Arité variable, énumérations

Théorie et pratique de la programmation
Michel Schinz – 2013-05-13

Arité variable

Arité des méthodes

Il est parfois utile d'offrir des méthodes ayant une **arité variable**, c'est-à-dire acceptant un nombre variable d'arguments.
Par exemple, on peut vouloir définir une méthode `sum` calculant la somme d'un nombre quelconque d'entiers qu'on puisse appeler ainsi :

```
sum();
sum(1);
sum(1, 2);
```

etc.

Comment faire ?

Utilisation de la surcharge

Une première idée serait d'utiliser la surcharge pour définir plusieurs versions de la méthode `sum` :

```
int sum() { return 0; }
int sum(int v1) { return v1; }
int sum(int v1, int v2) { return v1 + v2; }
```

… et ainsi de suite

Mais cette technique a bien entendu ses limites, puisqu'elle ne permet pas la définition d'une méthode acceptant un nombre réellement quelconque de valeurs.

Elle est de plus fastidieuse à mettre en œuvre...

Utilisation des collections

Une deuxième idée serait d'utiliser une collection, p.ex. une liste ou un tableau :

```
int sum(int[] vs) {
  int sum = 0;
  for (int v: vs)
    sum += v;
  return sum;
}
```

Cette solution fonctionne mais est lourde à l'usage car chaque appel implique la création d'un tableau :

```
sum(new int[]{ 1, 2, 3 });
```

Arité variable en Java

Pour résoudre ces problèmes, Java offre la possibilité de définir des méthodes à arité variable. Cela se fait en ajoutant trois points à la suite du type du dernier argument :

```
int sum(int... vs) {
  // ... calcule la somme de tous
  // les arguments
}
```

Cela fait, il devient possible d'appeler la méthode avec un nombre quelconque d'arguments, y compris 0:

```
sum();      // retourne 0
sum(1);     // retourne 1
sum(1, 2); // retourne 2
```

Arité variable en Java

Les méthodes à arité variable sont mises en œuvre au moyen de tableaux. Les arguments en nombre variable sont donc passés sous la forme d'un tableau et le paramètre correspondant a un type tableau.
Le corps de la méthode `sum` peut donc s'écrire exactement comme dans la version utilisant des tableaux explicites :

```
int sum(int... vs) {
  int sum = 0;
  for (int v: vs)
    sum += v;
  return sum;
}
```

(`vs` a le type `int[]`).

Arité variable en Java

Il est valide de passer directement un tableau contenant les arguments à une méthode à arité variable. Ainsi, les deux appels à `sum` ci-dessous sont équivalents :

```
sum(1, 2, 3);
int[] array = new int[]{ 1, 2, 3 };
sum(array);
```

Arité minimum

Dans certains cas, une méthode prend un nombre variable d'arguments mais ce nombre ne peut pas être inférieur à une valeur donnée – souvent 1.
Par exemple, une méthode qui calcule le minimum d'un nombre quelconque d'entiers doit recevoir au moins une valeur, faute de quoi le minimum n'est pas défini.
Dans un tel cas, les n arguments obligatoires peuvent être passés comme arguments normaux, tandis que les autres peuvent être passés dans un « argument variable ».

Arité minimum

Ainsi, une méthode calculant le minimum de n entiers, avec n > 0, se définit comme suit :

```
int min(int v1, int... vs) {
   // ???
}
```

Enumérations Java

Classe des dates (bis)

Rappel : dans la leçon précédente, nous avions écrit la classe ci-dessous pour modéliser les dates.

```
public final class GregorianDate {
  private final int year, month, day;
  public GregorianDate(int y,int m,int d) {
    this.year = y;
    this.month = m;
    this.day = d;
  }
  …
}
```

Que peut-on lui reprocher ?

Validité d'une date

Le constructeur de notre classe ne vérifie pas la validité des valeurs qu'on lui passe. Il faudrait donc l'augmenter ainsi :

```
public GregorianDate(int y, int m, int d) {
  if (! (1 <= m && m <= 12))
    throw new IllegalArgumentException(…);
  if (! (1 <= d && d <= daysInMonth(y, m)))
    throw new IllegalArgumentException(…);
  …
}
```

La règle déterminant la validité du jour est complexe. Mais celle pour le mois est triviale, ne peut-on pas la vérifier avant l'exécution ?

