B0 - HUB

B-INN-000

Introduction au Java

Programmation Orienté Objet

Introduction au Java

language: java
compilation: javac, gradle, automated compilation with IDE
build tool: gradle

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

Découverte du Java

Introduction

- Qu'est-ce que le Java ?
  - Java est un langage de programmation orienté objet, typé, "compilé" qui fonctionne grâce à une Java Virtual Machine (**JVM**)
- Qu'est-ce que la **JVM** ?
  - La JVM est une machine virtuelle qui permet l'interprétation du code java compilé (**.class**), et aussi des archives java (**.jar**)
  - C'est l'un des avantages du Java ! Grâce à la **JVM** le code est très portable, il peut être exécuté partout tant qu'une **JVM** est présente !
  - D'autres langages peuvent aussi fonctionner sur la **JVM** c'est le cas du **Scala** et du **Kotlin**

Pré-requis

Si vous n'avez pas déjà installé les outils nécessaires à ce workshop, suivez les instructions situées ici :

`https://github.com/alwyn974/workshop-java/blob/master/REQUIREMENTS.md`

Le Java ! La théorie

Les modifieurs de visibilité et d'accès

En java, on peut changer la visibilité d'une variable, d'une classe, d'une fonction, pour cela il faut utiliser des **keyword** spécifique

- Les modifieurs de visibilité
  - `private` personne ne peut modifier/accéder à cette variable à part la classe actuelle
  - `protected` uniquement les classes enfants ou appartenant au même package peuvent modifier/accéder à cette variable
  - `public` tout le monde peut modifier/accéder à cette variable

Par défaut une variable, classe ou fonction déclarée sans modifieur de visibilité est en privé

- Les modifieurs d'accessibilité
  - `static` une variable ou une fonction statique peut être accédée sans instancier la classe, elle est aussi unique à cette classe
  - `final` pour faire simple, c'est l'équivalent du `const` en C/C++
  - `super` bon techniquement, ce n'est pas un modifieur d'accessibilité, au contraire il permet d'accéder à des méthodes/variables de la classe parente, il permet surtout d'appeler le constructeur de la classe parente

Les variables statiques sont toutes les mêmes peu importe l'instance de la classe

Le keyword **super** est pratique lorsque l'on veut utiliser la méthode de la classe parente et y ajouter quelque chose, ou bien pour modifier une variable de la classe parente à la place d'une variable déclaré dans la classe enfant

LES TYPES

Vu que le Java est basé sur le C++ et donc le C, les types restent à peu près les mêmes, pour certains ils sont améliorés ou ajoutés

Variable

- `boolean` permet de stocker les états `true` et `false` (comme bool en C/C++)
- `byte` permet de manipuler des entiers codés sur 1 octet (comme char)
- `double`, `int`, `long`, `float`, `short` restent les mêmes
- Vos objets à vous

Pour `double`, `int`, `long`, `float`, `short` se sont des types primitifs, et non pas des objets.
Ils possèdent donc leur équivalent en tant d'objet `Double`, `Integer`, `Long`, `Float`, `Short`

Classes

- `enum` les enums en java sont beaucoup plus complets que ceux en C/C++, vu que ce sont des objets, on peut y assigner plus qu'un simple Integer comme valeur
- `interface` permet de créer une interface
- `abstract class` permet de créer une classe abstraite
- `class` permet de créer une classe normale

Le polymorphisme - L'héritage

L'héritage en java fonctionne comme en C++ à quelques différences près.
On ne peut hériter que d'une seule classe, néanmoins on peut hériter de plusieurs interfaces
Exemple:

```
public class Animal {
    public void eat() {
        System.out.prinln("Animal can't eat");
    }
}

public interface ILiveable {
  void live();
}

public interface IMovable {
    void move(int x, int y);
}

public class Cat extends Animal implements ILiveable, IMovable {

    private final String name;

    public Cat(String name) {
        this.name = name;
    }

    //Overriden function from interface IMovable
    @Override
    public void move(int x, int y) {
        System.out.println("X: " + x + " Y:" + y);
    }

    //Overriden function from interface IMovable
    @Override
    public void live() {
        System.err.println("I'm dead bruh");
    }

    //Overriden method from Animal class
    @Override
    public void eat() {
        System.out.println("I'm hungry !!!");
    }

}
```

