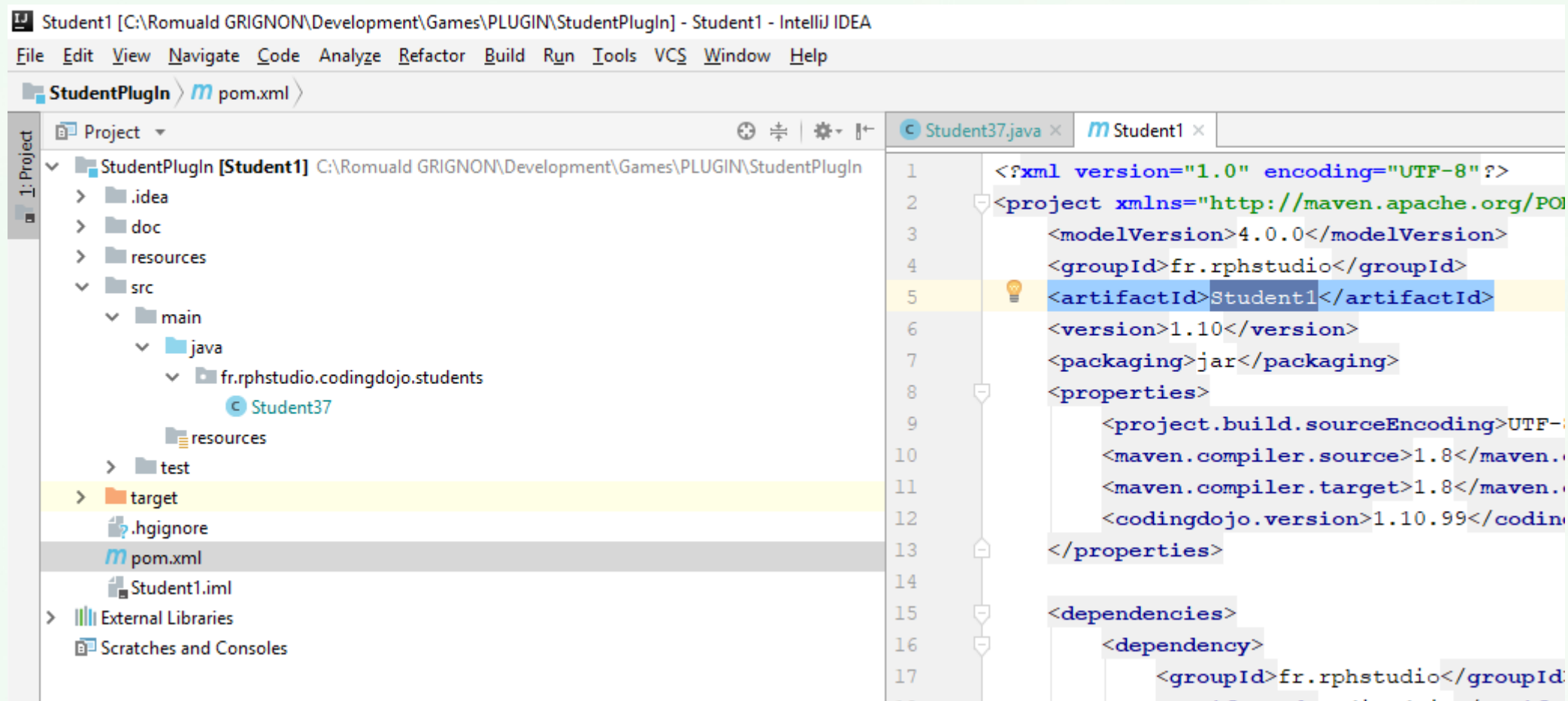
- Ici le projet est configuré pour le groupe '1'. Vous allez devoir modifier ce fichier afin de le renommer avec votre numéro de groupe. Par exemple, si vous portez le numéro '37', vous allez devoir renommer le fichier en '**Student37**'
- Pour effectuer cette opération, faites un clic-droit sur le fichier '**Student1**' et choisissez '**refactor**' → '**rename**'. Entrez le nom voulu et validez



- Les numéros 78, 87 et 89 sont des numéros réservés. Vous pouvez choisir un numéro de groupe entre 1 et 100

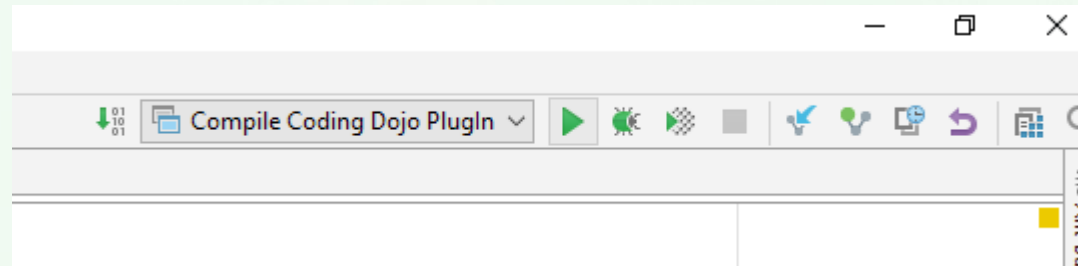
# Préparation de l'environnement

- Visualisez le fichier '**pom.xml**', et modifiez la ligne qui contient '**Student1**' par le numéro de votre groupe ( ici dans l'exemple remplacez par '**Student37**' )



