

PISCINE CODING FACTORY

Pierre-Henri FRICOT
Romuald GRIGNON
Robin PENEAE
Joachim THIBOUT

Coding Factory by ESIEE-IT
CCI Paris Ile-de-France
Cergy-Pontoise / Paris-Montparnasse

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 débiterons 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 des 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

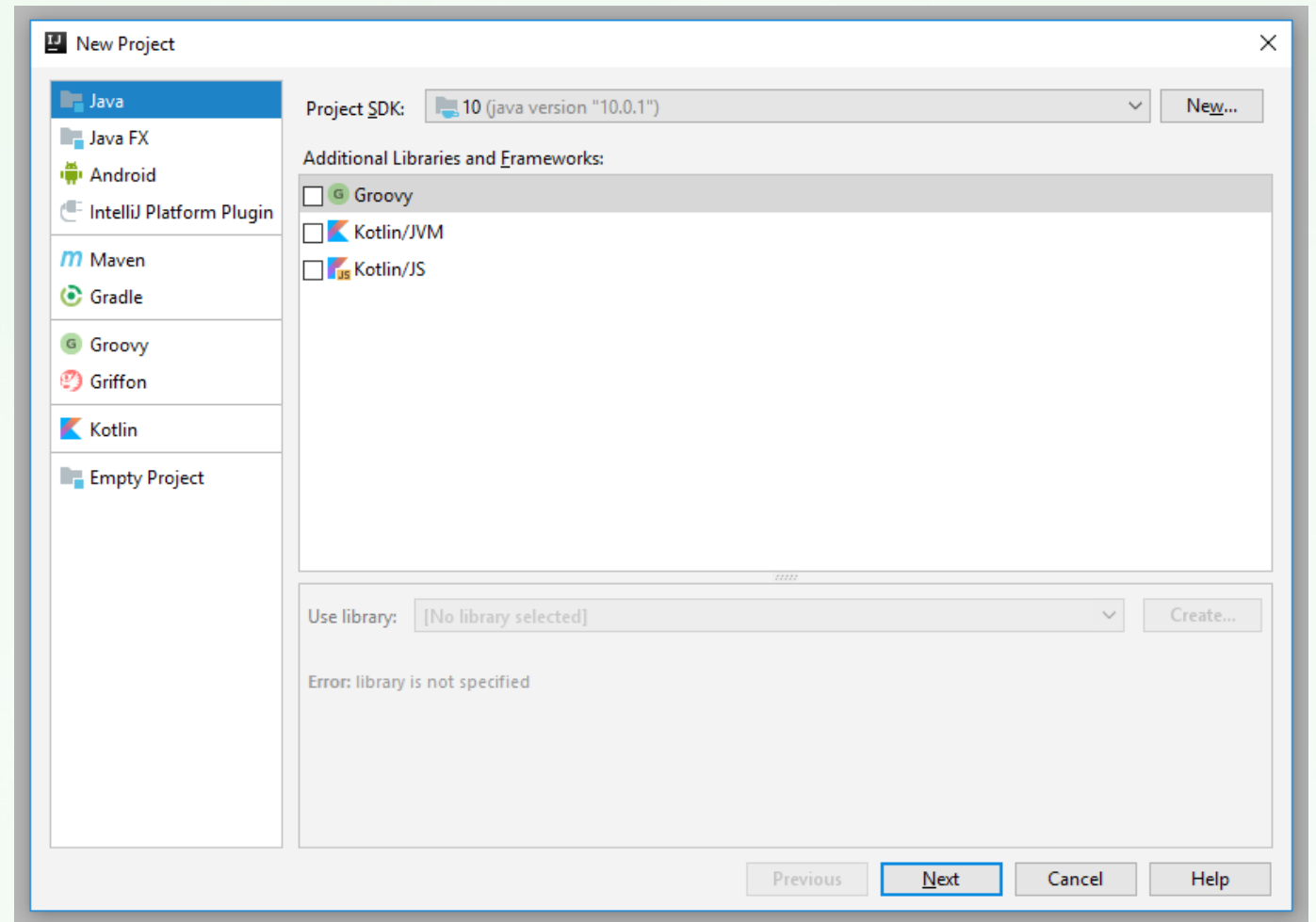
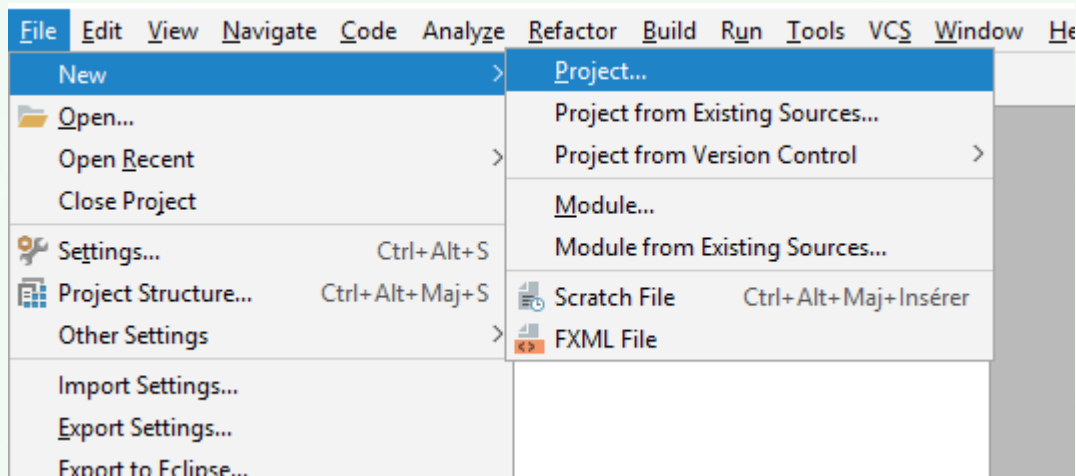
Environnement de travail