@Override est une annotation, elle permet de préciser qu'une méthode est override, cette annotation spécifique n'est pas obligatoire.
On peut créer nos propres annotations aussi. Les annotations seront très utiles dans les prochains workshop

Le polymorphisme - La surcharge

Comme en C++ on peut surcharger une méthode avec différents types de paramètre tout en ayant le même nom
Exemple:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("coucou"); //type of parameter is String
        System.out.println(1); //type of parameter is Integer
        System.out.println(0.0); //type of parameter is Double
        System.out.println(0.0f); //type of parameter is Float
        System.out.println(5 < 0); //type of parameter is Boolean
        System.out.println(new Object()); //type of parameter is Object, print an
            Object call the method toString() of the object
    }

}
```

Le polymorphisme - La redéfinition

Comme en C++, si l'on hérite d'une classe, d'une interface ou d'une classe abstraite, on peut surcharger les méthodes présentes dans la classe parente sauf si les méthodes ont le modifieur `private` ou `final`

Contrairement au C++ on ne peut pas hériter de toutes les variables/méthodes d'une classe, uniquement des méthodes et variables `public` et `protected`

Instancier un objet

C'est presque comme en C++, sauf qu'il faut utiliser new partout.

En java, on ne s'occupe pas de la gestion mémoire, c'est le Garbage Collector (**GC**) de la JVM qui s'en occupe automatiquement

Il faut quand même penser à faire un code un minimum optimisé

Exemple:

```
public class Main {

    public static void main(String[] args) {
        Object object = new Object(); //create object variable with Object type
        String str = new String("Hey"); //create str variable with String type
        String string = "toto"; //the good way to create a String
        int[] array = new int[42]; //create array with int[] type
    }

}
```

Structure d'un projet Java

Les projets Java sont principalement générés par des builds tools, comme `gradle` qu'on va utiliser dans ce workshop.
La structure reste plutôt basique un dossier `src` où l'on mettra tout notre code, néanmoins en utilisant gradle dans ce dossier vous aurez un dossier `test` et un dossier `main`.
Le dossier `test` est pour les tests unitaires et le dossier `main` pour mettre votre code.

Qu'est-ce qu'un package ?

En Java pour organiser votre code, on va créer ce qu'on appelle des `packages`, ce sont juste des sous dossiers qui seront dans `src/main`. Cela permet d'avoir un code organisé selon des catégories.

Généralement un package se construit sous la forme d'un ndd (nom de domaine) inversé + le nom du projet et différents sous packages.
Exemple:

```
re.alwyn974.monsuperworkshop
re.alwyn974.monsuperworkshop.subpackage
re.alwyn974.monsuperworkshop.epitech.workshop
```

Il y a une convention de nommage pour les packages, ils doivent être en minuscules. On n'est pas obligé de respecter cette convention, mais c'est mieux

Qu'est-ce qu'un import

En java si vous voulez avoir accès à d'autres classes, il vous faut utiliser les `import`, l'import est l'équivalent d'un include de header contenant les prototypes en C/C++.
Exemple:

```
import re.alwyn974.monsuperworkshop.Marvin;
import re.alwyn974.monsuperworkshop.*;
import static re.alwyn974.monsuperworkshop.Marvin.sayHello;
import static re.alwyn974.monsuperworkshop.Marvin.*;
```

Il existe 2 types d'import :
- L'import static, permet d'accéder à des variables et méthodes **statiques** sans préciser le nom de la classe
- L'import normal, permet d'accéder à une classe et ses méthodes qui se trouvent dans un autre package
L'import avec wildcard * permet d'importer toutes les classes d'un package ou toutes les méthodes/variables d'une classe avec l'import static.

En utilisant Intellij Idea et l'autocomplétion la plupart des imports se feront automatiquement, sauf en cas de conflits là il faudra choisir l'import spécifique.
Les classes contenues dans le même package sont accessibles directement

Création d'un nouvel objet

En Java tout code doit être dans une classe/interface/enum. On ne peut pas déclarer une méthode hors d'une classe comme en C/C++.
Pour créer une classe/interface/enum, il y a une structure à suivre :

- On commence par `public` ou rien (dans ce cas la classe sera en privée)
- Ensuite, on précise le type d'objet que l'on crée (`class`, `interface`, `enum`, `abstract class`)
- On précise le nom de l'objet
- Si on veut faire de l'héritage, on utilise `extends` et/ou `implements` pour les interfaces et on précise le nom de la classe parente
- Et après on ouvre des accolades

Exemple :

