Égalité, final, static, surcharge et documentation

Arnaud Labourel arnaud.labourel@univ-amu.fr

Section 1

L'égalité

Égalité de type primitif

Pour tester l'égalité de types primitifs *entiers* (`byte`, `short`, `int`, `long`, `char` et `boolean`), il suffit d'utiliser ==.

Pour tester l'égalité de types primitifs *flottants* (`float`, `double`), il faut faire attention aux erreurs d'approximation.

Bonne comparaison entre 2 doubles `d1` et `d2`

Fixer un ϵ petit et vérifier que $|d1 - d2| < \epsilon$.

Exemple :

```java
double d1 = 0.3;
double d2 = 0.1 + 0.1 + 0.1;
double epsilon = 0.000001;
d1 == d2// false : mauvaise comparaison
Math.abs(d1 - d2) < epsilon;// true : bonne comparaison
```

Égalités d'objets ?

Question

Quand peut-on dire que 2 objets sont égaux ?
Égalité de référence (même objet) ou de valeur (même état) ?

Égalité de référence

Les deux références sont identiques et pointent vers le même objet :
Égalité testée par l'opérateur ==

Égalité de valeurs

Les objets des deux références sont considérés équivalents :
Égalité testée par la méthode `equals` (définie dans la classe de l'objet appelant la méthode)

equals vs ==

```
String str1 = new String("Le Seigneur des Anneaux");
String str2 = new String("Le Seigneur des Anneaux");
```

- 2 références différentes sur 2 objets différents `str1 == str2` ⇒ `false`
- les deux objets référencés sont équivalents `str1.equals(str2)` ⇒ `true`

La classe `String` définit une méthode `equals` :

```
public boolean equals(Object o) { ... }
```

Un equals pour chaque classe

`equals` doit être (re)définie et adaptée pour chaque classe
(par défaut, elle se comporte comme ==)

Code méthode equals

```
public class Item {
  public boolean equals(Object o) {
    if (o instanceof Item) {
      // teste si o est un Item
      Item other = (Item) o;
      // cast o en Item afin d'accéder
      // aux attributs d'Item
      return (this.price == other.price) &&
             (this.id.equals(other.id));
    }
    return false;
  }
}
```

Explication code equals

1. La méthode `equals` compare l'objet courant `this` avec un `Object o` (et pas un `Item`).
2. `o` n'est pas forcément une instance d'`Item`. C'est pour cela, qu'on teste si `o` est une instance de `Item` avec `o instanceof Item` qui renvoie `true` si `o` est un `Item` et `false` sinon.
3. Si `o` n'est pas un `Item` alors `equals` retourne `false`.
4. Si `o` est un `Item`, on a besoin d'accéder à son `id` et son `price` pour pouvoir les comparer avec ceux de `this`. C'est pour cela qu'on fait un transtypage (*cast*) de `o` en `Item`.
5. Le transtypage est obligatoire, car une variable ou argument de type `Object` n'a pas d'attributs `id` ou `price`.

Section 2

Mot-clé `final`

Mot-clé `final` pour les attributs

Mot-clé `final` dans la déclaration d'un attribut : interdit la modification de la valeur de l'attribut (affectation) après la construction de l'objet.

```
public class Integer {
  public final int value;
  public Integer(int value) {
    this.value = value;
  }
}
```

- Un attribut `final` doit être initialisé après la construction de l'instance
- La valeur de l'attribut ne peut plus être modifiée ensuite (évite les effets de bord et permet de donner l'accès en lecture de l'attribut).

Utilisation mot-clé `final`

```java
public class Integer {
  public final int value;
  public Integer(int value) {
    this.value = value;
  }
}
```

```java
Integer i = new Integer(2);
i.value = 1; // Erreur à la compilation
```

`final` interdit l'affectation en dehors du constructeur et de l'initialisation par défaut.

Pourquoi `final` ?

- permet de donner l'accès à l'attribut sans crainte de modification (exemple : `length` des tableaux)
- permet de créer des objets immuable/immutable : objet dont l'état ne peut changer après leur construction :
  - évite la création d'effets de bord qui sont source d'erreurs en programmation
  - utile pour les petits objets (peu d'attributs : et donc facile de construire nouvelles instances)
  - utile pour les objets n'ayant pas vocation à changer d'état (objet contenant des données permettant un transfert d'information : `record`).

Exemple

Les `String` en Java sont immuables.

Section 3

Mot-clé `static`

Définition de π

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

Bonne pratique

Il faut nommer les constantes (surtout si elles apparaissent plusieurs fois dans le code) !
3.14 $\rightarrow$ pi

π en attribut ?

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

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

Attributs de classe

Solution : définir un attribut de classe (lié à la classe toute entière plutôt qu'à chaque instance).