- 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)
 - <https://www.jetbrains.com/idea/download/>
- 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)

NOUVEAU PROJET IntelliJ

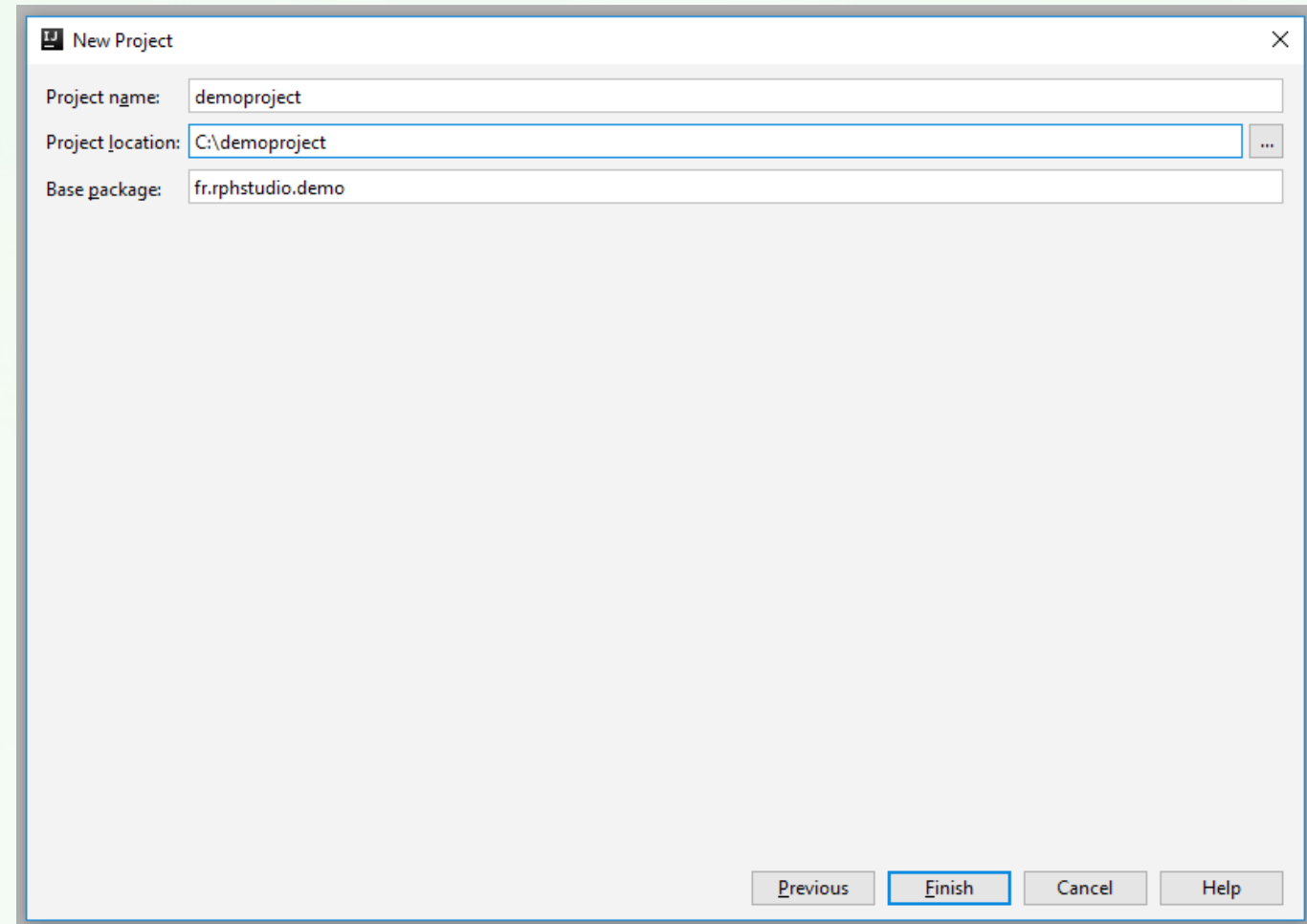
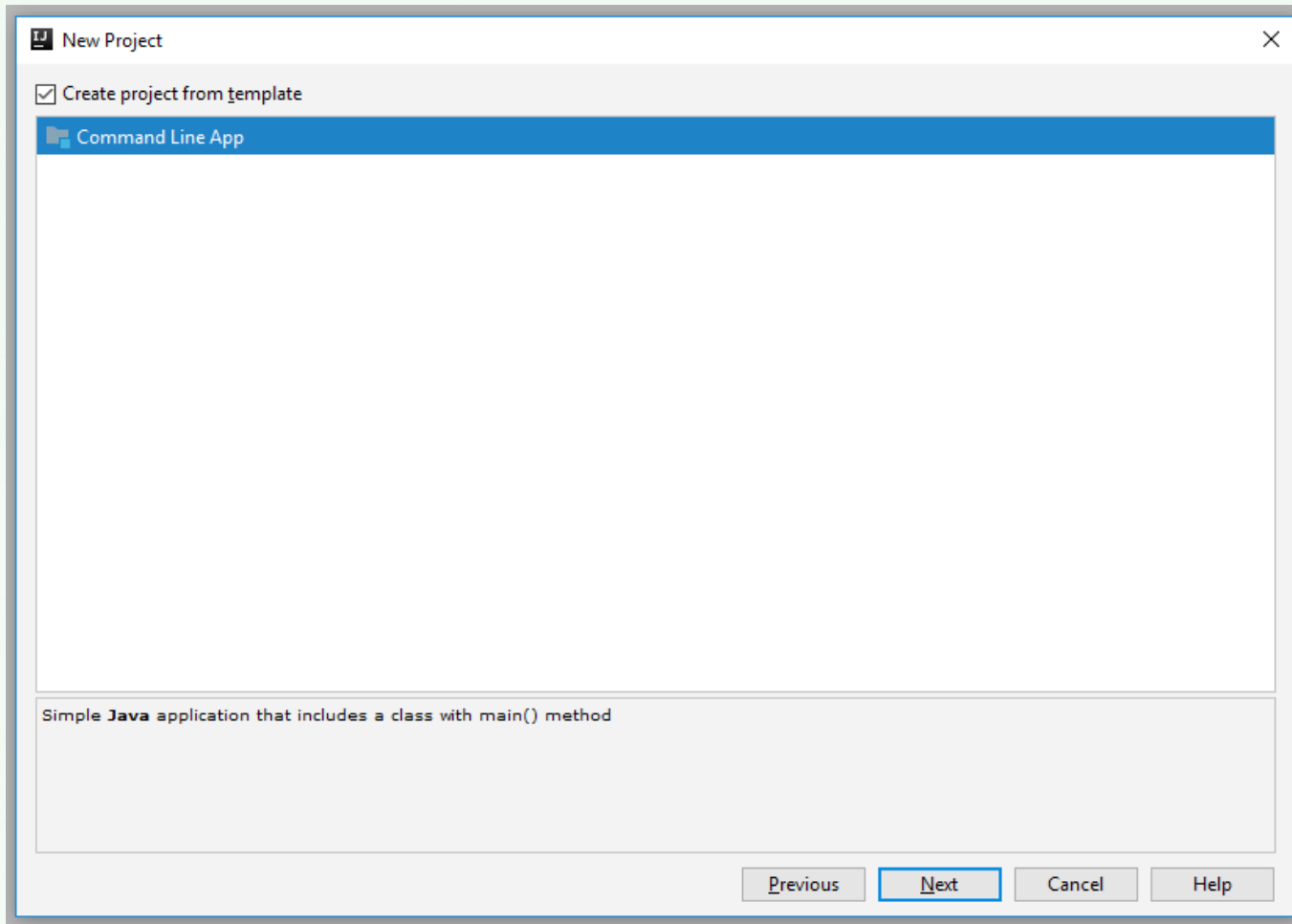
Nouveau projet IntelliJ