```
public class Marvin {
    //TODO: add code here
}

class ImNotAccessible {
    //TODO: add code here
}

public interface ITakeALotOfSpace {
    //TODO: add code here
}

public abstract class AbstractChooseAName {
    //TODO: add code here
}

public enum EnumTest {
    //TODO: add code here
}
```

Comme pour les packages, les noms d'objets doivent respecter une convention de nommage, pour les objets, on suit le PascalCase, qui consiste à mettre en majuscule la première lettre de chaque mot.

Le nom d'un objet doit être aussi le même que le fichier Java correspondant

Création d'un constructeur

Comme en C++ on peut ajouter un constructeur à une classe. Par contre, il n'y a pas de destructeur.
Pour créer un constructeur il y a une structure à suivre :

- On commence par un modifieur de visibilité (`public`, `private`, `protected`) ou rien (dans ce cas le constructeur sera en privé)
- On précise le nom de la classe
- On précise les paramètres du constructeur
- Et enfin on ouvre des accolades

Exemple:

```
public class Marvin {
    public Marvin(String name) {

    }
}

public class PrivateConstructor {
    private PrivateConstructor() {

    }
}

public class ProtectedConstructor {
    protected ProtectedConstructor() {

    }
}

public class ClassWithFinalVariable {
    final String toto;

    public ClassWithFinalVariable(String toto) {
        this.toto = toto;
    }
}
```

Si vous avez déclaré des variables `final` dans votre objet, sans les initialiser, il faudra les initialiser dans le constructeur

Création d'une variable ou une méthode

Pour créer une variable ou une méthode, il faut quelle soit dans une classe/interface/enum/abstract class. Il y a aussi certaines structures à suivre :

Variable

- On commence par un modifieur de visibilité (`public`, `private`, `protected`) ou rien (dans ce cas la variable sera en privée)
- On précise le type de la variable (String, int, boolean, etc.)
- On précise le nom de la variable
- On peut assigner la variable à une valeur ou non

Méthode

- On commence par un modifieur de visibilité (`public`, `private`, `protected`) ou rien (dans ce cas la méthode sera en privée)
- On précise le type de retour de la méthode (String, int, boolean, etc.)
- On précise le nom de la méthode
- On précise les paramètres de la méthode

Exemples:

```
public class Main {
    private static final String NAME = "Marvin";
    private boolean isMarvin = true;
    private int age = 42;

    private void method() {

    }

    private String getName() {
        return NAME;
    }

    private boolea isMarvin() {
        return isMarvin;
    }
}
```

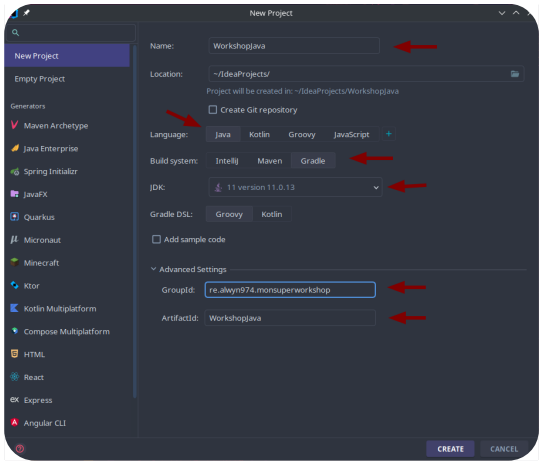
Ici on suit la convention de nommage du `camelCase` qui consiste à mettre en minuscule la première lettre du mot et en mettre une majuscule à chaque mot suivant.

Une petite spécification pour les variables `static final` on les écrit en `SCREAMING_SNAKE_CASE`

Le Java ! La pratique :3

Création d'un projet avec gradle

Si vous venez d'installer Intellij Idea, cliquez sur `New Project` sinon aller dans `File > New > Project`



Choisissez le nom du projet (ex : `WorkshopJava`)
Sélectionnez `Java`
Sélectionnez `Gradle`
Sélectionnez la version du JDK (11 si vous êtes sur le dump)
Dans `Advanced Settings` mettez votre package dans `GroupID` (ex : `com.github.username.workshop`)
Dans `ArtifactId` mettez le nom du jar (ex : `WorkshopJava`)
Il faudra alors attendre la fin de la création du projet gradle (Quand le dossier `src` apparait, c'est fini)

Lancement d'un projet avec Intellij Idea

Pour lancer un projet Java, à coter du `main` vous aurez un petit logo de lancement, il suffit de cliquer dessus et c'est parti !
Cela va créer une configuration sur Intellij pour lancer le programme avec ce `main` précisément, car en Java on peut avoir plusieurs mains et en choisir un en lançant un `.jar`

Build un jar avec Gradle

Si vous voulez générer un jar avec gradle il suffit de faire `gradle build` ou `./gradlew build`. Le jar sera généré dans le dossier `build/libs`
Vous pourrez le lancer avec `java -jar nomdufichier.jar`.
Vous aurez remarqué qu'en faisant cette commande il y a une erreur, c'est tout simplement, car aucun main n'a été précisé lors de la compilation

Pour résoudre ce problème, il suffit de mettre le `main` à utiliser dans le `build.gradle`:
Ajoutez donc ce bloc à la fin du build.gradle

