

## static et enum

### static, enum, outils dev

## Programmation Orientée Objet

Jean-Christophe Routier  
Licence mention Informatique  
Université Lille 1



UFR IEAA  
Formations en  
Informatique de  
Lille 1

```
public class Disc {
    private double radius;
    public Disc(double radius) {
        this.radius = radius;
    }
    public double perimeter() { // 2πr
        return 2 * 3.14 * this.radius;
    }
    public double surface() { // πr²
        return 3.14 * this.radius * this.radius;
    }
}
```

### bonne pratique

il faut nommer les constantes !

$3.14 \mapsto \pi$

quel statut pour pi ?

## un attribut ?

```
public class Disc {
    private double radius;
    private double pi;
    public Disc(double radius) {
        this.radius = radius;
        this.pi = 3.14;
    }
    public double perimeter() { // 2πr
        return 2 * this.pi * this.radius;
    }
    public double surface() { // πr²
        return 3.14 * this.pi * this.radius;
    }
}
```

un attribut « this.pi » pour chaque instance de Disc...  
est-ce raisonnable ?

## 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`

accès via le nom de classe (utilisation de la notation ".")

```
public class Disc {
    private double radius;
    private static double pi = 3.14;
    public Disc(double radius) {
        this.radius = radius;
    }
    public double perimeter() { // 2πr
        return 2 * Disc.pi * this.radius;
    }
    public double surface() { // πr²
        return 3.14 * Disc.pi * this.radius;
    }
}
```

une seule version de `Disc.pi` quelque soit le nombre d'instances de `Disc`

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

respect des modificateurs

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

`StaticExample.pi` visible partout

static : attributs

à utiliser avec parcimonie et pertinence

- 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

- en `private` : "mémoire" partagée par les instances

```
public class Order {
    // attributs de classes : static
    private static final String ORDER_ID_PREFIX="order@";
    private static int counter = 1; // pour compter les instances créées
    // attributs d'instance
    private Client client;
    private Catalogue catalogue;
    private String id;
    public Order(Client client, Catalogue cata) {
        this.client = client;
        this.cata = cata;
        this.id = Order.ORDER_ID_PREFIX+Order.counter++;
    }
    public String getId() { return this.id; }
    ...
}

// utilisation :
Order o1 = new Order(c,k); // c,k supposés définis
Order o2 = new Order(c,k); // et initialisés
System.out.println("o1 -> "+o1.getId());
System.out.println("o2 -> "+o2.getId());
```

-----  
| si1 -> order@1  
| si2 -> order@2

## Exemple

### Documentation de la classe java.lang.System

public static final PrintStream out

The "standard" output stream. This stream is already open and ready to accept output data. Typically this stream corresponds to display output or another output destination specified by the host environment or user.

For simple stand-alone Java applications, a typical way to write a line of output data is:

```
System.out.println(data)
```

See the println methods in class PrintStream.

On souhaite disposer d'une méthode pour calculer le sinus d'un nombre.

Signature ? `public double sin(double x) { ... }`

Une méthode se définit dans une classe : `Trigonometry`

Utilisation ? = invocation  $\Rightarrow$  il faut un objet...

```
Trigonometry trigo1 = new Trigonometry();
trigo1.sin(45);
trigo1.sin(60);
Trigonometry trigo2 = new Trigonometry();
trigo2.sin(60);
new Trigonometry().sin(60);
```

intérêt des objets trigo1, trigo2, ... ?

## Méthodes de classe aussi...

## static : méthodes

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

Invocation : **pas besoin d'instance !**

```
StaticExample.staticMethod()
```

NB : pas d'instance donc **this n'a aucun sens** dans le corps d'une méthode statique

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

- a priori quasiment **jamais**  
pas "objet", mais pratique...  
réservé pour les méthodes "utilitaires" = fonctions  
= méthodes donc le traitement ne dépend pas de l'état d'un objet
- intérêt : éviter la création d'objet "jetable".

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

### Documentation de la classe System

public static Console console()

Returns the unique Console object associated with the current Java virtual machine, if any.