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



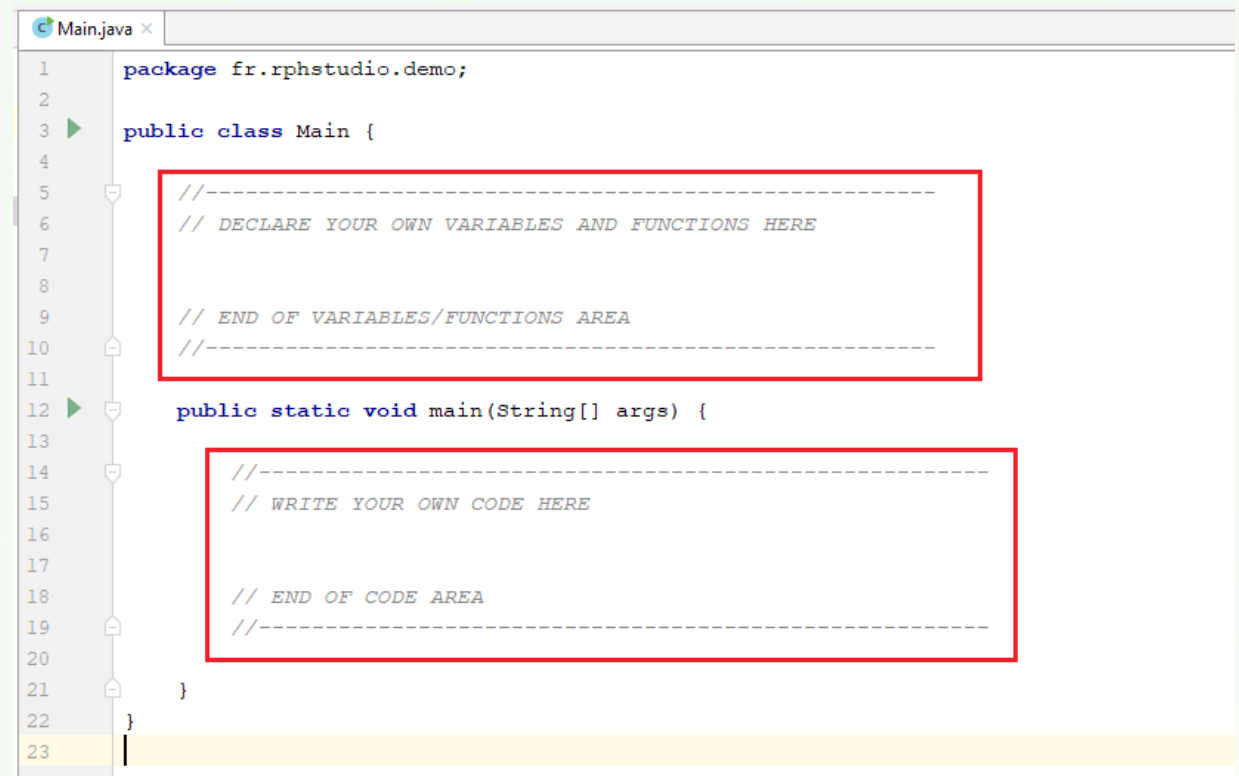
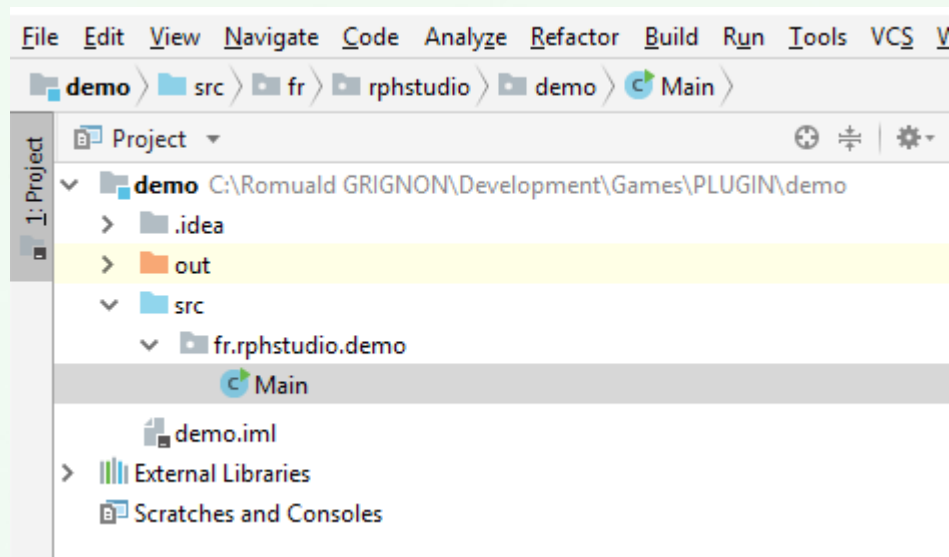
Nouveau projet IntelliJ