# Préparation de l'environnement

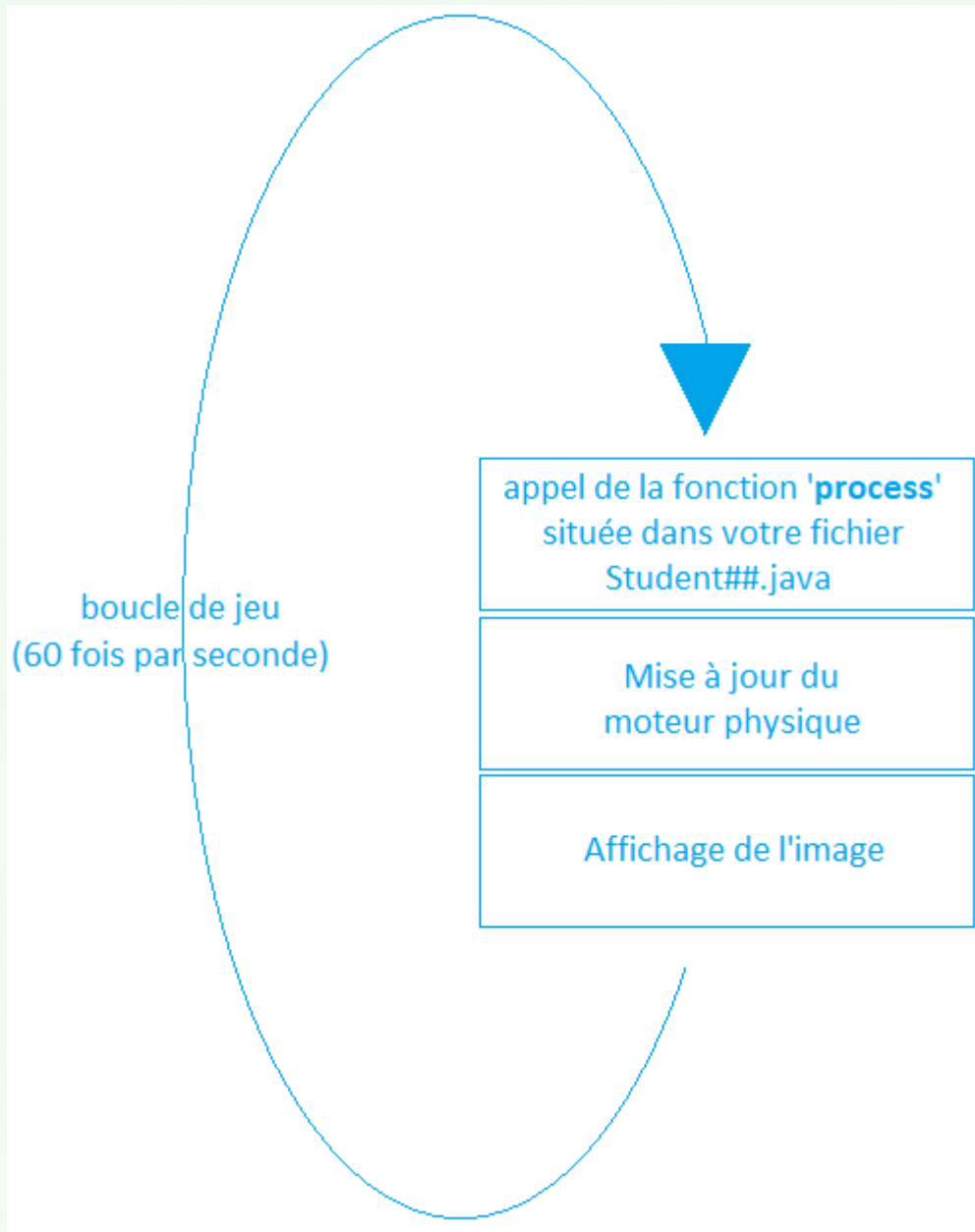
- Maintenant nous allons tester l'exécution du projet en cliquant sur le bouton '**play**'



