`static`, `enum` et fichiers de projets

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1

Université Lille1 Sciences et Technologies | IEEA - FIL Informatique

Le rôle essentiel d'un moule est de permettre la création d'objets.
... mais un moule a également des caractéristiques

Il en est de même pour une classe.
Il existe des attributs et méthodes de classe.

Les attributs et méthodes de classes sont dits de **statiques**

Attributs de classe

La définition de chaque classe est unique, donc les attributs de classes **existent en un seul exemplaire**.
Ils sont créés au moment où la classe est chargée en mémoire par la JVM.

et ce quel que soit le nombre d'instances (y compris 0)

- Il *n'est pas nécessaire de disposer d'une instance* pour utiliser une caractéristique statique.

`static`

La déclaration des attributs de classe se fait à l'aide du mot réservé `static`

```
public class StaticExample {
   private static int compteur;
   public static double pi = 3.14159;
}
```

Usage : *accès via le nom de classe*

utilisation de la notation "." et respect des modificateurs

`StaticExample.compteur` n'est visible que par des instances de la classe
↪ attribut (privé) partagé par les instances

`StaticExample.pi` visible partout

Les méthodes aussi...

```
public class StaticExample {
   public static void staticMethod() {
      System.out.println("ceci est une méthode statique");
   }
}
```

Invocation :

`StaticExample.staticMethod()`

pas besoin d'instance !!!

- pas d'instance donc pas d'utilisation de `this` dans le corps d'une méthode statique

`static` : méthodes

l'usage de `static` doit être **limité** et **justifié**

- a priori quasiment **jamais**

pas réellement "objet", mais pratique...
plutôt réservé pour les méthodes "utilitaires"

cf. dans `java.lang.Math`
`java.net.InetAddress.getLocalHost()`

- cas particulier, la méthode **main**, sa signature doit rigoureusement être :

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

méthode appelée lors du lancement de la JVM JAVA avec comme argument `AClass`, les autres arguments sont les valeurs de `args[]`.

```
java AClass arg0 arg1 ...
```

∼ "programme à exécuter"

static : attributs

- à user avec parcimonie et pertinence
  - en `private` : "mémoire" partagée par les instances
    Illustration : compteur d'instances créées.

```
public class StaticIllustration {
   private static int counter = 1;
   private String name;
   public StaticIllustration() {
      this.name = "instance-"+(StaticIllustration.counter++);
   }
   public String getName() { return this.name; }
}
...
StaticIllustration si1 = new StaticIllustration();
StaticIllustration si2 = new StaticIllustration();
System.out.println("si1 -> "+si1.getName());
System.out.println("si2 -> "+si2.getName());
...
                                    +---------------------
                                    | si1 -> instance-1
                                    | si2 -> instance-2
                                    +---------------------
```


- avec `final` : création de constantes

```
public class ConstantExample {
   public static final float PI = 3.141592f;
   public static final String BEST_BOOK = "Le Seigneur des...";
}
```

  - le qualificatif **final** signifie qu'une fois initialisée la valeur ne peut plus être modifiée.
  - convention de nommage : les identifiants des constantes sont en majuscules et usage "_").
    ↪ `Boolean.TRUE`, `Double.MAX_VALUE`

NB : on peut utiliser `final` sans `static` et réciproquement

enum

(java ≥ 1.5)

`enum` permet la définition de types énumérés

```
public enum Saison { hiver, printemps, ete, automne;}
```

Référence des valeurs du type énuméré :

```
Saison s = Saison.hiver;
```

- En fait, création d'une classe avec un nombre prédéfini et fixe d'instances.
- Les valeurs du type sont donc des instances de la "classe `enum`".
  ↪ `Saison` est une classe qui a (et n'aura) que 4 instances,
  `Saison.printemps` est l'une des instances de `Saison`.

Méthodes fournies

Pour un type énuméré `E` créé, on dispose des méthodes.
Méthodes d'instances :

- **`name():String`** retourne la chaîne de caractères correspondant au nom de *this* (sans le nom du type).
- **`ordinal():int`** retourne l'indice de *this* dans l'ordre de déclaration du type (à partir de 0).

Méthodes de classe (static) :

- **`static valueOf(v:String):E`** retourne, si elle existe, l'instance dont la référence (sans le nom de type) correspond à la chaîne `v` (réciproque de `name()`).
- **`static values():E[]`** retourne le tableau des valeurs du type dans leur ordre de déclaration

(à compléter plus tard dans le cours)

Exploitation

```
public enum Saison { hiver, printemps, ete, automne;}

// ailleurs
public class Test {
  public void suivante(String nomSaison) {
    Saison s = Saison.valueOf(nomSaison);
    int indice = s.ordinal();
    Saison suivante = Saison.values()[(indice+1)%(Saison.values().length)];
    System.out.println("apres "+nomSaison+" vient "+suivante.name());
  }


  public static void main(String[] args) {
    Test t = new Test();
    if (args.length > 0) {
      t.suivante(args[0]);
    }
    else {
      t.suivante("hiver");
    }
  }
}
```

Que se passe-t-il ?