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



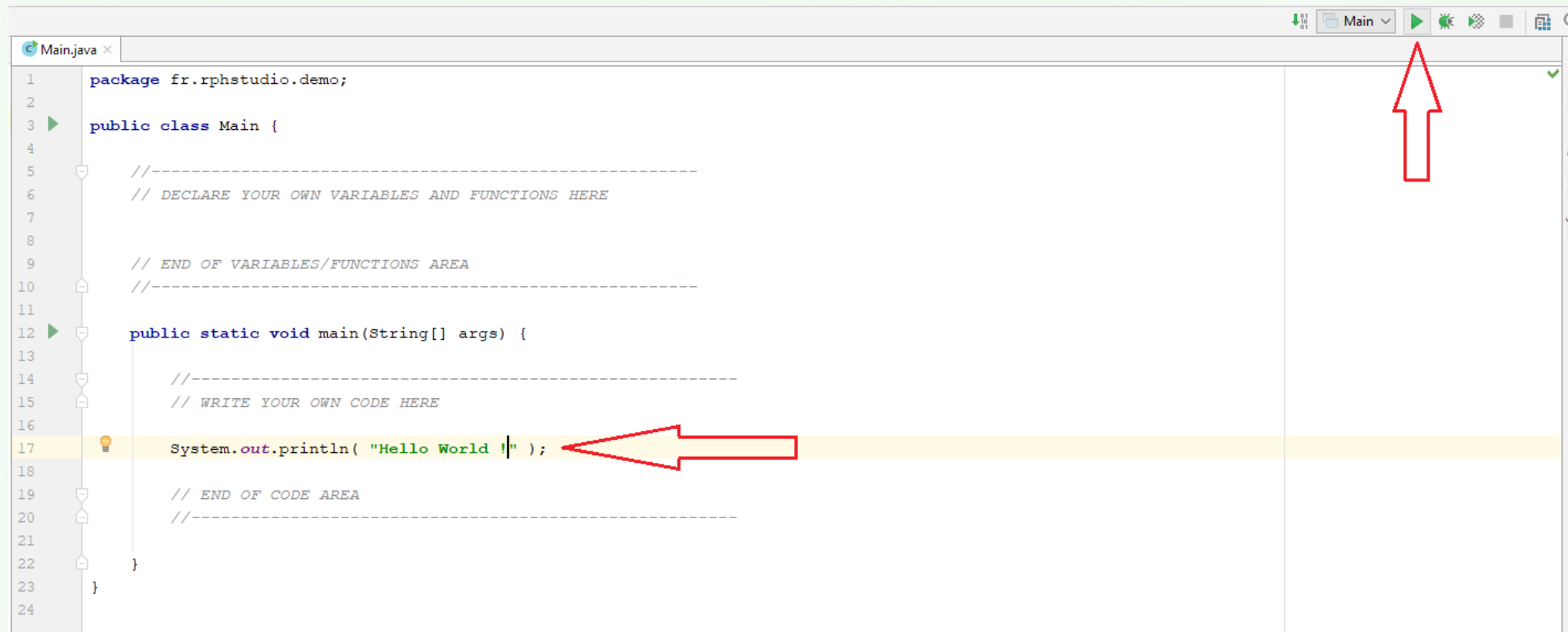
Nouveau projet IntelliJ

- 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

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 ← valeur arbitraire

SI A est supérieur ou égal à 5

ALORS

A ← A * 10

SINON

A ← 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 si le résultat du produit est positif
 - 2×2 donne un résultat positif
 - -3×2 donne un résultat négatif
 - -3×-3 donne un résultat positif
- 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

Branchements conditionnels : exercices

Déterminez si la valeur d'une année est bissextile ou non. Une année est bissextile si elle satisfait une de ces 2 conditions (l'une OU l'autre) :

- L'année est divisible par 4 ET NON divisible par 100
- L'année est divisible 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 ← valeur arbitraire

B ← valeur arbitraire

C ← 0

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

FAIRE

 C ← 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 ← valeur arbitraire

R ← 1

TANT QUE N>1

FAIRE

 N ← N - 1

 R ← 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
 - Exemple : $6^3 = 6 \times 6 \times 6$
- Affichez la table de multiplication de la valeur d'une variable

Les boucles : exercices

- Calculez la somme des entiers de 1 à N, et stoppez votre programme si cette somme dépasse la valeur M
- Affichez le miroir d'une String (ex : pour une valeur de "Coding", affichez "gnidoC")
 - ◆ Vous pouvez utiliser `charAt()` pour trouver le caractère à un indice donné dans une String
 - ◆ Vous pouvez utiliser la fonction `String.length()` pour avoir la taille de la chaîne.
 - ◆ `String name = "Coding" ;`
 - ◆ `int taille = name.length() ;`
 - ◆ `System.out.println(taille) ; // affiche 6`