```
jar {
  manifest {
    attributes "Main-Class" : "re.alwyn974.monsuperworkshop.Main"
  }
}
```

EXERCICE 1

- Créez votre propre package (ex: `com.github.username.workshop`)
- Dans ce package, créez une classe `HelloWorld.java`
- Créez une méthode main en `public static` et en prenant en argument un `String... args`
- Trouvez comment print un `Hello World !\n`

Concernant le paramètre que prend le main, le main peut prendre soit `String... args`, soit `String[] args`.
Les méthodes statiques ne peuvent accéder qu'à des variables statiques et des méthodes statiques, sans utiliser d'instance

Terminal

```
~/B-INN-000> Lancement avec intellij, parce que c'est plus rapide
Hello World!
```

EXERCICE 2

Fichier : FizzBuzz.java
Spécification : Contient un main

Faites une boucle qui va de 1 => 200 et pour chaque nombre suivez ces instructions :
- Si le nombre est divisible par 3 : on écrit Fizz
- Si le nombre est divisible par 5 : on écrit Buzz
- Si le nombre est divisible par 3 et par 5 : on écrit Fizzbuzz
- Sinon, on écrit le nombre

Terminal

```
~/B-INN-000> Lancement avec intellij, parce que c'est plus rapide
1
2
3 -> Fizz
4
5 -> Buzz
6 -> Fizz
7
8
9 -> Fizz
10 -> Buzz
11
12 -> Fizz
13
14
15 -> Fizzbuzz
```

EXERCICE 3

Fichier : GuessANumber.java
Spécification : Contient un main

Vous connaissez `Guess a number` ? C'est un jeu de devinette. Ou vous devez trouver le nombre qui a été créé aléatoirement, à l'aide d'indication du programme si votre nombre est inférieur ou supérieur au nombre créé.
Vous allez devoir le recréer, en faisant vos propres recherches.
Il faudra récupérer la valeur minimum et maximum en argument, si un argument n'est pas mis il faudra alors mettre en valeur par défaut: 1 et 100
Il faudra aussi afficher le nombre de tentatives à la fin du jeu.

```
Terminal                                                    - + x
~/B-INN-000> Lancement avec intellij, parce que c'est plus rapide
Your guess? 50
Too low!
Your guess? 100
Too high!
Your guess? 60
Too low!
Your guess? 70
Too high!
Your guess? 65
Too low!
Your guess? 69
Too high!
Your guess? 67
Too low!
Your guess? 68
You win!
It took you 7 tries.
```

Utilisez la classe Scanner et Random du package java.util

Une petite doc pour ajouter les arguments sur la configuration Intellij Idea : Configuration

EXERCICE 4

Fichier: Fibonacci.java
Spécification: Contient un main

Réimplémenter la suite de Fibonacci :
- En récursive (`recursiveFibonacci(int n)`)
- En itérative (`iterativeFibonacci(int n)`)
- Récupérer la valeur de n en argument dans le main, sinon mettre 10 par défaut