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.
- Ils sont définis à l'aide du mot-clé `static`

Utilisation du mot-clé `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() {
    return 2 * Disc.pi * this.radius;
  }
  public double surface() {
    return Disc.pi * this.radius * this.radius;
  }
```

static, public et private

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

Sens de public et private

`StaticExample.compteur` n'est visible qu'à l'intérieur de la classe `StaticExample` :

- dans des méthodes/constructeurs de la classe
- pour l'initialisation d'autres attributs

$\Rightarrow$ attribut de classe (privé) partagé par les instances de la classe

`StaticExample.pi` est visible partout

static : attributs

static est un mot-clé à utiliser avec parcimonie et pertinence

avec final : définition de constantes

```
public class ConstantExample {
  public static final float PI = 3.141592f;
  public static final String BEST_MOVIE_TITLE = "Matrix";
}
```

- le qualificatif final signifie qu'une fois initialisée la valeur ne peut plus être modifiée, exemple : Double.MAX_VALUE, Math.PI, Boolean.TRUE, ...
- convention de nommage : les identifiants des constantes sont en majuscules avec _ pour séparer les mots (SNAKE_CASE).

Exemple d'utilisation

```
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 Catalog catalog;
  private String id;
  public Order(Client client, Catalogue cata) {
    this.client = client;
    this.catalogue = cata;
    this.id = Order.ORDER_ID_PREFIX + Order.counter++;
  }
  public String getId() { return this.id; }
}
```

Exemple d'utilisation

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

o1 : Order	
client	c
catalog	k
id	"order#1"

o2 : Order	
client	c
catalog	k
id	"order#2"

Exemple de static

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. The encoding used in the conversion from characters to bytes is equivalent to `Console.charset()` if the Console exists, `Charset.defaultCharset()` otherwise.

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.

Méthode sur les doubles

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 : `Math`

Utilisation ? = appel/invocation ⇒ il faut un objet

```
Math math1 = new Math();
math1.sin(45);
math1.sin(60);
Math math2 = new Math();
math2.sin(60);
```

Intérêt des objets `math1`, `math2` ?

Méthodes de classe

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

Appel/Invocation : pas besoin d'instance (juste le nom de la classe)

```java
StaticExample.staticMethod()
```

Important

Pas d'instance pour appeler la méthode donc `this` n'a aucun sens dans le corps d'une méthode statique.

Mot-clé `static` pour les méthodes

L'usage de `static` doit être limité et justifié à priori quasiment jamais car « pas objet ».

⇒ pratique réservée pour des méthodes dites « utilitaires » = fonctions

fonctions = méthodes de classe (`static`) dont le traitement ne dépend pas de l'état d'un objet

```
public class Math {
  public static double sin(double x) { ... }
  public static double sqrt(double x) { ... }
  public static int max(int a, int b) { ... }
}
```

Intérêt : éviter la création d'objet « jetable ».

⇒ on retrouve la notion de fonction comme en Python

Méthode `main`

Cas particulier, la méthode `main`, sa signature doit rigoureusement être

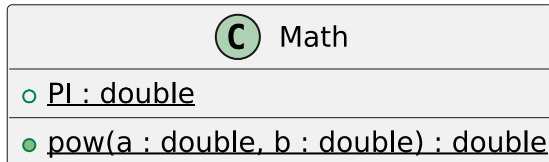
```
public class SomeClass {

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

Méthode appelée en utilisant la commande `java` avec comme argument `SomeClass` puis potentiellement d'autres arguments qui seront les valeurs de `args[]`.

```
java SomeClass arg0 arg1 ...
```

Méthode `static` dans les diagrammes



Règles

Membres (attributs ou méthode) de classe soulignés
Exemple : <u>staticMember</u>

Section 4

Surcharge de méthode/constructeurs

Méthodes ayant le même nom

Dans une classe, plusieurs méthodes peuvent avoir le même nom.

C'est ce qu'on appelle la surcharge de méthode.

Il est par contre nécessaire que la séquence dans l'ordre des types des arguments soit différente pour chaque méthode ayant le même nom.

La méthode est choisie par le compilateur de la façon suivante :

- Le nombre de paramètres doit correspondre
- Les affectations des paramètres doivent être valides

Exemple de surcharge de méthode

```java
public class DataArtist {

    public void draw(String s) {
        /* ... */
    }
    public void draw(int i) {
        /* ... */
    }
    public void draw(double f) {
        /* ... */
    }
    public void draw(int i, double f) {
        /* ... */
    }
}
```

Exemple de surcharge de méthode

```java
class Adder {
  public static int add(int intVal1, int intVal2) {
    System.out.println("integer");
    return intVal1 + intVal2;
  }

  public static double add(double doubleVal1,
                           double doubleVal2) {
    System.out.println("double");
    return doubleVal1 + doubleVal2;
  }
}
```

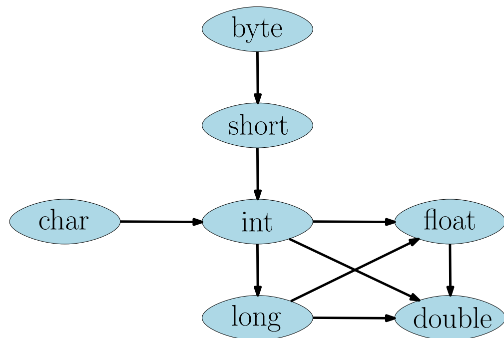
Exemple de surcharge de méthode

```java
int intValue = 1;
double doubleValue = 2.2;
double result1 = Adder.add(doubleValue, doubleValue);
// → double

int result3 = Adder.add(intValue, intValue);
// → int
```

