Introduction

Mon premier programme

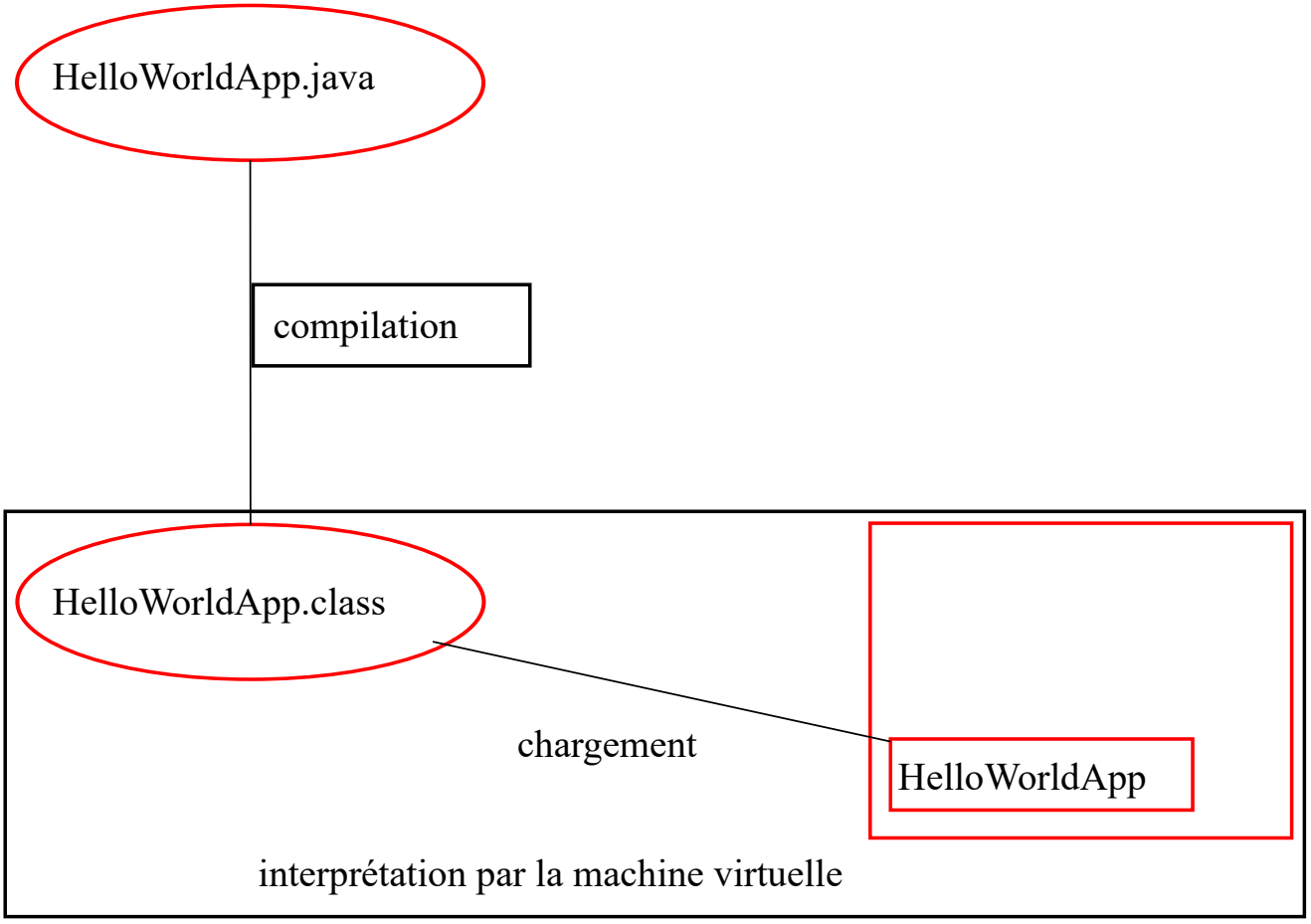
Ex.1. Hello World (5 min)

```java
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

- Copiez ce bloc de code dans un fichier appelé `HelloWordApp.java` (le nom sans extension est le nom de la classe).
- Compilez en byte code : **javac HelloWordApp.java** (un fichier `HelloWordApp.class` a été créé).
- Exécutez : **java HelloWordApp**. Que se passe-t-il ?

Fonctionnement ¶ §

Compilé en un langage intermédiaire unique (bytecode) puis interprété: “write once, run everywhere”.



En pratique...

- un répertoire par séance
- un fichier source par classe : *NomClasse*.java
- un éditeur pour écrire les fichiers sources
- un shell pour compiler (javac *NomClasse*.java)...
- (on obtient un fichier compilé par classe : *NomClasse*.class)
- ...et pour exécuter (java NomClasse)

Compiler proprement

- On peut diriger le bytecode (c'est-à-dire les fichiers .class) vers un répertoire donné:
  - `mkdir build`
  - `javac *.java -d build/`
- On peut exécuter une classe qui se trouve dans un répertoire différent de celui dans lequel on se trouve en modifiant le *classpath* (option `-cp`):
  - `java -cp build HelloWorldApp`
- Comme on a séparé fichiers sources et bytecode, on peut nettoyer le projet en supprimant le répertoire `build`.

Les bases du langage

Adresses utiles

Pensez à consulter régulièrement

- des tutoriaux
- des foires aux questions
- l'API standard

Utilisez des marques-pages.

Types primitifs

- void
- boolean
- char (16-bit, Unicode)
- byte (8-bit)
- short (16-bit)
- int (32-bit)
- long (64-bit)
- float (32-bit)
- double (64-bit)

Opérateurs

- arithmétiques: `+ - * / %`
- relationnels: `==` (égalité) `<= >= < > !=`
- logiques: `!` (non) `&&` (et) `||` (ou)
- incréments: `++ --`
- opérateurs sur les bits: `& | ^ ~ >> <<`
- affectation: `= -= += *= /=`
- conditionnel: `?:`

Instructions et blocs

- une expression est un assemblage de variables, d'opérateurs (et d'appels de méthodes), évaluée en une valeur.
- une instruction peut être:
  - une déclaration: `int x;`
  - une expression: `x = 2;`
  - une structure de contrôle: if, switch (sur types primitifs), for, while, do while
- un bloc d'instructions est délimité par des accolades. La portée d'une variable est celle du bloc dans lequel elle est déclarée.

Tableaux

- Déclaration

