Enumérations et classes imbriquées

Pratique de la programmation orientée-objet
Michel Schinz – 2015-03-09

Enumérations

Cartes à jouer

On désire modéliser les cartes à jouer d'un jeu de 52 cartes. Chacune d'entre-elles est caractérisée par sa couleur – cœurs, carreaux, trèfles ou piques – et sa valeur – 2, 3, …, 10, valet, dame, roi, as.

```
public final class Card {
  private final ??? suit; // couleur
  private final ??? rank; // valeur
  public Card(??? suit, ??? rank) {
    this.suit = suit;
    this.rank = rank;
  }
}
```

Quel(s) type(s) utiliser pour la couleur et la valeur ?

Solution 1 : chaînes, entiers

Une première solution est d'utiliser le type `String` pour la couleur et le type `int` pour la valeur, et d'introduire des constantes pour les valeurs valides :

```
public final class Card {
  public final static String CLUBS ="CLUBS";
  // … idem pour HEARTS, SPADES et DIAMONDS
  public final static int JACK = 11;
  // … idem pour QUEEN, KING et ACE
  private final String suit;
  private final int rank;
  public Card(String suit, int rank) { … }
}
```

Est-ce satisfaisant ? Si non, pourquoi ?

Solution 1 : chaînes, entiers

La solution basée sur les types `String` et `int` n'est pas satisfaisante, pour plusieurs raisons :

- ces types sont trop généraux et incluent énormément de valeurs invalides, p.ex. la chaîne `"bonjour"` pour la couleur, ou l'entier 40 pour la valeur,
- ces types ne sont pas assez parlants à la lecture du code,
- il n'est pas possible d'attacher des attributs ou des méthodes à ces types.

Solution 1 : chaînes, entiers

Il y a une infinité de valeurs de type `String`, mais seules quatre d'entre-elles sont des couleurs valides. De même, il y a 2^{32} valeurs de type `int`, mais seules treize d'entre-elles sont des valeurs de cartes valides…
Cela force à faire constamment des vérifications coûteuses et fastidieuses, p.ex. dans le constructeur de la classe `Card` :

```
public Card(String suit, int rank) {
  if (! (suit.equals(SPADES)
         || suit.equals(HEARTS)
         …)
    throw new IllegalArgumentException(…);
  …
}
```


Solution 2 : objets

Une solution plus propre et conforme aux principes de la programmation orientée-objets consiste à définir une classe pour les couleurs et une pour les valeurs, et des constantes :

```
public final class Suit {
  public final static Suit SPADES =
    new Suit("SPADES");
  // … idem pour HEARTS, CLUBS et DIAMONDS
  private final String name;
  private Suit(String name) {
    this.name = name;
  }
}
public final class Rank { … }
```

Question : pourquoi le constructeur est-il privé ?

Solution 2 : objets

Une fois les classes `Suit` et `Rank` définies de la sorte, le constructeur de `Card` devient plus simple car il n'a plus besoin de vérifier ses arguments, qui sont valides par construction (si on ignore `null`, en tout cas) !

```
public final class Card {
  private final Suit suit;
  private final Rank rank;
  public Card(Suit suit, Rank rank) {
    this.suit = suit;
    this.rank = rank;
  }
  // … autres méthodes
}
```

Solution 2 : objets

Autre avantage de cette solution orienté-objet : il est possible d'ajouter des méthodes ou des champs aux classes `Suit` et `Rank`. Exemple :

```
public final class Suit {
  // … comme avant
  public String frenchName() {
    if (this == SPADES)   return "piques";
    if (this == DIAMONDS) return "carreaux";
    if (this == CLUBS)    return "trèfles";
    assert this == HEARTS;
    return "cœurs";
  }
}
```


Solution 3 : énumérations

La solution orientée-objets est bonne, mais fastidieuse à mettre en œuvre, car elle demande l'écriture de beaucoup de code répétitif.
Pour éviter de devoir écrire ce code à chaque fois, Java offre la notion d'énumération.

Solution 3 : énumérations

Au moyen des énumérations, les types `Suit` et `Rank` peuvent se définir simplement ainsi :

```
// fichier Suit.java
public enum Suit {
  SPADES, DIAMONDS, CLUBS, HEARTS
};
// fichier Rank.java
public enum Rank {
  TWO, THREE, FOUR, FIVE, SIX, SEVEN,
  EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE
};
```

Java se charge de traduire ces énumérations en classes similaire aux nôtres, mais encore plus complètes.

Traduction de enum

Une énumération Java est traduite en une sous-classe de `java.lang.Enum` et chaque élément de l'énumération est traduit en une instance de cette classe.
Ces instances sont des champs statiques immuables de la classe de l'énumération, exactement comme dans notre version.
Java définit de plus quelques méthodes utiles sur les énumérations et leurs éléments, examinées ci-après.