Le compilateur crée la classe (à peu près) :

```
public class Saison {
  private static int cpt =0;
  private String name;
  private int index;
  private Saison(String theName){
    this.name = theName;
    this.index = cpt++;
  }
  public static final Saison hiver = new Saison("hiver");
  public static final Saison printemps = new Saison("printemps");
  public static final Saison ete = new Saison("ete");
  public static final Saison automne = new Saison("automne");
  public String name() { return this.name; }
  public int ordinal () { return this.index; }
  public static Saison[] values() {
    return { Saison.hiver, Saison.printemps, Saison.ete, Saison.automne };
  }
  public static Saison valueOf(String s) { // à peu près
    if (s.equals("hiver") { return Saison.hiver; }
    else if (s.equals("printemps") { return Saison.printemps; }
    // idem pour ete et automne...
  }
}
```

- Constructeur **privé**.

Logistique à faire soi même en java $<$ 1.4

Ce sont des classes...

On peut donc ajouter des attributs, méthodes, constructeurs...

```
public enum Jour {
   lundi(true), mardi(true), mercredi(true),       // constantes
   jeudi(true), vendredi(true), samedi(false), dimanche(false);

   private final boolean travaille;                //attribut

   private Jour(boolean value) {                   // constructeur
      this.travaille = value;
   }

   public boolean estTravaille() {                 // méthode
      return this.travaille;
   }
}

// usage
for(Jour j :  Jour.values()) {
   System.out.println(j.name()+" vaut "+j.estTravaille());
}
```


javac et java

- JAVA est un langage compilé

compilateur (de base) = **javac**

NomClasse.java ⟶ *NomClasse*.class

Exécution d'un programme (le ".class") :

java *NomClasse [args]*

à condition que la classe *NomClasse* définisse la méthode statique

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


CLASSPATH

- La variable d'environnement **CLASSPATH** est utilisée pour localiser toutes les classes nécessaires pour la compilation ou l'exécution.
- Elle contient la liste des répertoires où chercher les classes nécessaires.
- Par défaut elle est réduite au répertoire courant (".").
- Les classes fournies de base avec le *jdk* sont également automatiquement trouvées.
- Il est possible de spécifier un "classpath" propre à une exécution/compilation :

  (WINDOWS) : `java/javac -classpath bid;.;/truc/classes;%CLASSPATH% ...`

  (LINUX) : `java/javac -classpath bid:.:/truc/classes:$CLASSPATH ...`

Paquetages

∼ bibliothèques JAVA

- regrouper les classes selon un critère (arbitraire) de cohésion :
  - dépendances entre elles (donc réutiliser ensemble)
  - cohérence fonctionnelle
  - ...
- un paquetage peut eux aussi être décomposé en sous-paquetages
- le nom complet de la classe `NomClasse` du sous-paquetage `souspackage` du package `nompackage` est :

`nompackage.souspackage.NomClasse`

notation UML : **`nompackage::souspackage::NomClasse`**

Utilisation de paquetages

- Utiliser le nom complet :
  `new java.math.BigInteger("123");`
- Importer la classe : **import**
  - Permet d'éviter la précision du nom de paquetage avant une classe (sauf si ambiguïté)
  - On peut importer tout un paquetage ou seulement une classe du paquetage.
  - La déclaration d'importation d'une classe se fait avant l'entête de déclaration de la classe.

```
import java.math.BigInteger;
public class AClass {
   ...  new BigInteger("123");
}
```

```
import java.math.*;
public class AClass {
   ...  new BigInteger("123");
}
```

L'importation `java.lang.*` est toujours réalisée.

Création de paquetage

elle est implicite

- *déclaration* : première ligne de code du fichier source :
  `package` ***`nompackage`***;
  ou `package` ***`nompackage.souspackage`***;
- convention : nom de paquetage en minuscules
  - Le paquetage regroupe toutes les classes qui le déclarent.
  - Une classe ne peut appartenir qu'à un seul paquetage à la fois.