Mois énuméré

La totalité des mois valides peut facilement être énumérée : janvier, février, …, décembre. Il serait donc mieux de représenter le mois par l'une de ces valeurs que par un entier quelconque.
Java offre pour cela la notion d'**énumération**, introduite au moyen du mot-clef `enum` :

```
public final class GregorianDate {
   public enum Month { JANUARY, FEBRUARY,
                       MARCH, APRIL, MAY,
                       JUNE, JULY, AUGUST,
                       SEPTEMBER, OCTOBER,
                       NOVEMBER, DECEMBER };
   …
}
```

énumération

valeurs de l'énumération

Date avec mois énuméré

Le constructeur de `GregorianDate` n'a maintenant plus besoin de vérifier la validité du mois, puisque celui-ci ne peut qu'être l'une des 12 valeurs de l'énumération.

```
public final class GregorianDate {
  public enum Month { JANUARY, … };
  private final int year;
  private final Month month;
  private final int day;
  public GregorianDate(int y,Month m,int d){
    if (!(1 <= d && d <= daysInMonth(y, m)))
      throw new IllegalArgumentException(…);
    …
  }
}
```

Date avec mois énuméré

L'affirmation précédente n'est pas tout à fait correcte : comme nous le verrons, les valeurs d'énumération sont des objets, donc `null` peut être passé au constructeur... Comme celui-ci se contente de stocker le mois reçu, il peut être intéressant de vérifier immédiatement qu'il n'est pas nul, faute de quoi l'erreur risque de se manifester trop tard :

```
public final class GregorianDate {
  public GregorianDate(int y,Month m,int d){
    if (m == null)
      throw new NullPointerException(…);
    …
  }
}
```

Date avec mois énuméré

Les mois énumérés lèvent partiellement l'ambiguïté quant à la signification des différents arguments du constructeur de date. Ainsi, il est facile de savoir de quelle date on parle dans l'expression suivante :

```
new GregorianDate(2013, Month.MAY, 10)
```

alors qu'avec la version précédente, il était difficile de déterminer laquelle de ces deux dates représentait le 10 mai 2013 :

```
new GregorianDate(2013,  5, 10)
new GregorianDate(2013, 10,  5)
```

Enumération / boolean

De manière similaire, une énumération remplace souvent avantageusement un argument booléen, en rendant les appels plus facile à comprendre.
Par exemple, une méthode de tri permettant de choisir entre ordre croissant ou décroissant pourrait être définie ainsi :

```
<T> void sort(T[] array, boolean ascending)
```

mais ses appels sont alors difficiles à comprendre :

```
sort(myArray, true) // que signifie true ?
```

Question : comment résoudre ce problème au moyen d'une énumération ?

Enumération / `boolean`

Réponse : l'ordre de tri peut être spécifié par une énumération contenant deux valeurs :

```
public enum Order { ASCENDING, DESCENDING }
<T> void sort(T[] array, Order order)
```

ce qui rend les appels très faciles à comprendre :

```
sort(myArray, Order.ASCENDING);
```

switch

Les énumérations sont utilisables avec l'énoncé `switch` :

```
int daysInMonth(int y, Month m) {
  switch (m) {
    case JANUARY:
    case MARCH:
    …
      return 31;
    case APRIL:
    case JUNE:
    …
      return 30;
    case FEBRUARY:
      return isLeapYear(y) ? 29 : 28;
  }
}
```

Traduction des énumérations

Traduction

Chaque énumération est traduite par le compilateur en une classe et ses valeurs en instances (uniques) de cette classe.

Ainsi, la déclaration suivante :

```
public final class GregorianDate {
  public enum Month { JANUARY, … };
  …
}
```

provoque la création d'une classe `Month` imbriquée dans la classe `GregorianDate`. Les valeurs `JANUARY`, `FEBRUARY`, etc. sont des instances de cette classe.

Lorsqu'une énumération est imbriquée dans une classe (comme ici), la classe correspondante est *toujours* imbriquée *statiquement*.