**Returns:** The system console, if any, otherwise null.

**Since:** 1.6

## types énumérés : enum

- 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"
```

(java ≥ 1.5)

`enum` permet la définition de types énumérés

```
public enum Season { winter, spring, summer, autumn; }
```

Référence des valeurs du type énuméré :

```
Season s = Season.winter;
```

- En fait, **un type énuméré est une classe** avec un nombre prédéfini et fixe d'instances.
- Les valeurs du type sont donc des **objets, instances** de la classe créée.

↪ `Season` est une classe qui a (et n'aura) que 4 instances, `Season.spring` désigne l'un des objets instances de `Season`.

Université Lille 1 - Licence Informatique	Programmation Orientée Objet	13	Université Lille 1 - Licence Informatique	Programmation Orientée Objet	14
static ○○○○○○○○○○	Types énumérés ○○○○○○○○	« Outils » de développement ○○○○○○○○○○○○○○○○○○	static ○○○○○○○○○○	Types énumérés ●○○○○○○○	« Outils » de développement ○○○○○○○○○○○○○○○○○○

## 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** (statiques) :

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

Pour un objet `e` d'un type énuméré `E`, on a toujours :

- `E.valueOf(e.name()) == e`
- `E.values()[e.ordinal()] == e`

### Affirmation

*Pour tester l'égalité de valeurs entre 2 références d'un même type énuméré on peut utiliser `==`.*

Pourquoi ?

(à compléter plus tard dans le cours)

## Exploitation

```
// Dans le fichier Season.java
public enum Season { winter, spring, summer, autumn;}

// Dans le fichier SeasonExample.java
public class SeasonExample {
    public void nextSeason(String seasonName) {
        Season s = Season.valueOf(seasonName);
        int index = s.ordinal();
        Season next = Season.values()[ (index+1)%(Season.values().length)];
        System.out.println("after "+seasonName+" comes "+next.name());
    }
}

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

## Un petit coup de décompilateur

On compile Thermometer.java  
et on appelle : `javap -private Thermometre` ~

```
Compiled from "Thermometre.java"
public class Thermometer {
    private float temp;
    public Thermometer(float);
    public float temperatureInCelsius();
    public void changeTemperature(float);
    public float temperatureInFahrenheit();
    public java.awt.Color temperatureColor();
}
```

## Une vieille connaissance

```
import java.awt.Color;
public class Thermometer {
    private float temp;
    public Thermometer(float tempInit) {
        this.temp = tempInit;
    }
    public float temperatureInCelsius() {
        return this.temp;
    }
    public float temperatureInFahrenheit() {
        return (9.0/5.0)*this.temp+32;
    }
    public void changeTemperature(float newTemp) {
        this.temp = newTemp;
    }
    public Color temperatureColor() {
        if (this.temp < 0) {
            return Color.BLUE;
        }
        else if (this.temp < 30) {
            return Color.GREEN;
        }
        else return Color.RED;
    }
}
```

## Et pour Season ?

```
public enum Season { winter, spring, summer, autumn;}
```

compilation puis `javap -private Season` ~

```
Compiled from "Season.java"
public class Season {
    public static final Season winter;
    public static final Season spring;
    public static final Season summer;
    public static final Season autumn;
    private Season(java.lang.String, int);
    public static Season[] values();
    public static Season valueOf(java.lang.String);
    public final int ordinal();
    public final java.lang.String name();
}
```

## Que se passe-t-il ?

Le compilateur crée la classe ~

```
public class Season {
    private String name;
    private int index;
    private Season(String theName, int idx){
        this.name = theName;
        this.index = idx;
    }
    public static final Season winter = new Season("winter",0);
    public static final Season spring = new Season("spring",1);
    public static final Season summer = new Season("summer",2);
    public static final Season autumn = new Season("autumn",3);
    public String name() { return this.name; }
    public int ordinal () { return this.index; }
    public static Season[] values() {
        return { Season.winter, Season.spring, Season.summer, Season.autumn };
    }
    public static Season valueOf(String s) { // à peu près
        if (s.equals("winter")) { return Season.winter; }
        else if (s.equals("spring")) { return Season.spring; }
        // idem pour summer et autumn...
    }
}
```

■ Constructeur **privé**.

