Égalité, final, static et surcharge

Arnaud Labourel arnaud.labourel@univ-amu.fr

28 février 2022

Section 1

L'égalité d'objets

Deux égalités d'objets ?

Question

Quand peut-on dire que 2 objets sont égaux ?
Égalité d'objets ou de valeur ?

Égalité d'objet

Les deux références désignent le même objet :
Égalité testée par l'opérateur ==

Égalité de valeurs

Les objets des deux références sont équivalents :
Égalité testée par la méthode `equals`

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));
    }
    else {
      return false;
    }
  }
}
```

Explication code `equals`

- la méthode `equals` compare l'objet courant `this` avec un `Object o` (et pas un `Item`).
- `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.
- Si `o` n'est pas un `Item` alors `equals` retourne `false`.
- 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`.
- 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 après la construction de l'objet.

Exemple :

```
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`

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

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

```
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

```
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";
}
```

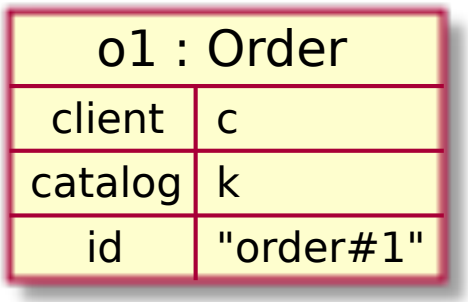
- le qualificatif final signifie qu'une fois initialisée la valeur ne peut plus être modifiée, exemple : Double.MAX_VALUE, Math.PI, Boolean.TRUE, ...
- convention de nommage : les identifiants des constantes sont en majuscules avec _ pour séparer les mots (SNAKE_CASE).

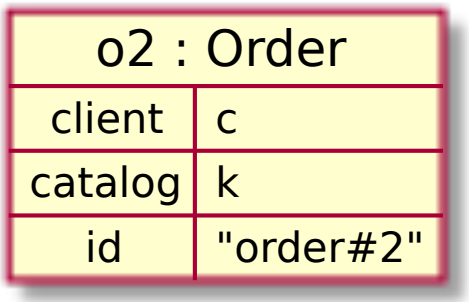
Exemple d'utilisation

```
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());
```





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 $\Rightarrow$ il faut un objet

```
Math math1 = new Math();
math1.sin(45);
math1.sin(60);
Math math2 = new Math();
trigo2.sin(60);
```

Intérêt des objets `math1`, `math2` ?

Méthodes de classe

```
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)

`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

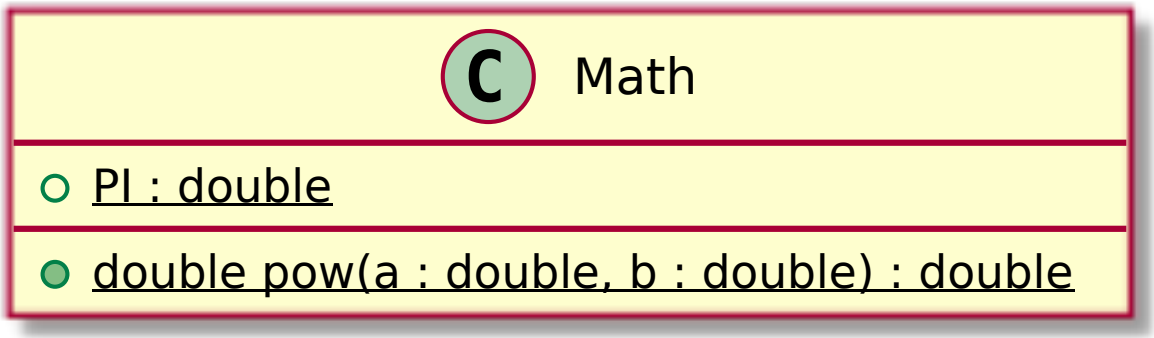
```
public class SomeClass {

  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 : staticMember