Programmation Orientée Objet – Java

Cours 3 : Création de classes

Viviane Pons

Master BIBS Université Paris-Saclay

Créer une classe

Comment créer nos propres classes ?

Dans ce cours, on va voir les étapes de création d'une classe en suivant étape par étape l'exemple suivant.

Où ? Les packages

Créer une classe

Comment créer nos propres classes ?

Dans ce cours, on va voir les étapes de création d'une classe en suivant étape par étape l'exemple suivant.

Où ? Les packages

On a vu que les classes de l'API Java étaient organisées en **package**.
En fait le vrai nom de la classe `Scanner` est :
`java.util.Scanner`
La classe `Scanner` appartient au package `java.util`. En effectuant un `import` on s'autorise simplement à oublier la partie du nom qui relève du package.
Il pourrait tout à fait exister une autre classe appelée `Scanner` dans un autre package : l'utilisation des packages évite les conflits de noms.

```
com.myorg.Scanner // exemple factice
```

Créer nos propres packages

Jusqu'à présent, nous avons toujours créé une **unique classe** (contenant le main) dans le "default package" (pas de package).
On va arrêter !

Créer nos propres packages

Jusqu'à présent, nous avons toujours créé une **unique classe** (contenant le main) dans le "default package" (pas de package).
On va arrêter !
Traditionnellement, le nom du package reflète la provenance de la classe (par exemple, avec le nom de l'entreprise) et ressemble à une url à l'envers. On imagine qu'on crée nos classes de façon unique dans **le grand univers qui contient toutes les classes Java**.
Pour nos classes à nous, on utilisera le nom de base suivant :
`fr.upsaclay.bibs`
Que l'on déclinera ensuite en fonction des packages créés.
`fr.upsaclay.bibs.monpackagebla`
`fr.upsaclay.bibs.encoreunpackage`

Démo

Création des packages

`fr.upsaclay.bibs`

`fr.upsaclay.bibs.rational`

avec Eclipse.

On remarque que la structure de package crée l'équivalent en arborescence de fichier.

Comment créer une classe ?

Le fichier

Dans le dossier du package correspondant, on va créé un **fichier par classe**

- le nom du fichier doit être le même que le nom de la classe
- le nom de la classe commence toujours par une majuscule `MaSuperJolieClasse`
- le package est indiqué en haut du fichier (et doit correspondre à l'emplacement de la classe)

```
package fr.upsaclay.bibs;

public class RationnalExample {
```

Démo : création des classes `RationalExample` dans `fr.upsaclay.bibs` et `Rational` dans `fr.upsaclay.bibs.rational`.

Public or not public ?

La présence (ou non) du *modificateur* `public` avant le mot `class` indique la **visibilité de la classe** :

- si on écrit `public` : la classe est visible (utilisable) par les classes en dehors du package
- si on écrit rien : la classe n'est visible que dans le package

Il existe d'autres modificateurs que l'on peut rajouter en plus de `public` avant le mot `class` : `abstract` et `final`, nous verrons leur signification quand nous traiterons de la question de **l'héritage**.

Champs, méthodes et constructeurs

Ajoutons des champs et un constructeur

```
public final int n;
public final int d;

public Rational(int n, int d) {
    this.n = n;
    this.d = d;
}
```

Visibilité (des champs et des méthodes) :

- `public` tout le monde peut voir le champ (et le modifier !)
- `protected` : les éléments du package ainsi que les classes héritées ont accès au champ
- rien : les éléments du package ont accès au champ
- `private` : seule la classe a accès au champ

Que signifie `final` pour un champ ?

Le champ est fixé à la création de l'objet et ne pourra plus être modifié

Le mot clé a aussi une signification pour les méthodes que l'on verra plus tard.

Le(s) constructeur(s)

- Une méthode spéciale pour construire l'objet
- En Java : porte le nom de la classe et n'a pas de valeur de retour
- Les mêmes paramètres de visibilité `public` / `protected` / `private` s'applique

Quel est son rôle ? Initialiser les champs (en particulier, les champs de type `final` et effectuer les vérifications / calculs initiaux nécessaires au fonctionnement de l'objet créé)

Illustration

```
public Rational(long n, long d) {
    if(d == 0) {
        throw new IllegalArgumentException();
    }

    long div = gcd(n, d);
    n = n / div;
    d = d / div;

    if(n > max_operand || n < min_operand || d > max_operar
        throw new IllegalArgumentException("Exceeds operanc
    }
    this.n = (int)n;
    this.d = (int)d;
}
```

Plusieurs constructeurs

En Java, il est possible de définir plusieurs fonctions portant le même nom si les paramètres (nombre / type) sont différents.

```
protected Rational(long n, long d, boolean check) {
    ...
}

public Rational(long n, long d) {
    this(n, d, true);
}



public Rational(long n) {
    this(n,1, false);
}

public Rational() {
    this(0);
}
```

L'appel à un autre constructeur de la classe se fait par `this(...)`

Créons des méthodes

```
/**
* Adds rationals
* @param r2 another rational object
* @return the Rational representing the sum of object and
*/
public Rational add(Rational r2) {
    return new Rational((long)n*r2.d + (long)r2.n * d, (lon
}
```

Portées des variables

- Une variable existe à l'intérieur de son bloc `{ ... }` (classe, méthode, bloc `if` / `for`
- On peut utiliser directement le nom d'un champ ou d'une méthode à la l'intérieur de la classe (statique ou non)
- Si conflit de nom (avec variable locale) on peut écrire `this.champ`