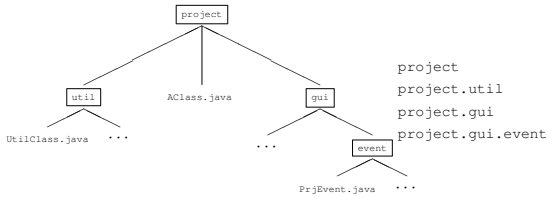
Assurer l'unicité des noms : utilisation des noms de domaine "renversés"

`fr.univ-lille1.fil.licence.project`

PB : Quand créer un nouveau paquetage ? Quoi regrouper ?

Correspondance avec la structure de répertoires

- À chaque paquetage doit correspondre un répertoire de même nom.
- Les fichiers sources des classes du paquetage **doivent** être placés dans ce répertoire.
- Chaque sous-paquetage est placé dans un sous-répertoire (de même nom).



à partir de la racine des paquetages :

`javac project\*.java` `javac project\util\*.java`

et les fichiers `.class` sont placées dans une hiérarchie de répertoire copiant celle des paquetages/sources

`java nompackage.souspackage.NomClasse` *`[args]`*

et il faut que le répertoire racine du répertoire `nompackage` soit dans le `CLASSPATH`

Le modificateur " "

- Nouvelle **règle de visibilité** pour attributs, méthodes et classes : **absence de modificateur** (mode "*friendly*").
- Tout ce qui n' est pas marqué est *accessible uniquement depuis le paquetage* dans lequel il est défini (y compris les classes).
- Intérêt : masquer les classes propres au choix d'implémentation
- Il existe **toujours** un paquetage par défaut : le paquetage "*anonyme*". Toutes les classes qui ne déclarent aucun paquetage lui appartiennent.

toujours créer un paquetage

- "Officialisation" de l'existence du paquetage
- Permettre une réutilisation sans craindre l'ambiguïté de nom.
- Permettre la distribution.

Archives : `jar`

- Regrouper dans une archive les fichiers d'un projet (compressés). Faciliter la distribution.
- syntaxe et paramètres similaires au `tar`

`jar ctxu[vfm0M] [nom-jar] [nom-manifest] [-C rép] fichiers ...`

c création	**v** "verbose" : bavard
x extraction	**f** spécifier le nom du fichier d'archives
t afficher "table"	**m** inclure le manifeste
u mettre à jour (update)	**etc.**

```
jar cf archive.jar Classe1.class Classe2.class
jar cvf archive.jar fr gnu
jar xf archive.jar
```



```
jar cvfm archive.jar mymanifest fr -C ../images/ image.gif
```

- manifest : fichier dans `META-INF/MANIFEST.MF`
  - `jar` "exécutable" :
    `Main-Class: classname` (sans `.class`)
    puis `java -jar archive.jar`

Utilisation des classes contenues dans une archive sans extraction :

- mettre le fichier jar dans le `CLASSPATH`.

```
SETENV CLASSPATH $CLASSPATH:/home/java/jars/paquetage.jar
```

ou `java -classpath $CLASSPATH:/home/java/jars/paquetage.jar ...`

JavaDoc

FichierClasse.java ⟶ *FichierClasse*.html

- Commentaires encadrés par `/** ... */`
- utilisation possible de tags HTML
- Tags spécifiques :
  - classe `@version`, `@author`, `@see`, `@since`
  - méthode `@param`, `@return`, `@exception`, `@see`, `@deprecated`
- conservation de l'arborescence des paquetages
- liens hypertextes "entre classes"

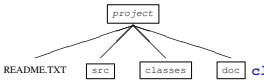
```
javadoc TestJavaDoc.java -author -d ../javadoc
```



```
/** description de la classe
* @author <a href=mailto:bilbo@theshire.me> Bilbo Baggins</a>
* @version 0.0.0.0.1
*/
public class TestJavaDoc {
   /** commentaire attribut */
   public int i;
   /** ... */
   public void f() {}
   /** commentaire sur la methode avec <em>tags html</em>
   * sur plusieurs lignes aussi
   * @param i commentaire paramètre
   * @return commentaire retour
   * @exception java.lang.NullPointerException commentaire exception
   * @see #f()
   */
   public String uneMethode(Integer i) throws NullPointerException{
      return("value");
   }
} // TestJavaDoc
```


Organisation des fichiers

Pour chaque projet, créer l'arborescence :



- **project** répertoire racine à terme le `jar` exécutable et le manifeste.
- **src** racine de l'arborescence des paquetages avec sources `.java`
- **classes** les `.class` générés
- **doc** la javadoc générée
- **+** ...

```
.../project/src> javac -d ../classes *.java package1/*.java etc.
.../project/src> javadoc -d ../doc .
.../project/classes> jar cvfm ../project.jar themanifest .
.../project> jar uvf project.jar src doc
```