Logistique à faire soi même en java ≤ 1.4

## Remarque

En utilisant un compteur statique d'instances :

```
public class Season {
    private static int cpt = 0;
    private String name;
    private int index;
    private Season(String theName){
        this.name = theName;
        this.index = Season.cpt++;
    }
    public static final Season winter = new Season("winter");
    public static final Season spring = new Season("spring");
    public static final Season summer = new Season("summer");
    public static final Season autumn = new Season("autumn");
    ... idem ...
}
```

Université Lille 1 - Licence Informatique	Programmation Orientée Objet	21	Université Lille 1 - Licence Informatique	Programmation Orientée Objet	22
static ○○○○○○○○○○	Types énumérés ○○○○○○○○	« Outils » de développement ○○○○○○○○○○○○○○○○○○	static ○○○○○○○○○○	Types énumérés ○○○○○○○○	« Outils » de développement ○○○○○○○○○○○○○○○○○○

## Ce sont des classes...

On peut donc ajouter des attributs, méthodes, constructeurs (privés !)...

```
public enum Day {
    monday(true), tuesday(true), wednesday(true),    // constantes du type énuméré
    thursday(true), friday(true), saturday(false), sunday(false);

    private final boolean working;                    // attribut ajouté

    private Day(boolean value) {                      // constructeur ajouté
        this.working = value;
    }

    public boolean isWorkingDay() {                   // méthode ajoutée
        return this.working;
    }
}

// utilisation
for(Day d : Day.values()) {
    System.out.println(d.name()+" vaut "+d.isWorkingDay());
}
```

## « Outils » de développement

static, enum, outils dev

Programmation Orientée Objet

Jean-Christophe Routier  
Licence mention Informatique  
Université Lille 1



## javac et java

- JAVA est un langage compilé

voir variable système PATH

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 lib;./truc/classes;%CLASSPATH% ...

(LINUX) : java/javac -classpath lib:./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 aussi être décomposé en « sous-paquetages »
- le nom complet de la classe *NomClasse* du sous-paquetage *souspackage* du package *nompaketage* est :

*nompaketage.souspackage.NomClasse*

notation UML : *nompaketage::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 :  

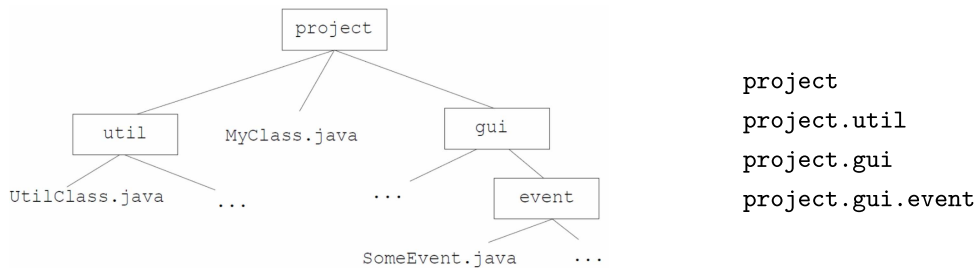
```
package nompaketage;
```

 ou 

```
package nompaketage.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 ?



à partir de la **racine des paquetages** :

```
javac project/*.java          javac project/util/*.java
```

et les fichiers .class sont placés dans une hiérarchie de répertoires copiant celle des paquetages/sources

```
java nompaketage.souspackage.NomClasse [args]
```

et il faut que le répertoire racine du répertoire nompaketage soit dans le CLASSPATH

## 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.
- cChaque sous-paquetage est placé dans un sous-répertoire (de même nom).

## 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).
- Il existe **toujours** un paquetage par défaut : le paquetage "anonyme". Toutes les classes qui ne déclarent aucun paquetage lui appartiennent.

### Règle

Il faut toujours **toujours** créer un paquetage

- "Officialisation" de l'existence du paquetage
- Permettre une réutilisation sans craindre l'ambiguïté de nom.
- Permettre la diffusion des classes, utilisation dans autres contextes.



## JavaDoc

*SomeClass.java* → *SomeClass.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 -d ../javadoc
```

```
/** description de la classe
 * @author <a href=mailto:bilbo@theshire.me>Bilbo Baggins</a>
 * @version 0.0.0.0.1
 */
public class JavaDocExample {
    /** commentaire attribut */
    public int i;
    /** ... */
    public void f(String s,Timoleon t) {}
    /** commentaire sur la methode avec <em>tags html</em>
     * sur plusieurs lignes aussi
     * @param o commentaire paramètre
     * @return commentaire valeur de retour
     * @exception java.lang.IllegalArgumentException commentaire exception
     * @see #f(String,Timoleon)
     */
    public String someMethod(Order o) throws IllegalArgumentException {
        return(o.getId());
    }
} // JavaDocExample
```

## Tests

### Règle

Un code non testé n'a aucune valeur.