Méthodes spéciales

Représentation sous forme de chaîne de caractère

```
@Override
public String toString() {
    if(d != 1) {
        return n + "/" + d;
    } else {
        return String.valueOf(n);
    }
}
```

Égalité d'objets

```
/**
* Return whether r2 is equal to object as rationals
* @param r2 a rational
* @return true if this is equal to self
*/
public boolean equals(Rational r2) {
    return (r2.n == n && r2.d == d);
}


@Override
public boolean equals(Object obj) {
    if(obj == null) {
        return false;
    }
    if(!Rational.class.isAssignableFrom(obj.getClass())) {
        return false;
    }
    return this.equals((Rational)obj);
```

Hachons

```
@Override
public int hashCode() {
    return toString().hashCode();
}
```

La valeur de hachage, ou “hash code” est une clé associée à l'objet.

- Quand deux objets sont égaux, ils doivent avoir la même clé
- Quand deux objets sont différents, ils doivent avoir autant que possible des clés différentes (mais le nombre de clés est fini et le nombre d'objets infini. . .)

Cette valeur est utilisée par les structures de données type `Set` et `Map`.

Overide quoi ?

L'annotation `@Overide` est utilisée pour indiquer au compilateur que la méthode redéfinit une méthode définie précédemment (voir le prochain chapitre sur l'héritage). Les méthodes `toString` `equals` et `hashCode` sont héritées de la classe `Object` et utilisent l'adresse mémoire par défaut.

!! Quand on redéfinit `equals` on doit redéfinir `hashCode` !!

Implémenter des interfaces

Pourquoi faire ?

Implémenter des interfaces

Pourquoi faire ?

Pour utiliser des fonctionalités génériques. Par exemple, car je voudrais trier des listes de rationnels.

Exemple : interface Comparable

Voir la Javadoc

Exemple : interface Comparable

Voir la Javadoc

```
public class Rational implements Comparable<Rational> {
@Override
public int compareTo(Rational r2) {
    if(equals(r2)) {
        return 0;
    }
    return (doubleValue() < r2.doubleValue())? -1 : 1;
}
```

Un peu de static

Quelques méthodes

```java
/**
* Value of given string as Rational
* @param s a String value
* @return
*/
public static Rational valueOf(String s) {
    String[] parts = s.split("/");
    if (parts.length == 1) {
        return Rational.valueOf(Integer.valueOf(s));
    } else if (parts.length == 2) {
        return new Rational(Integer.valueOf(parts[0]), Inte
    } else {
        throw new NumberFormatException("For input string:
    }
}
```

```java
/**
* Sums the given rationals
* @param args a list of Rational
* @return the sum as a Rational
*/
public static Rational sum(Rational... args) {
    Rational s = new Rational();
    for(Rational r : args) {
        s = s.add(r);
    }
    return s;
}
```

Une variable static privée ?

```
/*
* the minimal Rational that has been instantiated
*/
private static Rational minimal_created;

/*
* the maximal Rational that has been instantiated
*/
private static Rational maximal_created;
```

```
protected Rational(long n, long d, boolean check) {
    ...
    if (minimal_created == null || this.compareTo(minimal_c
        minimal_created = this;
    }
    if (maximal_created == null || this.compareTo(maximal_c
        maximal_created = this;
    }
}

public static Rational getMinimalCreated() {
    return minimal_created;
}

public static Rational getMaximalCreated() {
    return maximal_created;
}
```

Les Tests

Pour l'instant, on a testé notre programme en faisant des essais dans la fonction `main`. Mais "dans la vraie vie", la fonction `main` sert à écrire le coeur de l'application : toutes les fonctionnalités et les cas de figure ne sont pas appelés.

On veut créer des tests de façon séparée, indépendants d'une application particulière.

JUnit : une application de test pour Java

Un **test unitaire** est une procédure permettant de tester le bon fonctionnement une partie spécifique d'un programme.
JUnit est un "framework" (un "cadre", une "infrastructure", un "schéma") permettant de réaliser simplement des tests unitaire en Java.

Mise en oeuvre

On crée un package
`fr.upsaclay.bibs.rational.test`
Dans ce package on crée une classe `RationalTest` en faisant (avec Eclipse) “New » JUnit Test Case”

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import fr.upsaclay.bibs.rational.Rational;

class RationalTest {
```

On crée des méthodes de test

```
@Test
public final void testRational() {
    assertEquals(new Rational(1,2), new Rational(2,4));
    assertEquals(new Rational(2), new Rational(4,2));
    assertEquals(new Rational(-2,4), new Rational(2,-4));
    ...
}
```

On lance les tests en faisant “Run as » JUnit Test”

Junit nous indique :

- quels tests passent et quels tests ne passent pas
- la partie du code qui est “couverte” par les tests

Quels tests ?

Pour faire les tests, on utilise les méthodes statiques de la classe Assertions

- `assertEquals`
- `assertNotEquals`
- `assertThrows`

Que faut-il tester ?

Typiquement, on crée une méthode de test par méthode de la classe testée et on teste . . . tout ce à quoi on pense.
Les tests unitaires ne pourront **jamais vous assurer** que votre programme fonctionne. Ils peuvent être complétés par des **tests fonctionnels** ou de la **vérification formelle**.
Cependant, ils permettent de détecter de nombreuses erreurs et facilitent la maintenance du code.