```
Terminal
~/B-INN-000> Lancement avec intellij parce que build, c'est long.
Suite de Fibonacci de 10:
Recursive: 55
Iterative: 55
```

Lien vers la suite de Fibonacci

EXERCICE 5

Sur les prochains exercices, on va faire un peu d'héritage et utiliser chaque type d'objet.
Pour cela vous allez créer un sous package `inheritance`

On va commencer par créer une class `AbstractVehicule` qui sera abstraite et devra contenir :
- Une méthode publique `void move()` qui sera abstraite
- Une variable privée et finale `String name` qui sera le nom du véhicule et aura un Getter
- Créer un constructeur protégé qui prendra `String name` en paramètre et initialisera la variable finale `name`
- Une méthode publique `double getSpeed()` qui sera abstraite
- Une surcharge de la méthode `String toString()` qui retournera le nom du véhicule et sa vitesse

```
Terminal
~/B-INN-000> La méthode toString() devrait renvoyer quelque chose comme ça:
AbstractVehicule: {name: UnNomRandom, speed: 0}
```

EXERCICE 6

On va créer un petit enum `VehiculeType` pour notre classe `AbstractVehicule`

L'enum `VehiculeType` devra contenir :
- Un **constructeur** qui prendra une `String` en argument et qui sera le type du véhicule.
- Une `String` en privée et finale qui sera le nom du type de véhicule
- Un getter `String getType()` qui retournera le type du véhicule

Comme type de véhicule, il nous faudra au moins 3 types de véhicule différents :
- Car
- Plane
- Boat

Maintenant, il faut modifier la classe abstraite `AbstractVehicule` pour qu'elle contienne un getter abstrait `VehiculeType getType()`

EXERCICE 7

Maintenant qu'on a un type de Véhicule, ce serait bien d'ajouter un peu de couleur non ?

On va donc modifier notre classe `AbstractVehicule` pour ajouter :
- Une variable privée de type `Color` avec un getter/setter
- Ajouter un autre constructeur de `AbstractVehicule` qui prendra le nom et la couleur en paramètre toujours en protected
- Modifiez aussi la méthode `toString()` pour maintenant afficher la couleur en plus !

Vous connaissez sprintf ? Trouvez l'équivalent en Java pour facilité la méthode toString()

EXERCICE 8

Créer 3 sous packages à `inheritance` : `fly`, `drive`, `floaty`
Dans chaque package correspondant créer une interface `IFlyable`, `IDriveable` et `IFloatyable` :
- `IFlyable` devra contenir une méthode publique `void fly()`
- `IDriveable` devra contenir une méthode publique `void drive()`
- `IFloatyable` devra contenir une méthode publique `void floaty()`

EXERCICE 9

Créer 3 classes qui hériteront de `AbstractVehicule` et d'une interface `IFlyable`, `IDriveable` et `IFloatyable` :
- `Car`, `Plane`, `Boat`
- Chaque constructeur prendra le nom, la vitesse et la couleur en paramètre et sera publique
- Déclarer une variable protégée pour la vitesse
- Dans chaque méthode abstraite implémentée, afficher un message correspondant dans la console
- La méthode `getType()` devra retourner le type de véhicule correspondant

```
Terminal
~/B-INN-000> Exemples:
move => "Moved"
drive => "Driving"
fly => "Flying"
floaty => "Floating"
```

EXERCICE 10

Fichier: Main.java

Créer une méthode qui prendra en paramètre un véhicule et qui affichera son nom, son type, sa couleur et sa vitesse (`showVehicule`)

- Faites une liste d'`AbstractVehicule` et ajouter une dizaine de véhicules dans cette liste. (Différents Véhicules)
- Pour chaque élément dans la liste, appeler la méthode `showVehicule`
- Afficher le nombre de véhicules dans la liste
- Afficher le nombre de véhicules ayant le type `Car`
- Afficher le nombre de véhicules ayant le type `Plane`
- Afficher le nombre de véhicules ayant le type `Boat`