- Cette opération permet de compiler votre code et de créer un fichier nommé '**Student##.jar**' dans le sous-répertoire '**/resources/students**' de votre dossier de projet (avec ## le numéro de votre groupe tel que configuré précédemment)
- Normalement l'application devrait se lancer automatiquement après la compilation : si vous voyez les vaisseaux se déplacer c'est que votre environnement est prêt : il ne reste plus qu'à coder
- Ici, le mot-clé static qui était demandé durant les exercices n'est plus nécessaire. Pour vos fonctions ou variables globales, utilisez simplement la syntaxe du cours
- Vous verrez qu'il existe un autre fichier déjà généré, **Teachers-2.1.jar**, qui contient le code des vaisseaux des formateurs : un challenge de plus à relever ^^



# Fonctionnement du code



- Votre code est appelé 1 fois par image (environ 60 fois par seconde)
- Votre code pourra récupérer l'état de votre vaisseau (position, vitesse, ...) et décider d'effectuer des actions en retour (accélérer ou freiner, tourner, ...)
- Ces actions auront un impact sur la physique de votre vaisseau
- Une fois votre fonction terminée de s'exécuter, le moteur physique sera mis à jour, et l'image à l'écran sera renouvelée
- Entre 2 appels de votre fonction process, les variables disparaissent car elles sont locales à la fonction, comme vu précédemment
- Pour mémoriser des informations d'un appel à l'autre, il vous faudra utiliser des variables globales
- Vu que votre code est appelé séquentiellement dans la boucle de jeu, il est impératif d'éviter des boucles infinies qui auraient comme conséquence de geler l'application
- Rassurez-vous nous avons mis en place un mécanisme permettant de détecter les traitements trop longs et de ne pas geler l'application en « tuant » le traitement incriminé



# **COURSE DE VAISSEAUX : description**

# Visuel de l'application

- L'application est découpée en 3 grandes parties :
  - Les commandes possibles en haut à gauche
  - Les statuts de chaque vaisseau et le classement sur la droite
  - La zone de course au centre



# Commandes de l'application

- Changer de **piste** avec les flèches GAUCHE et DROITE
- Changer de **niveau de code** avec les flèches HAUT et BAS (voir règles spéciales)
- Changer le **déroulement du temps** avec les touches + et – (ou P et M)
  - Il est possible de modifier le temps avec les rapports de **[1:8]** à **[x256]** par puissance de 2
  - Si vous réduisez le temps en dessous de [1:8] le jeu passe en pause
- Activer les **collisions** entre vaisseaux avec la touche C (par défaut les collisions ne sont pas activées et les vaisseaux se chevauchent en mode 'ghost race')
- Activer le **mode 'batterie'** avec la touche B (voir règles spéciales)
- Activer le mode '**nombre de tours infini**' avec la touche I
- Faire défiler le **classement** avec les touches PAGE UP et PAGE DOWN
- Basculer en **mode fenêtré** avec la touche F11
- Afficher le nombre d'**images par seconde** avec la touche F
- **Quitter** l'application avec la touche ESCAPE

# Statut des vaisseaux

- Des indicateurs visuels existent pour préciser l'état de chaque vaisseau
- Les informations suivantes sont disponibles à tout instant :
  - Nom du vaisseau fourni par l'algorithme (également affiché au dessus du vaisseau lui-même)
  - Nombre de tours terminés
  - Barre de progression du tour en cours
  - Barre de niveau de batterie (reste à 100% quand le mode batterie est désactivé)
  - Barre de chargement du boost (voir règles spéciales)
  - Meilleur temps de tour
  - Temps total de course
  - Classement du vaisseau ou état spécifique (ex : cause de disqualification)

# Statut des vaisseaux

- Un symbole décrit l'état du vaisseau pendant la course :
  - [ - ] le vaisseau est en pleine course : état par défaut
  - [ # ] un nombre indiquant le classement final du vaisseau (dans le cas où le vaisseau vient à manquer d'énergie, son classement peut déjà être fourni avant la fin de la course)
- Le symbole peut indiquer que le vaisseau a été disqualifié :
  - [ ? ] le vaisseau est sorti trop loin de l'écran et s'est perdu dans l'espace
  - [ ! ] le vaisseau a utilisé un appel de fonction non autorisé par le niveau de code actuel, ou bien le vaisseau a appelé plusieurs fois des fonctions pour accélérer ou plusieurs fois des fonctions pour tourner (dans la même frame) entraînant un problème au niveau système : le vaisseau ne fonctionne alors plus jusqu'à la fin de la course
  - [ T ] le vaisseau a mis trop de temps pour rejoindre le prochain checkpoint
  - [ ~ ] l'algorithme du vaisseau a généré une boucle infinie : ce vaisseau sera disqualifié pour cette course ainsi que toutes les suivantes !