### Corollaire

Tout code doit être testé

- **test unitaire**  
Tester les différentes parties d'un programme indépendamment les unes des autres.
- **test de non régression**  
Vérifier que le nouveau code ajouté ne corrompt pas les codes précédents : les tests précédemment réussis doivent encore l'être.

## Mise en œuvre

- Nous utiliserons le framework JUnit.
- Utilisation d'**assertions**
- Voir document accompagnant le TP 4.

```
package ...;
public class Box {
    /** creates a box with given initial weight
     * @param weight initial weight of this box
     */
    public Box(int weight) {
        this.weight = weight;
    }
    /** weight of the box */
    private int weight;
    /** @return this box's weight */
    public int getWeight() {
        return this.weight;
    }
}

java -jar test-1.7.jar BoxTest

...
public class BoxTest {

    @Test
    public void testBoxCreation() {
        Box someBox = new Box(10);
        assertNotNull(someBox);
    }

    @Test
    public void testGetWeight() {
        Box someBox = new Box(10);
        assertEquals(10,someBox.getWeight());
    }
    ...
}
```

Un robot peut porter une caisse à la fois d'un poids maximal défini à la construction du robot. Initialement un robot ne porte pas de caisse. S'il porte déjà une caisse il ne peut en prendre une autre.

```
import ...;

public class RobotTest {
    @Test
    public void RobotCanTakeOnlyOneBox {
        Robot robbie = new Robot(15);
        Box b1 = new Box(10);
        robbie.takeBox(b1);
        // b1 est bien la caisse portée
        assertEquals(b1, robbie.getCarriedBox());
        Box b2 = new Box(5);
        // exécution de la méthode testée
        robbie.takeBox(b2);
        // la caisse portée est toujours b1
        assertEquals(b1, robbie.getCarriedBox());
    }
}
```

Robot
...
+ Robot(int) + isCarryingABox() : boolean + takeBox(b : Box) + getCarriedBox() : Box

## Méthodologie

Travailler une méthode à la fois :

- 1 définir la signature de la méthode,
- 2 écrire la javadoc de la méthode,
- 3 écrire les tests qui permettront de contrôler que le code produit pour la méthode est correct,
- 4 coder la méthode,
- 5 exécuter les tests définis à l'étape 3, en vérifiant la non régression,
- 6 si les tests sont réussis passer à la méthode suivante (étape 1) sinon recommencer à l'étape 4.

**Il ne s'agit pas de travailler plus, mais d'être plus efficace.**

## Test Driven Development

## Archives : jar

voir outil système tar

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

```
jar ctXu[vfmOM] [nom-jar] [nom-manifest] [-C rép] fichiers ...
```

- c** création
- x** extraction
- t** afficher “**t**able”
- u** mettre à jour (**u**ppdate)
- v** “**v**erbose” : bavard
- f** spécifier le nom du fichier d’archives
- m** inclure le **m**anifeste
- etc.**

```
jar cf archive.jar Class1.class Class2.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

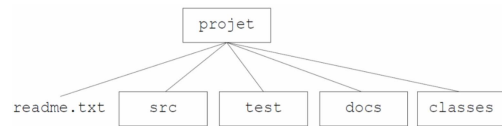
### Utilisation des classes contenues dans une archive sans extraction :

- mettre le fichier jar dans le CLASSPATH.

```
export CLASSPATH=$CLASSPATH:/home/java/jars/paquetage.jar
OU java -classpath $CLASSPATH:/home/java/jars/paquetage.jar ...
```

## organisation les fichiers

Pour chaque projet, créer l'arborescence :



**projet** répertoire racine

**src** racine de l'arborescence des  
paquetages avec sources .java

**test** les tests qui valident le code

**docs** la javadoc générée

**classes** les .class générés

+ ... (bibliothèques, images,  
etc.)

```
.../projet/src> javac -d ../classes *.java package1/*.java etc.
```

```
.../projet/src> javadoc -d ../docs .
```

```
.../projet/classes> jar cvfm ../project.jar themanifest .
```

```
.../projet> jar uvf project.jar src docs
```