LES FONCTIONS

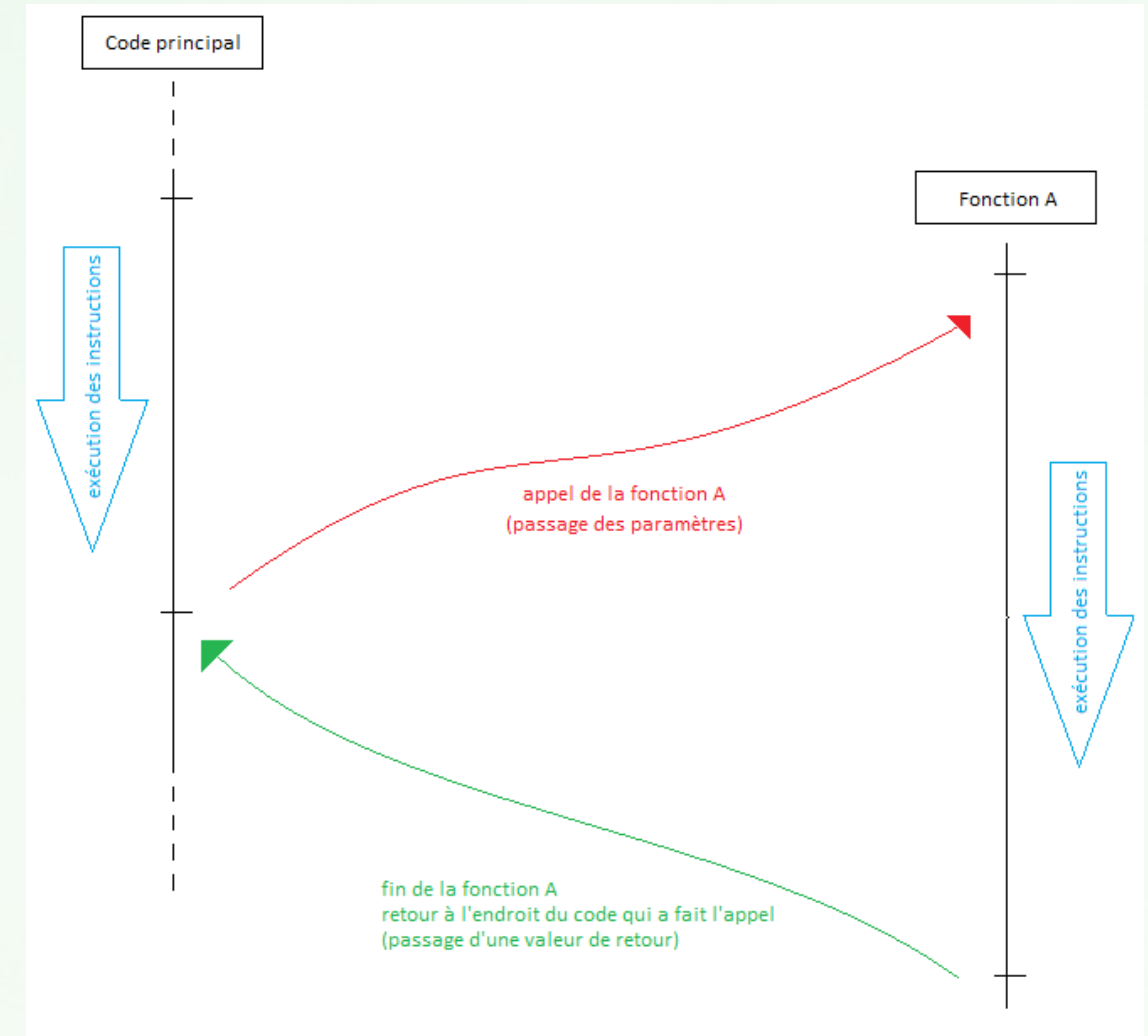
Fonctions

- 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 »