# **COURSE DE VAISSEAUX : règles spéciales**



# Niveau de code

- Pour coder l'algorithme de déplacement de votre vaisseau, vous avez à votre disposition plusieurs **fonctions** qui vous permettront de réaliser diverses opérations
- Ces fonctions sont fournies pour que vous puissiez avoir un code fonctionnel dès le départ, et il vous faudra recoder ces fonctions vous même étape par étape
- Dans la **documentation** fournie, ces fonctions ont un **niveau de code** maximal
- Si le niveau de code de la course est supérieur strictement à celui de la fonction que vous utilisez, alors cette fonction ne s'exécutera pas et votre vaisseau sera **disqualifié**
- Pour pouvoir faire fonctionner votre vaisseau avec un niveau de code plus élevé, vous devrez coder vous-même la fonctionnalité fournie par la fonction que vous appelez
- Une description plus détaillée des étapes de niveaux de code est disponible dans les diapositives qui suivent



# Mode Batterie

- Lorsque l'on active le mode '**batterie**' à l'aide de la touche B, les vaisseaux démarrent une course pendant laquelle les accélérations leur font **consommer de l'énergie**.
- La **barre d'énergie** est visible sur l'interface graphique, et la valeur restante de batterie est accessible en appelant la fonction **getShipBatteryLevel** qui retourne un pourcentage d'énergie restante (valeur décimale entre 0.0 et 100.0)
- Lorsque l'énergie arrive à 0, le vaisseau ne peut plus bouger et la course s'arrête pour lui. Il conservera son classement si les autres vaisseaux ne le dépassent pas entre temps, sinon il perdra les places en conséquence
- Il est possible de **recupérer de l'énergie** de 2 manières :
  - **En freinant** (appel à **accelerateOrBrake** avec paramètre négatif) on récupère un peu d'énergie proportionnellement à la vitesse actuelle du vaisseau et la puissance de freinage. Dans tous les cas, cela ne compensera pas la perte d'énergie due aux accélérations
  - En passant au-dessus d'un **checkpoint** qui porte le **symbole électrique**, on recharge sa batterie : il convient donc de faire stationner ou ralentir suffisamment son vaisseau au dessus d'un checkpoint de recharge le temps que la batterie soit remplie, avant de reprendre le cours de la course
- Le nombre de tours par course est de 10 en mode normal, et de 50 en mode batterie

# Boost

- Il est possible d'effectuer une action de **boost** qui va **temporairement augmenter** la **vitesse** de votre vaisseau
- Une **jauge de boost** est disponible sur l'interface graphique et son niveau est accessible via l'appel à la fonction **getShipBoostLevel** qui retourne un pourcentage de chargement du boost (valeur décimale entre 0.0 et 100.0)
- Pour activer ce boost il faut appeler la fonction **useBoost**. Cet appel va vider la jauge à 0%. Il faudra donc attendre la **fin de son rechargement** pour pouvoir le réutiliser
- Attention, si la jauge de chargement de boost n'est pas remplie totalement à 100%, l'utilisation du boost réduira la vitesse du vaisseau temporairement au lieu de l'augmenter : pensez-donc bien à effectuer un **test de la jauge** avant de l'utiliser
- L'utilisation du boost ne **consomme pas d'énergie** quand le mode batterie est activé
- L'utilisation du boost doit être **synchronisée avec le placement** dans le circuit : une activation juste avant un virage ne ferait qu'envoyer votre vaisseau loin du chemin optimal et vous ferait perdre un temps précieux

# **COURSE DE VAISSEaux**

## **Documentation du code**

# Documentation du code

- Lorsque vous avez récupéré le projet, un sous-répertoire appelé « [doc](#) » contient un fichier HTML ([index.html](#)) qui réunit l'ensemble des [fonctions](#) que vous pouvez utiliser pour votre algorithme de pilotage de vaisseau
- Cette documentation est divisée en plusieurs [catégories](#), notamment :
  - [Display] : esthétique du vaisseau (couleur, forme, nom)
  - [Actions] : commandes principales du vaisseau (accélérer/ralentir, tourner, activer le boost)
  - [Status] : état physique de votre vaisseau (position, angle, vitesse, niveaux de batterie et boost)
  - [Global] : progression dans la course (nombre de tours, progression dans le tour courant, numéro du prochain checkpoint à valider)
  - [CheckPoints] : informations sur les checkpoints (positions, recharge électrique)
  - [Trigonometry] : toutes les fonctions nécessaires aux calculs d'angles et de distances
- N'hésitez pas à faire appel aux formateurs quant à l'utilisation de cette documentation

# **COURSE DE VAISSEaux**

## **Evaluation**

# Evaluation des étudiants

- Malgré l'aspect ludique de l'application, le travail effectué sur la programmation de l'algorithme de votre vaisseau sera **évalué**, à la fois sur l'avancement en terme de niveau de code, mais aussi sur le résultat **fonctionnel**
- Au moins 1 fois par demi-journée, plusieurs courses avec des **mesures de performances** seront lancées par les formateurs. A l'issue de ces courses, seront récupérés les **classements et niveaux de code** de chaque vaisseau
- Une **moyenne** des ces résultats sera effectuée en fin de semaine et sera la « **note d'entraînement** »
- La dernière mesure officielle (« **note finale** ») quant à elle sera isolée des autres
- Il y aura donc 2 résultats par vaisseau (entraînement et finale) : le résultat global sera la moyenne des 2 classements et le niveau maximal atteint
- Le classement des résultats se fera d'abord par niveau de code décroissant, puis, pour les ex-æquo, par classement de vos vaisseaux dans la course
- L'évaluation se fait avec le mode batterie activé