```
int[] array_of_int; // équivalent à : int array_of_int[];
int rgb[][][];
```

- Création/Initialisation

```
array_of_int = new int[42];
array_of_int[0] = 3;
rgb = new int[256][256][256];
int[] primes = {1, 2, 3, 5, 7, 7 + 4};
```

- Utilisation

```
int length = array_of_int.length;//length == 42
int integer = array_of_int[0];//integer == 3
```

Ex.2. Arguments (10 min)

- Ecrivez une classe `ArgumentsApp` qui affiche à l'écran les arguments passés à l'exécutable. Si aucun argument n'est passé, un message avertit l'utilisateur.

  Bien sûr, vous utiliserez le paramètre obligatoire `args` de `main`, qui est de type `String[]` (tableau de chaînes de caractères).

```
public static void main (String[] args) {
}
```

Le jeu des différences

Tout ça ressemble au **C**. Mais en y regardant de plus près, certains détails laissent penser que **Java** génère un monde un peu différent, peuplé d'objets:

- un tableau est un objet (avec notamment un champs `length`),
- les chaines de caractères de type `String` sont des objets (qui savent s'afficher sur la sortie standard),
- le flux de sortie standard `out` est un objet, lui-même champs de `System`, à qui on peut demander de réaliser un affichage (par `println`).

Programmation orienté-objet

Introduction

Dans ce petit exemple introductif, on comprend qu'on va décrire les tâches de la machine plus seulement comme une liste d'instructions, mais comme un système dynamique d'objets qui inter-agissent.

A l'exécution, sont crées en mémoire un ensemble d'objets. Chaque objet possède un état (qui peut changer) et des opérations qu'il sait réaliser. Les objets inter-agissent en s'adressant des requêtes les uns aux autres sur le mode "je te demande de faire telle opération".

Classe

Le programmeur va définir les caractéristiques d'une famille d'objets en écrivant une **classe**.

La classe a

- un **nom**
- des **membres**
  - les **champs** (= attributs) décrivent la structure de l'état des objets
  - les **méthodes** décrivent les opérations que savent réaliser les objets

Ex.3. Interrupteur/Classe (5 min)

Copiez et compilez.