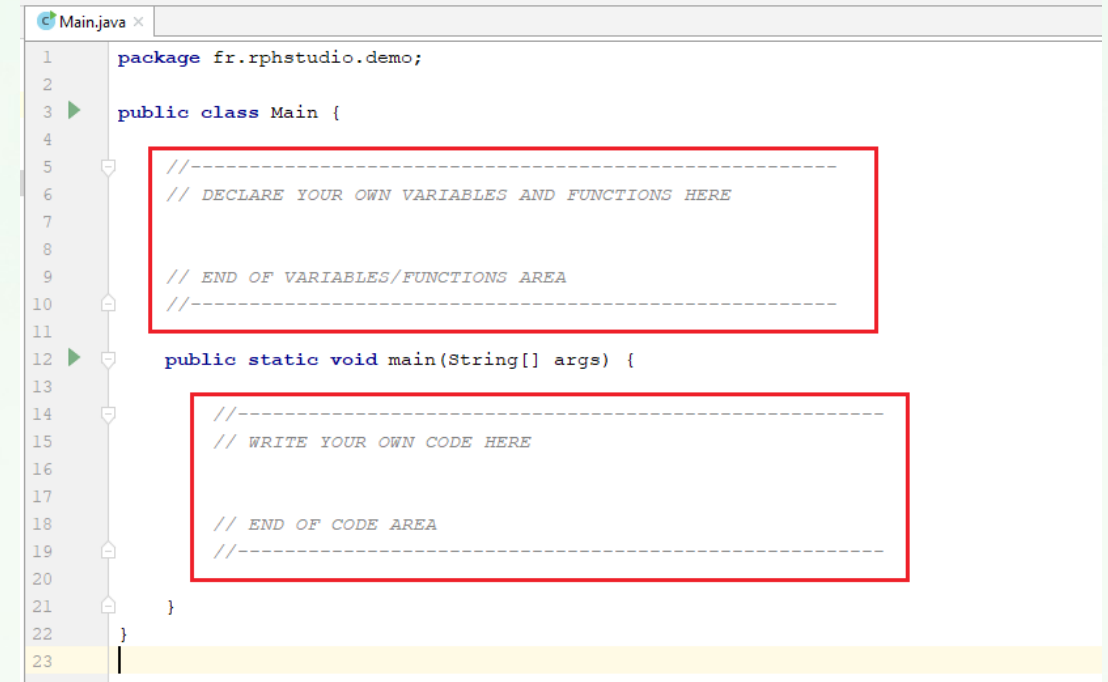
Fonctions

- 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     public static void main(String[] args) {
12
13         //-----
14         // WRITE YOUR OWN CODE HERE
15
16         // END OF CODE AREA
17         //-----
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
 - Valeur minimum entre 2 nombre
 - Exemple : `min(2, 9)` retourne le résultat 2
- Fonction « abs » : 1 entier en paramètre, 1 entier en sortie
 - Valeur absolue d'un nombre
 - Exemple : `abs(-5)` retourne le résultat 5
- Fonction « sum » : 1 entier N en paramètre, 1 entier en sortie (somme des N entiers)
 - Fait la somme de 1 jusqu'à N
 - Exemple : `sum(4)` retourne le résultat 10
- Fonction « power » : 2 entiers en paramètres, 1 entier en sortie
 - Calcul la puissance de a par b
 - Exemple : `power(2, 4)` retourne le résultat 16