# **COURSE DE VAISSEaux**

## **Niveaux de code**



# Niveau de code #01

- C'est le niveau de départ. Il ne contient qu'une seule fonction, qui est celle fournie dans votre code : **moveAndRecharge**
- Cette fonction va faire en sorte de tourner votre vaisseau vers le prochain checkpoint et le faire accélérer dans cette direction.
- Si le niveau de batterie est trop faible, il se dirigera vers le premier checkpoint avec la fonction de recharge, et ne reprendra le cours de la course qu'une fois sa batterie revenue à un niveau acceptable

# Niveau de code #02

- Au niveau 2, la fonction **moveAndRecharge** ne fonctionne plus, il faut donc la remplacer
- Le but ici est d'utiliser la fonction **updateChargingMode** qui fournit en retour une valeur booléenne sur le besoin ou non de charger son vaisseau en énergie
- En fonction du besoin de recharge, il faudra effectuer un **branchement conditionnel**
  - Soit aller vers le prochain checkpoint de la course (**moveToNextCheckPoint**)
  - Soit aller vers le premier checkpoint de recharge (**moveToFirstChargingCheckPoint**)

# Niveau de code #03

- Au niveau 3, les deux fonctions de déplacements précédentes ne sont plus disponibles
- Il faut donc maintenant séparer en 2 la tâche qu'elles remplissaient : c'est à dire se tourner vers la cible, et accélérer
- Pour cela vous pourrez utiliser les fonction **turnTowardNextCheckPoint** et **turnTowardFirstChargingCheckPoint** pour vous tourner vers la cible que vous désirez (respectivement le prochain checkpoint ou un checkpoint de recharge)
- Il vous restera à utiliser la fonction **accelerateOrBrake** pour accélérer en direction de votre cible
- Remarquez que maintenant vous pouvez accélérer à des niveaux différents en fonction de votre cible

# Niveau de code #04

- Au niveau 4, les 2 fonctions de rotation précédentes ne sont plus disponibles
- En remplacement nous allons utiliser la fonction **turnTowardPosition** qui nécessite de connaître les coordonnées X et Y de votre cible
- Vous utiliserez donc les fonctions **getNextCheckPointX**, **getNextCheckPointY** pour récupérer les coordonnées du prochain checkpoint
- Vous utiliserez **getFirstChargingCheckPointX**, **getFirstChargingCheckPointY** pour récupérer les coordonnées du premier checkpoint de recharge
- Ces coordonnées seront passées en paramètres à la première fonction présentée ici

# Niveau de code #05

- Les fonctions de récupération de coordonnées précédentes ne sont plus disponibles au niveau 5
- Nous allons utiliser pour cela la fonction générique **getNextCheckpointIndex** pour récupérer le numéro du prochain checkpoint à rallier
- De la même manière en utilisant **getFirstChargingCheckpointIndex** nous pouvons récupérer le numéro du premier checkpoint de recharge
- Ce numéro peut être passé en paramètre des fonctions **getCheckpointX** et **getCheckpointY** pour récupérer les coordonnées de n'importe quel checkpoint à partir de son numéro
- Comme pour le niveau précédent, à partir des coordonnées, vous pouvez vous tourner vers votre cible

# Niveau de code #06

- Ici c'est la fonction **updateChargingMode** qui n'est plus disponible au niveau 6
- Nous n'avons donc plus moyen automatique de savoir si nous devons recharger le vaisseau ou non : c'est donc cette tâche qu'il s'agit de résoudre ici
- Vous utiliserez le concept de **variable globale** pour mémoriser le fait que vous êtes en train de recharger ou non
- Vous mettrez à jour cette variable en comparant le niveau de batterie restante grâce à la fonction **getShipBatteryLevel**



# Niveau de code #07

- Ici c'est la fonction qui recherche l'index du premier checkpoint de recharge qui n'est plus disponible
- Il va falloir trouver ce checkpoint par vos propres moyens en utilisant le concept de **boucle**
- En parcourant tous les numéros de checkpoint possibles (le nombre maximum de checkpoints est fourni par la fonction **getNbRaceCheckPoints**), il faudra tester si chaque checkpoint possède la possibilité de recharger grâce à la fonction **isCheckpointCharging**
- Si vous trouvez un checkpoint qui possède cette capacité, il faudra mémoriser son numéro
- Le reste est commun au code du niveau précédent, à partir du numéro de checkpoint, vous pouvez retrouver sa position, et orienter votre vaisseau dans cette direction
- A ce stade votre vaisseau recherche tout seul son point de recharge quand il en a besoin