Méthode values

La méthode statique `values`, définie sur l'énumération elle-même, retourne un tableau contenant tous les éléments de l'énumération, dans l'ordre de déclaration.
Par exemple, pour l'énumération `Suit` suivante :

```
public enum Suit {
  SPADES, DIAMONDS, CLUBS, HEARTS
};
```

l'appel `Suit.values()` retourne un tableau de taille 4 contenant (l'objet) `SPADES` à la position 0, (l'objet) `DIAMONDS` à la position 1, et ainsi de suite.

Méthode ordinal

La méthode `ordinal` retourne un entier indiquant la position de l'élément dans l'énumération (à partir de zéro).
Par exemple, étant donnée l'énumération `Suit` suivante :

```
public enum Suit {
  SPADES, DIAMONDS, CLUBS, HEARTS
};
```

`SPADES.ordinal()` retourne 0, `DIAMONDS.ordinal()` retourne 1, et ainsi de suite.
Les méthodes `ordinal` et `values` sont donc complémentaires.

Autres méthodes

Les éléments des énumérations sont également équipés d'autres méthodes, parmi lesquelles :

- une redéfinition de `toString` qui retourne le nom de l'élément, p.ex. `"SPADES"`,
- une redéfinition de `compareTo` de l'interface `Comparable` – automatiquement implémentée par toutes les énumérations – qui compare les éléments de l'énumération par ordre de définition.

Enoncé switch

Les éléments d'une énumération sont utilisables avec l'énoncé `switch`. Par exemple :

```
public final class Card {
  private final Suit suit;
  // … autres champs, constructeur, etc.
  public String frenchSuitName() {
    switch (suit) {
      case SPADES:   return "piques";
      case DIAMONDS: return "carreaux";
      case CLUBS:    return "trèfles";
      case HEARTS:   return "cœurs";
      default:       throw new Error();
    }
  } }
```

obligatoire en raison d'une limitation de Java

Ajout de membres

Il est possible d'ajouter des membres – attributs ou méthodes – aux énumérations, p.ex. pour placer la méthode `frenchName` dans `Suit` plutôt que dans `Card` :

```
public enum Suit {
  SPADES, DIAMONDS, CLUBS, HEARTS;
  public String frenchName() {
    switch (this) {
      case SPADES:   return "piques";
      case DIAMONDS: return "carreaux";
      case CLUBS:    return "trèfles";
      case HEARTS:   return "cœurs";
      default:       throw new Error();
    }
  } }
```


Constructeur

Il est aussi possible de définir un constructeur pour l'énumération, mais il ne peut être public. Exemple :

```
public enum Suit {
  SPADES("piques"), DIAMONDS("carreaux"),
  CLUBS("trèfles"), HEARTS("cœurs");

  private final String frenchName;
  public String frenchName() {
    return frenchName;
  }
  private Suit(String frenchName) {
    this.frenchName = frenchName;
  }
}
```


Enumérations imbriquées

Les énumérations `Suit` et `Rank` ne sont pas utiles en dehors de la classe `Card`. Il est préférable de les imbriquer dans celle-ci, ce qui est tout à fait possible :

```
public final class Card {
  public enum Suit { SPADES, …, HEARTS };
  public enum Rank { TWO, …, ACE };
  private final Suit suit;
  private final Rank rank;
  public Card(Suit suit, Rank rank) { … };
}
```

Lorsqu'une énumération est ainsi imbriquée, elle l'est automatiquement de manière statique. Le modificateur `static` est autorisé mais facultatif – car implicite.

EnumSet, EnumMap

Les éléments d'énumérations peuvent bien entendu être placés dans des ensembles ou des tables de hachages standard, de type `HashSet/Map` ou `TreeSet/Map`. Toutefois, l'API Java fournit également des mises en œuvre plus efficaces, à savoir :

- `EnumSet` pour les ensembles dont les éléments proviennent d'une énumération,
- `EnumMap` pour les tables associatives dont les clefs proviennent d'une énumération.

Classes imbriquées

Intervalle d'entiers

Admettons que l'on désire écrire une classe immuable représentant un intervalle d'entiers – p.ex. [5,7] – itérable.

```
public final class IntInterval
    implements Iterable<Integer> {
  private final int b, e; // begin, end
  public IntInterval(int b, int e)
    { this.b = b; this.e = e; }
  public int size()
    { return Math.max(0, e - b + 1); }
  @Override
  public Iterator<Integer> iterator()
    { return new IIIterator(b, e); }
  // … définition de la classe IIIterator.
}
```


Itérateur

Naturellement, la classe de l'itérateur peut être définie comme une classe imbriquée statique et privée :

```
private static final class IIIterator
    implements Iterator<Integer> {
  private int n;       // next
  private final int e; // end
  public IIIterator(int b, int e)
    { this.n = b; this.e = e; }
  @Override public boolean hasNext()
    { return n <= e; }
  @Override public Integer next() {
    if (!hasNext()) throw new NoSuchElem…;
    return n++;
  } } // remove lève UnsupportedOperatioEx.
```


Accès aux bornes

Etant donné que la classe de l'itérateur est imbriquée statiquement dans celle de l'intervalle :

```
public final class IntInterval
    implements Iterable<Integer> {
  private final int b, e;
  // … comme avant
  private static class IIIterator
      implements Iterator<Integer> {
    public IIIterator(int b, int e) { … }
    // … comme avant
  }
}
```

pourquoi ne peut-elle accéder directement aux bornes b et et e de la classe englobante – `IntInterval` ?