```
Terminal
~/B-INN-000> Exemple:
Name: HydroJet
Color: java.awt.Color[r=255,g=0,b=0]
Speed: 300.0
Type: PLANE
...
Vehicules: 10
Cars: 4
Planes: 3
Boats: 3
```

Javadoc de l'interface List : lien

EXERCICE 11

Vous avez sûrement utilisé `getType()` pour compter le nombre de véhicules de chaque type. On va utiliser l'équivalent du `dynamic_cast` (C++) maintenant. (Uniquement pour vérifer le type d'une classe)
Vous allez devoir créer 3 listes de véhicules différentes :
- Une liste de `Car`
- Une liste de `Plane`
- Une liste de `Boat`

En utilisant la liste d'avant, vous devez ajouter les véhicules de chaque type dans les listes correspondantes.

En fonctions des listes, appeler la méthode correspondante à chaque type. (Méthode de l'interface)

Je vous laisse chercher l'équivalent du **dynamic_cast**. C'est un keyword spécifique à Java

Terminal

```
~/B-INN-000> Exemple:
Driving
Driving
Driving
Driving
Floaty
Floaty
Floaty
Flying
Flying
Flying
```

Psst regardez les `Stream` de java 8: lien

Exercice 12

Comme en C++ le Java possède des exceptions. On va donc créer une Exception pour l'enum `VehiculeType`.
Créez une exception : `VehiculeTypeNotFound` qui sera dérivée d'exception et devra prendre un paramètre `String message`

Ajoutez une méthode `fromString` qui prendra en paramètre une `String`, qui retournera un `VehiculeType` et sera publique et statique
Si la `String` ne correspond à aucun type de véhicule, la méthode devra lancer l'exception qu'on vient de créer
Testez votre code voir si l'exception est bien lancée en cas d'erreur.

Javadoc de l'exception : lien
Javadoc d'Enum : lien, psst, regardez la méthode `values()`
N'oubliez pas d'ajouter `throws` à la méthode

Interdiction de faire des conditions pour chaque type.

Tests unitaires

Java dispose de plusieurs bibliothèques pour faire des tests unitaires, on va utiliser la plus connue: `JUnit`
Dans le dossier `src/test/` recréer votre package, mais en rajoutant `test` à la fin

On va tester le package inheritance.
Créez une classe `InheritanceTest`

Importez statiquement toutes les méthodes de classe `Assertions` de junit.
Syntaxe d'un test unitaire :

```
public class InheritanceTest {
  @Test
  public void testVehiculeTypes() {
    assertEquals("Car", VehiculeType.CAR.getType()); //assertEquals is a static
        method from "Assertions"
  }
}
```

Exercice 13

Tester l'égalité entre différent String et le type d'un véhicule. Faites-le pour chaque type de véhicule.
Il faudra aussi tester la non-égalité entre un String et un type de véhicule.

EXERCICE 14

Maintenant, on va tester notre méthode `fromString`. Testez-la avec différents types de véhicule.
Des types valides et non valides.
Il faudra tester avec la méthode spéciale de JUnit pour gérer les exceptions !

Toutes les méthodes d'assertions de JUnit : lien

EXERCICE 15

Fichier: WorkshopTest.java

On va tester notre méthode itérative et récursive de Fibonacci. Testez que chaque méthode renvoie les mêmes valeurs pour les nombres de 0 à 20.

EXERCICE 16

Fichier: FizzBuzzTest.java
On va aussi pouvoir tester la sortie standard `System.out`, pour faire cela il faudra :

- Créer une variable finale `ByteArrayOutputStream` qui sera le nouveau flux de sortie standard
- Créer une variable finale `ByteArrayOutputStream` qui sera le nouveau flux de sortie d'erreur
- Créer une variable finale `PrintStream` qui sera l'ancien flux de sortie standard
- Créer une variable finale `PrintStream` qui sera l'ancien flux d'erreur

Vous allez devoir changer le flux de sortie standard et d'erreur pour que les tests unitaires puissent fonctionner, il faudra avant chaque test et le restauré après.

Vous allez pouvoir tester l'output que produit le **Main** de FizzBuzz. (Oui, oui, on peut tester un main directement)

Cherchez l'annotation qui vous facilitera la vie ici: Javadoc JUnit

Pour approfondir votre apprentissage

Beaucoup de site propose des cours sur le Java, en voici quelques un :
- Jmdoudoux.fr
- Koor.fr

Merci

Merci d'avoir suivi ce workshop, n'hésitez pas à me contacter si vous avez des questions ou des remarques.