# Niveau de code #08

- Ici c'est la fonction **turnTowardPosition** qui n'est plus disponible
- Il va falloir utiliser la fonction **turnToAngle** qui permet de s'orienter vers un angle absolu donné
- Il existe une fonction **getAbsoluteAngleFromPositions** qui permet de calculer l'angle nécessaire pour orienter son vaisseau en fonction d'une position de destination
- Il faudra passer la position du vaisseau lui meme, récupérable grâce aux fonctions **getShipPositionX** et **getShipPositionY**
- Il faudra faire ce travail à la fois pour le prochain checkpoint quand le vaisseau a suffisamment d'énergie, et pour le premier checkpoint de recharge quand il a besoin de refaire le plein
- A ce stade, vous orientez votre vaisseau en indiquant une direction absolue (un angle)

# Niveau de code #09

- A ce stade, c'est la fonction de rotation **turnToAngle** qui n'est plus disponible
- Il va falloir indiquer simplement à votre vaisseau de tourner de manière relative (vers la gauche ou vers la droite) et de combien de degrés
- Pour vous aider, il existe une fonction **getRelativeAngleDifference** qui calcule la différence entre 2 angles
- Grâce à l'orientation de votre vaisseau (angle de départ) et la direction vers votre cible (calculée dans votre code du précédent niveau) vous allez pouvoir calculer l'angle relatif à appliquer à votre vaisseau
- Il ne vous reste plus qu'à utiliser la fonction **turn** pour orienter votre vaisseau
- A ce stade, vous pilotez votre vaisseau en utilisant seulement 2 fonctions d'actions : **accelerateOrBrake** et **turn**.
- En d'autres termes, vous avez maintenant 2 commandes, l'une permet d'accroître ou de réduire la vitesse (accelerateOrBrake), et l'autre permet de diriger vers la gauche ou la droite votre vaisseau (turn)

# Niveau de code #10

- A partir du niveau 10 et au delà, seuls des aspects trigonométriques sont impliqués dans la progression du niveau de code
- Si l'aspect mathématiques vous rebute, vous pouvez rester au niveau 9 et vous concentrer sur les « fonctionnalités » permettant à votre vaisseau d'être plus efficace dans ses courses
- A ce stade, c'est le calcul des distances entre 2 points qui n'est plus disponible
- La fonction **getDistanceBetweenPositions** doit être recodée
- C'est un simple calcul du théorème de Pythagore qui doit être fait ici en utilisant la fonction racine carrée **sqrt**
- A ce stade, toutes les distances de votre algorithme sont calculées à l'aide de votre propre code

# Niveau de code #11

- A ce niveau, la fonction **getAbsoluteAngleFromPositions** n'est plus disponible
- Vous allez donc devoir calculer vous même les angles absolus nécessaires à votre algorithme
- Pour cela vous utiliserez la fonction tangente inverse (**atan2**) pour réaliser cette fonctionnalité
- A ce stade, tous les angles absolus de votre algorithme sont calculés par votre propre code

# Niveau de code #12

- A ce niveau, la fonction **getRelativeAngleDifference** n'est plus disponible
- Vous allez donc devoir calculer vous même les différences d'angles présentes dans votre algorithme
- A ce stade, tous les angles relatifs de votre algorithme sont calculés par votre propre code
- Votre code n'utilise plus aucune fonction spécifiquement développée pour vous aider. Vous êtes autonome : il ne vous reste qu'à vous focaliser sur les « fonctionnalités » qui permettront de rendre votre algorithme plus performant et donc à votre vaisseau de gravir les places du classement

# **COURSE DE VAISSEaux**

## **Fonctionnalités avancées**

# Fonctionnalités

- Maintenant que vous avez atteint un niveau de code suffisant (niveau 9+), votre algorithme ne contient que des fonctions que vous avez codées. Il vous reste à améliorer cet algorithme afin que votre vaisseau puisse être plus performant
- Il y a plusieurs axes de travail qui ont chacun leur impact, et vous pouvez travailler sur ceux que vous voulez, dans l'ordre que vous voulez :
  - #A – Optimisation de trajectoire
  - #B – Economie d'énergie
  - #C – Utilisation du 'Boost'