Accès aux membres

La classe de l'itérateur ne peut accéder aux bornes de l'intervalle car elle est imbriquée *statiquement* dans la classe de l'intervalle, et n'a donc accès qu'aux membres statiques de sa classe englobante…
Peut-on l'imbriquer de manière non statique ? Oui, Java autorise les classes imbriquées non statiques, appelées *inner classes* en anglais – traduit ici par **classes intérieures**.
Contrairement à une classe imbriquée statique, une classe intérieure a accès à *tous* les membres de sa classe englobante – statiques ou non.

Itérateur intérieur

L'itérateur peut donc se définir ainsi, sans son constructeur devenu inutile :

```
public final class IntInterval
    implements Iterable<Integer> {
  private final int b, e;
  // … comme avant, sauf iterator()
  private final class IIIterator
      implements Iterator<Integer> {
    private int n = b; // next
    @Override public boolean hasNext()
      { return n <= e; }
    // … next et remove comme avant
  }
}
```


Itérateur intérieur

La classe intérieure `IIIterator` n'ayant plus de constructeur explicite, la méthode `iterator` de la classe `IntInterval` s'en trouve simplifiée :

```
public final class IntInterval
    implements Iterable<Integer> {
  // … comme avant
  @Override
  public Iterator<Integer> iterator() {
    return new IIIterator();
  }
}
```

plus d'arguments…

Imbrication (non) statique

Les classes intérieures étant « plus puissantes » que les classes imbriquées statiques – dans le sens où elles ont accès à tous les membres de la classe englobante – pourquoi ne pas définir que des classes intérieures ?
Parce que l'accès aux membres non statiques de la classe englobante implique qu'une instance de cette classe soit à disposition…

Classes intérieures et new

On l'a vu avec les bâtisseurs, une instance d'une classe imbriquée *statique* se crée aussi facilement qu'une instance de classe de niveau supérieur, p.ex.

```
new Date.Builder();
```

Par contre, pour créer une instance d'une classe intérieure, il faut disposer d'une instance de la classe englobante ! Par exemple, si la classe *intérieure* `IIIterator` était publique, on ne pourrait en créer une instance depuis l'extérieur de la classe `IntInterval` qu'en disposant d'une instance d'un tel intervalle :

```
IntInterval iv = new IntInterval(5, 7);
Iterator<Integer> i = new iv.IIIterator();
```


Classes intérieures et new

En examinant le code de la méthode `iterator` de `IntInterval` dans la version où `IIIterator` est une classe intérieure, on constate qu'aucune instance d'intervalle n'est utilisée explicitement dans l'énoncé `new` :

```
public Iterator<Integer> iterator() {
  return new IIIterator();
}
```

Pourquoi ? Parce que – comme d'habitude en Java – cette instance est implicitement `this` ! Le code ci-dessus est donc équivalent à :

```
public Iterator<Integer> iterator() {
  return new this.IIIterator();
}
```


this englobant

On l'a vu, une classe intérieure a accès aux membres de sa classe englobante, p.ex. pour l'attribut `b` de `IntInterval` :

```
public class IntInterval … {
  private final int b, e;
  private class IIIterator … {
    private int n = b;
```

Là aussi un `this` implicite se cache, mais il s'agit de celui de la classe englobante, qui peut être rendu explicite en nommant celle-ci, p.ex. :

```
public final class IntInterval … {
  private final int b, e;
  private final class IIIterator … {
    private int n = IntInterval.this.b;
```


Classe anonyme

La classe `IIIterator` est privée car elle ne doit pas être visible de l'extérieur. Elle constitue un détail de mise en œuvre de la classe `IntInterval`.

Dès lors, pourquoi la nommer ? Ne pourrait-on pas la définir à l'endroit où on l'utilise ?

Oui, cela est possible grâce aux **classes intérieures anonymes** de Java. De telles classes peuvent être définies directement dans l'énoncé `new` qui les crée.

Itérateur anonyme

La classe de l'itérateur peut se définir de manière anonyme directement dans la méthode `iterator` :

```
public final class IntInterval
    implements Iterable<Integer> {
  // … comme avant
  @Override
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      private int n = b; // next
      // … comme avant (ce qui était la
      // classe IIIterator)
    }
  }
}
```

classe anonyme implémentant l'interface `Iterator<Integer>`

Classes anonymes

Une classe anonyme se définit toujours dans un énoncé **new**, avec la syntaxe suivante :

new C (a_1, …) { … *attributs et/ou méthodes* … }

où C peut être

1. le nom d'une classe non finale – auquel cas la classe anonyme en hérite et les arguments a_1, … sont passés à son constructeur,
2. le nom d'une interface – auquel cas la classe anonyme hérite de `Objet` et implémente l'interface, et aucun argument ne doit figurer entre parenthèses.

Une classe anonyme ne peut avoir de constructeur.