```
/** Classe modelisant un interrupteur à bascule. */
class Interrupteur {
    /** booleen indiquant l'etat de l'interrupteur */
    boolean estEnMarche = false;
    /** Methode basculant l'état de l'interrupteur. */
    void basculer() {
        estEnMarche = (!estEnMarche);
    }
}
```

Attention, cette classe n'est pas exécutable, car elle ne contient pas de `main`, point d'entrée obligatoire de toute classe exécutable.

Instanciation

Un objet est manipulé via une variable dont le **type** porte le nom de sa classe. Cette variable contient une **référence** vers la zone mémoire allouée pour l'objet.

```
//déclaration d'une référence (aucun objet n'est créé)
Interrupteur unInterrupteur;
```

Pour créer en mémoire un nouvel objet (= **instance**), on utilise l'opérateur `new`, suivi de l'appel à une méthode portant le nom de la classe et appelée **constructeur**. Si aucun constructeur n'est écrit par le programmeur, celui-ci est automatiquement créé à la compilation.

```
//création de l'objet, référencé par la variable unInterrupteur
unInterrupteur = new Interrupteur();
```

Valeurs par défauts

Les *champs* d'une classe ont tous une valeur par défaut:

- boolean false
- char *\u0000* (null)
- byte (byte)0
- short (short)0
- int 0
- long 0L
- float 0.0f
- double 0.0d
- tout objet null

NB: Seulement les champs, pas les variables locales.

Initialisation des champs

La première fois que la classe `Interrupteur` est impliquée dans l'éxécution du programme (ex. un objet de type `Interrupteur` est créé), la machine virtuelle charge `Interrupteur.class` en mémoire.

Lorsque l'on crée un nouvel objet de type `Interrupteur`, suffisamment d'espace mémoire est alloué et mis à zéro (`estEnMarche` contient sa valeur par défaut: `false`).

Enfin, les champs sont initalisés dans l'ordre de déclaration (on affecte `false` à `estEnMarche`, ce qui est inutile, mais rend le code plus lisible), avant que le constructeur (ici, par défaut) soit appelé.

Interaction entre objets

Une fois qu'un objet est créé, un objet tiers (= **client**) peut lui envoyer des **requêtes** (= message), c'est-à-dire

- soit lire/modifier un champs

```
unInterrupteur.estEnMarche = true;
System.out.println( "est sur ON ?" + unInterrupteur.estEnMarche );
```

- soit lui demander de réaliser une opération qu'il sait faire.

```
unInterrupteur.basculer(); //je lui demande de changer d'état
```

```
System.out.println( "est sur ON ?" + unInterrupteur.estEnMarche );
```

Ex.4. Interrupteur/Test (10 min)

- Ecrivez le code client dans un fichier `InterrupteurTest.java` qui, en utilisant le champs `estEnMarche` et la méthode `basculer()`, teste:
  - que l'interrupteur est à l'état d'arrêt à sa création.
  - qu'il est à l'état de marche après la bascule.

```
/** Classe testant l'interrupteur à bascule. */
class InterrupteurTest {
    public static void main(String[] args) {
        Interrupteur i = new Interrupteur();
        //TODO
    }
}
```

Ce qu'il faut retenir

- Dans le code client:
  - on peut créer un objet en mémoire (par le constructeur précédé de l'opérateur `new`),
  - mais on ne les détruit pas: la machine virtuelle possède un **garbage collector** qui s'en charge.
  - l'objet est référencé par une variable dont le type est le nom de la classe,
  - on peut adresser des requêtes à un objet pour lire/modifier son état ou activer un de ses comportements,

Copie d'objets

Ex.5. Affectation (5 min)

Comparez ce que font ces deux blocs (dans un fichier `DemoAffectation.java`).

```
boolean b1 = false;
boolean b2 = b1;
b2 = !b2;
System.out.println( b1 );
```

```
Interrupteur i1 = new Interrupteur();
Interrupteur i2 = i1;
i2.basculer();
System.out.println( i1.estEnMarche );
```

Ex.6. Passage de paramètres (5 min)

Comparez ce que font ces deux fonctions (dans un fichier `DemoPassageParametres.java`).

```
static void faireBasculerBooleen(boolean unBool) {
   unBool = !unBool;
}
```

```
static void faireBasculerInterrupteur(Interrupteur unInterrupteur) {
   unInterrupteur.basculer();
}
```

Ce qu'il faut retenir

- L'affectation **copie** le contenu d'une variable dans une autre.
- Les passages de paramètres se font aussi par **copie**.
- Copie des **valeurs** pour les variables de type primitif.
- Mais copie des **références** (et pas des objets eux-même) pour les variables de type personnalisé.