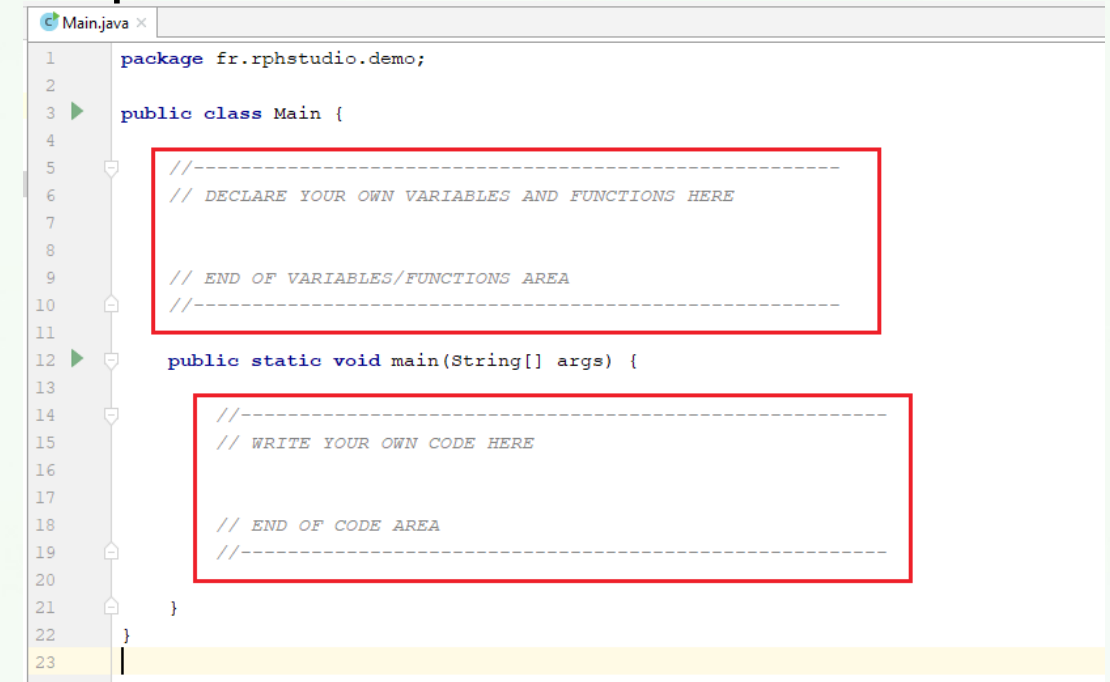
Les fonctions : exercices

- Fonction « stringMirror » : 1 string en paramètre, 1 string en sortie
 - Affichez le miroir d'une String (ex : pour une valeur de "Coding", affichez "gnidoC")
 - Exemple : stringMirror("Coding") retourne le résultat "gnidoC"
- Fonction « displayMultTable » : 1 entier en paramètre, affichage de la table
 - Affiche la table de multiplication du nombre passé en paramètre
- Fonction « isLeapYear » : 1 entier en paramètre, 1 booléen en sortie
 - Vérifie qu'une année passée en paramètre est bissextile
 - Exemple : isLeapYear (2020) retourne le résultat vrai
 - Exemple : isLeapYear (2021) retourne le résultat faux
- Fonction « isCorrectDate » : 3 entiers en paramètres, 1 booléen en sortie
 - Vérifie qu'une date passée en paramètre est valide
 - Exemple : isCorrectDate(2021, 09, 31) retourne le résultat faux
 - Exemple : isCorrectDate(2021, 09, 30) retourne le résultat vrai

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

Exercice de synthèse (BONUS)

- Programmer un Morpion permettant à 2 joueurs de s'affronter
- A chaque tour :
 - Afficher l'état de la grille
 - Le jouer courant devra saisir les coordonnées pour mettre X ou O
- Exemple :
 - A quelle ligne voulez-vous jouer ? 1
 - A quelle colonne voulez vous jouer ? 3
 - Le programme mettra donc une X a la ligne 1 case 3
- Pour lire l'entrée clavier, vous pourrez utiliser le code suivant :

```
Scanner sc = new Scanner(System.in);  
  
int i = sc.nextInt();
```


Exercice de synthèse (BONUS)

- Vous devrez stocker l'état de la grille dans un tableau
- Pour déclarer un tableau à une dimension en Java
 - `String[] tab = new String[3] ; // Je crée un tableau de String avec 3 cases`
 - `tab[0] = "Hello" ; // J'affecte la valeur Hello dans la 1ère case du tableau (indice 0)`
 - `System.out.println(tab[0]) ; // Affiche Hello, la valeur dans la 1ère case du tableau`
- Puisque le morpion est une grille, il faudra déclarer un tableau à double dimensions (une matrice)
 - `String[][] tab = new String[3][3] ; // Je crée un tableau de String de 3x3`
 - `tab[0][1] = « Bye » ; // J'affice la valeur Bye dans la 1ère ligne, 2ème case`

Exercice de synthèse (BONUS)

- Quelques indications supplémentaires !
- Le programme doit continuer son exécution tant que la partie n'est pas terminée :
 - Joueur 1 gagne
 - Joueur 2 gagne
 - Match nul
- Il y aura donc une boucle « while » qui continuera tant qu'une de ces conditions n'est pas respectée
- Cette boucle doit :
 1. Afficher l'état de la grille
 2. Demander au joueur courant la case dans laquelle il veut jouer
 3. Jouer le coup et regarder si une condition de fin de partie est rencontrée
 4. Et on recommence !

Exercice de synthèse (BONUS x2)

- Programmer le jeu 2048
- Afficher une grille de 4x4 avec 2 chiffres 2 placés aléatoirement
- Le joueur dispose des actions suivantes :
 - Flèche du haut : déplacer les tuiles vers le haut
 - Flèche de droite : déplacer les tuiles vers la droite
 - Flèche du bas : déplacer les tuiles vers le bas
 - Flèche du gauche : déplacer les tuiles vers le gauche
 - Q : quitter le jeu
- Quand le joueur déplace les tuiles, chaque tuile « glisse » dans la direction donnée :
 - Si elle rencontre une tuile de même valeur, les 2 tuiles s'additionnent ($2 + 2 = 4$)
 - Si elle rencontre une tuile d'une autre valeur, elle s'arrête à la case juste avant
- Une nouvelle tuile apparaît sur une case libre
 - Case aléatoire
 - Valeur comprise entre [2, tuile max de la grille]

Exercice de synthèse (BONUS x2)

- L'objectif est d'atteindre 2048
- Idées supplémentaires
 - Touche R pour « retry » et redémarrer une nouvelle partie
 - Rendre le chiffre de départ paramétrable (ex. commencer à 8 plutôt qu'à 2)
- Vous pouvez colorer votre sortie dans le terminal
- Pour pouvez « vider » votre terminal

```
System.out.print("\033[H\033[2J");
```

```
System.out.flush();
```