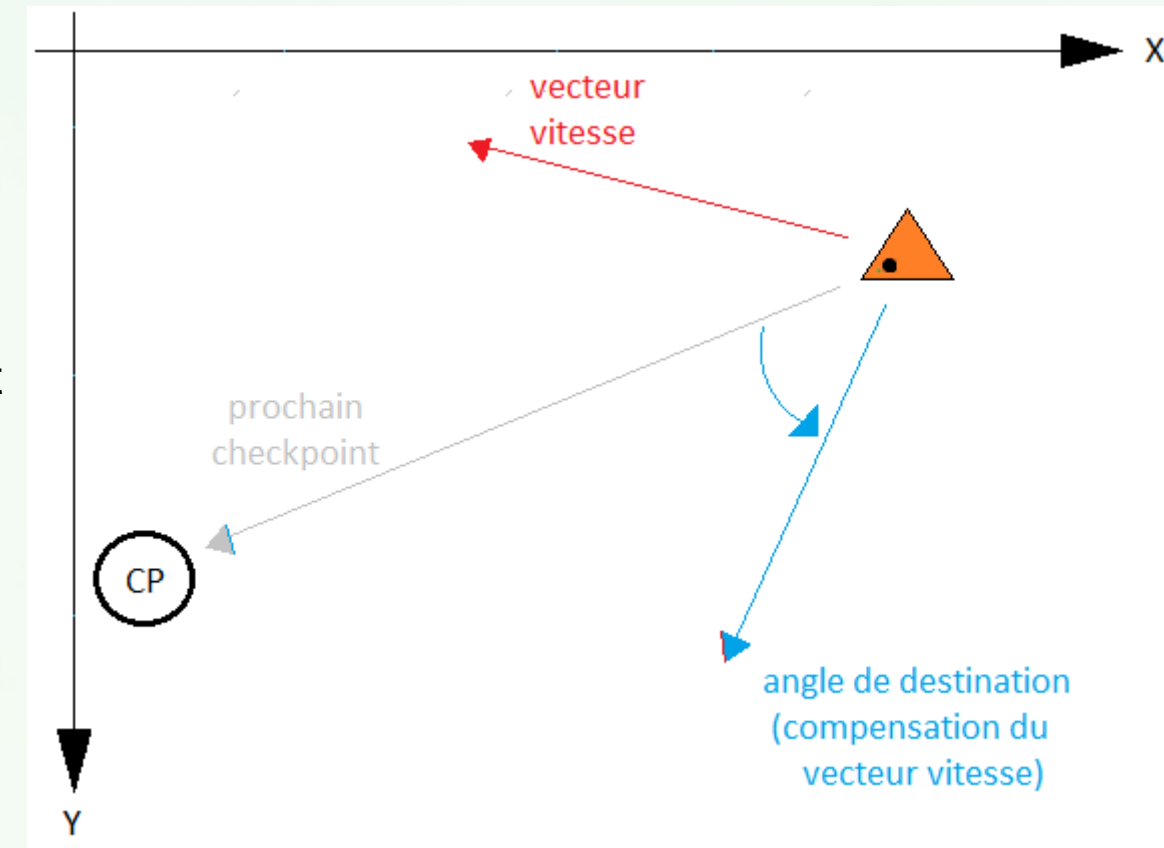


# Optimisation de trajectoire

- Freinage en arrivant sur le checkpoint :
  - Il peut être intéressant de freiner en arrivant sur le prochain checkpoint afin de ne pas trop 'glisser' et ainsi être dans une position plus intéressante d'un point de vue de la trajectoire optimale
  - Pour cela vous pouvez vérifier la distance qui vous sépare du prochain checkpoint et freiner quand vous passez en dessous d'un certain seuil
  - Il sera peut être nécessaire de vérifier l'angle d'arrivée sur le checkpoint, la distance n'étant pas forcément le seul élément utile à gérer
  - Avec 3 checkpoints alignés, freiner en arrivant sur celui du milieu sera contre productif
- Drift :
  - En arrivant à proximité du prochain checkpoint, ET en étant bien aligné dans sa direction, vous pouvez commencer à orienter votre vaisseau vers le second checkpoint, a minima en arrêtant l'accélération, voire en freinant, et ainsi voir votre vaisseau partir en glissade sur le côté afin d'être bien aligné pour la suite de la course
  - Le fait d'être bien aligné vers le prochain checkpoint concerne votre vecteur vitesse et non pas l'angle vers lequel s'oriente votre vaisseau. Utilisez bien les composantes du vecteur vitesse **getShipSpeedX** et **getShipSpeedY** pour vous aider dans cette tâche
  - Si votre vitesse est trop réduite, partir en drift sera contre-productif

# Optimisation de trajectoire

- Compensation de vecteur vitesse :
  - Avec l'inertie, votre vaisseau va très probablement dépasser le prochain checkpoint et va être poussé vers l'extérieur du virage que vous essayez d'effectuer
  - Il va donc falloir compenser ce vecteur vitesse en vous tournant un peu plus vers l'intérieur du virage
  - Attention à ne pas trop compenser sinon votre vaisseau va perdre la vitesse qu'il avait au début du virage et cela deviendra contre-productif
  - il s'agira donc de 'limiter' d'un angle maximum la compensation
  - Combiner cette fonctionnalité avec un freinage adéquat avant d'arriver sur le checkpoint pourrait s'avérer très performant