Promotion pour les types primitifs

Si les types des arguments ne correspondent pas aux types des paramètres, les types des arguments sont promus en suivant les flèches:




```
double result2 = Adder.add(intValue, doubleValue);
// → double
```

Plusieurs constructeurs

C'est exactement les mêmes règles qui s'appliquent pour les constructeurs d'une classe.

```java
public class Point{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Point() {
        this(0, 0);
    // this appelle le constructeur de la classe.
    }
    public Point(Point p) {
        this(p.x, p.y);
    }
```

Section 5

Commentaires et documentation

Commentaires inutiles

```java
String get(String[] source, int index) {
    // Teste si l'index est dans les limites du tableau.
    if (index < 0 || index >= source.length)
        return null;
    return source[index];
}
```

Si un commentaire semble nécessaire, le remplacer par une méthode :

```java
boolean indexIsInBounds(String[] source, int index) {
    return index >= 0 && index < source.length;
}
String get(String[] source, int index) {
    if (!indexIsInBounds(source, index)) return null;
    return source[index];
}
```

Commentaires inutiles

Les commentaires se désynchronisent souvent du code :

```
/* doit toujours retourner true. */
boolean isAvailable() {
    return true;
}
```

risque de devenir un jour :

```
/* doit toujours retourner true. */
boolean isAvailable() {
    return false;
}
```

Commentaires inutiles = répétition

Commentaires inutiles

Des commentaires qui peuvent sembler utiles :

```
/* une méthode qui retourne que les carrés : */
List<Rectangle> get(List<Rectangle> list) {
    /* le résultat sera stocké dans cette liste : */
    List<Rectangle> list2 = new ArrayList<Rectangle>();
    for (Rectangle x : list)
        if (x.w == x.h /* un carré ? */)
            list2.add(x);
    return list2;
}
class Rectangle {
    public int w; /* largeur */
    public int h; /* hauteur */
}
```

Commentaires inutiles

On peut se passer de commentaire en rajoutant une méthode et en nommant correctement les éléments du code.

```
List<Rectangle> findSquares(List<Rectangle> rectangles) {
    List<Rectangle> squares = new ArrayList<Rectangle>();
    for (Rectangle rectangle : rectangles)
        if (rectangle.isSquare())
            squares.add(rectangle);
    return squares;
}
class Rectangle {
    private int width, height;
    boolean isSquare() {
        return width == height;
    }
}
```

Commentaires **utiles**

Des commentaires pour décrire les tâches à réaliser peuvent être utiles

```
void processElement(Stack<Formula> stack,
                    String element) {
    // TODO : prendre en compte les signes '-' et '/'
    switch (element) {
        case "+": processSum(stack); break;
        case "*": processProduct(stack); break;
        default : processInteger(stack, element);
        break;
    }
}
```

Commentaires **utiles**

Documentation ou spécification du comportement d'une méthode :

```
/**
 * Returns <tt>true</tt> if this list contains the
 * specified element. More formally, returns
 * <tt>true</tt> if and only if this list contains
 * at least one element <tt>e</tt> such that
 * <tt>(o==null ? e==null : o.equals(e))</tt>.
 *
 * @param o element whose presence in this list is
 * to be tested
 * @return <tt>true</tt> if this list
 * contains the specified element
 */
public boolean contains(Object o) {
    return indexOf(o) >= 0;
```

JavaDoc

JavaDoc permet de générer automatiquement une documentation du code à partir de commentaires associés aux classes, méthodes, propriétés, ...

La documentation contient :

- Une description des membres : attributs et méthodes (publics et protégés) des classes
- Une description des classes, interfaces, ...
- Des liens permettant de naviguer entre les classes
- Des informations sur les implémentations et extensions

Un bloc de commentaire Java commençant par /** deviendra un bloc de commentaire Javadoc qui sera inclus dans la documentation du code source.

JavaDoc

Tag	Description
@author	pour préciser l'auteur de la fonctionnalité
@deprecated	indique que l'attribut, la méthode ou la classe est dépréciée
@return	pour décrire la valeur de retour
{@code `literal`}	Formate `literal` en code
{@link `reference`}	permet de créer un lien

Pour générer la javadoc en IntelliJ

Tools → generate Javadoc

JavaDoc

```
/**
 * The {@code Byte} class wraps a value of primitive
 * type {@code byte} in an object.  An object of type
 * {@code Byte} contains a single field whose type is
 * {@code byte}.
 *
 * <p>In addition, this class provides several methods
 * for converting a {@code byte} to a {@code String}
 * and a {@code String} to a {@code byte}, as well as
 * other constants and methods useful when dealing
 * with a {@code byte}.
 *
 * @author  Nakul Saraiya
 * @author  Joseph D. Darcy
 */
```