# Economie d'énergie

- Ralentir suffisamment en arrivant sur des checkpoints de recharge vous permettra de faire remonter votre niveau de batterie et peut vous permettre de recharger moins de fois au cours de la course mais vous perdez du temps à chaque tour c'est donc un compromis à trouver
- Globalement, accélérer vous fait perdre de l'énergie mais vous permet de gagner du temps sur chaque tour donc le conseil (totalement subjectif) des formateurs c'est :

**« Full Throttle » !!!**

- Arrêt complet à la recharge :
  - Il faut tenter de se stabiliser le plus rapidement possible au dessus du point de recharge afin de diminuer au maximum le temps d'attente
  - Pour cela il faut garder le « pied sur le frein » quand on est au dessus
  - Si le vaisseau, avec l'inertie, s'éloigne du checkpoint il faut réaccélérer dans sa direction et forcer le freinage complet à nouveau une fois au dessus
  - Se stabiliser au dessus du point de recharge aura un effet très bénéfique sur le temps global

# Economie d'énergie

- Point de recharge le plus proche
  - Votre boucle qui recherche le premier checkpoint de recharge peut être modifiée en vérifiant la distance entre chaque checkpoint et votre vaisseau, et ne retenir QUE le checkpoint le plus proche
  - Ainsi, dans les courses où il y a plusieurs points de recharge, vous vous assurerez de vous diriger vers le plus proche à tout moment de la course (vous ne savez pas quand votre vaisseau aura besoin de se recharger)
  - Vous atteindrez le point de recharge plus tôt → Vous consommerez moins d'énergie pour atteindre le point de recharge → Vous réduirez le temps de recharge (il y a moins à recharger)
- Recharge Now !
  - Globalement le principe que nous avons adopté c'est de passer en mode recharge quand la batterie passe sous un seuil que nous considérons comme critique
  - Mais il peut s'avérer intéressant de recharger son vaisseau avant d'atteindre ce seuil critique (un seuil légèrement plus haut) car le prochain checkpoint devant nous possède la capacité de recharge
  - Cette technique permet d'éviter de passer le checkpoint à fond et, une fois lancé dans la suite de la course, le niveau de batterie devient faible et le vaisseau stoppe ce qu'il fait pour faire demi tour vers le point de recharge : ici on anticipe de manière simple la perte d'énergie et on s'arrête au premier point de recharge rencontré

# Economie d'énergie

- Historique de consommation
  - On peut améliorer le principe global de recharge en tenant à jour un historique de la consommation « entre checkpoints » : on mesure le niveau de batterie à chaque changement de checkpoint, et on stocke les différences afin d'avoir un histogramme des consommations par portion de course
  - Les estimations de consommation, vous permettront d'avoir une moyenne de consommation par distance
  - Cette estimation de la consommation sur la course permet de choisir le moment optimum pour recharger en fonction de sa position dans la course et de la distance du plus proche point de recharge
  - Vous l'avez compris, cette fonctionnalité nécessite beaucoup plus de code : elle est donc réservée aux plus motivé(e)s ^^



# Utilisation du Boost

- Le boost permet de voir sa vitesse augmenter très vite pendant un court instant
- Il ne consomme pas d'énergie ; il est donc très utile pour repartir d'un point de recharge par exemple, où la vitesse est nulle
- Boost simple :
  - Il faut vérifier que le niveau de boost est de 100% avant de l'utiliser sinon le vaisseau se verra ralentir
  - Il faut vérifier que le vaisseau est bien orienté dans la direction que l'on souhaite avant de l'activer
  - Il faut vérifier que le prochain checkpoint n'est pas trop proche, sinon le boost vous fera partir trop loin derrière le checkpoint et cela sera contre-productif
  - Cette première étape d'utilisation du boost sera simplement un test de distance et d'alignement vers votre prochaine cible
  - Attention tout de même à l'inertie : il est sage de vérifier que le vaisseau est bien orienté mais aussi que le vecteur vitesse est bien orienté pour garantir une utilisation optimale du boost

# Utilisation du Boost

- Boost avec 2 checkpoints alignés :
  - Dans le paragraphe précédent nous avons indiqué qu'il fallait que la distance du prochain checkpoint soit grande pour utiliser sans risque le boost
  - Il est possible de modifier légèrement cette condition si les 2 prochains checkpoints sont « alignés », alors cela veut dire que nous avons devant notre vaisseau une très grande ligne droite et c'est avec la distance du second checkpoint que nous devons faire la comparaison
  - D'un point de vue mathématiques, il sera peu probable que le vaisseau et les 2 checkpoints soient correctement alignés mais ici il faut tester que la variation de l'angle n'est pas trop importante afin de pouvoir activer le boost malgré tout
- Boost et économie d'énergie :
  - Il est possible de réduire sa consommation en arrêtant (ou diminuant) l'accélération, et ce, tant que la vitesse du vaisseau est sous l'influence du boost
  - Attention tout de même, le fait d'arrêter d'accélérer ne permettra plus de compenser la trajectoire il convient d'être prudent quant à la manière de le faire