Méthodes statiques

Les classes d'énumérations sont automatiquement équipées des méthodes statiques suivantes :

- `values`, qui retourne un tableau contenant toutes les valeurs de l'énumération, dans l'ordre (p.ex. `Month.values()` retourne un tableau contenant `JANUARY`, `FEBRUARY`, etc.),
- `valueOf`, qui, étant donnée une chaîne de caractère, retourne la valeur ayant ce nom (p.ex. la valeur `JANUARY` si on lui passe la chaîne `"JANUARY"`) ; lève l'exception `IllegalArgumentException` si le nom est invalide.

Méthodes

Les classes représentant des énumération sont de plus équipées de quelques méthodes d'instance :

- `name`, qui retourne le nom de la valeur à laquelle on l'applique (p.ex. la chaîne `"JANUARY"` pour la valeur `JANUARY`, etc.)
- `ordinal`, qui retourne la position de la valeur à laquelle on l'applique dans son énumération, à partir de zéro (p.ex. 0 pour `JANUARY`, 1 pour `FEBRUARY`, etc.)
- `toString`, qui retourne la même chose que `name`,
- `equals` et `hashCode`, bien entendu compatibles,

et quelques autres que nous n'examinerons pas ici.

Interface `Comparable`

Les énumérations implémentent automatiquement l'interface `Comparable` et possèdent donc une mise en œuvre de la méthode `compareTo`.
L'ordre des valeurs d'une énumération est celui de définition.

Par exemple, pour les mois, on a :

- `APRIL.compareTo(MAY)` `<  0`
- `JULY.compareTo(JUNE)` `>  0`
- `AUGUST.compareTo(AUGUST)` `== 0`
- etc.

Exemple

Traduction (simplifiée) de l'énumération `Month`:

```
public final class GregorianDate {
  public final static class Month
      extends Enum<Month> {
    public static final Month JANUARY =
      new Month("JANUARY", 0);
    public static final Month FEBRUARY =
      new Month("FEBRUARY", 1);
…
private Month(String name,
              int ordinal) {
  super(name, ordinal);
}
// … continue à la page suivante
```

Exemple (suite)

```
// … suite de la classe Month
private final static Month[] VALUES =
  { JANUARY, FEBRUARY, … };
public static Month[] values() {
  return VALUES.clone();
}
public static Month valueOf(String s) {
    for (Month m: VALUES) {
      if (m.name().equals(s))
        return m;
    }
    throw new IllegalArgumentException();
  }
}
```

La classe Enum

La classe `Enum` (du paquetage `java.lang`) sert de classe-mère abstraite à toutes les classes d'énumération. Elle est définie ainsi :

```
abstract class Enum<E extends Enum<E>>
      implements Comparable<E> {
  private final String name;
  private final int ordinal;
  protected Enum(String name, int ordinal)
    { … }
  public int ordinal() { return ordinal; }
  …
  public boolean compareTo(E that) { … }
}
```

Borne de Enum

La classe `Enum` possède un paramètre de type `E` dont la borne peut paraître effrayante :

```
Enum<E extends Enum<E>>
```

La raison d'être de cette borne est que `Enum` doit implémenter l'interface `Comparable`.

Il est intéressant d'examiner comment l'un implique l'autre.

Enum v1

Commençons par une version aussi simple que possible de `Enum`, qui ne soit même pas générique. L'interface `Comparable` étant générique, cette classe serait obligée d'implémenter `Comparable<Enum>` :

```
class Enum implements Comparable<Enum> {
  public int compareTo(Enum that) { … }
}
```

Malheureusement, cette version est trop générale puisqu'elle autorise la comparaison de valeurs appartenant à des énumérations différentes (p.ex. `JANUARY` et `ASCENDING`).

Enum v2

Pour résoudre ce problème, on utilise la même idée que pour `Comparable`, c-à-d qu'on ajoute un paramètre de type représentant le type de l'énumération :

```
class Enum<E> implements Comparable<E> {
  public int compareTo(E that) { … }
}
```

C'est mieux, mais pas encore suffisant : pour écrire la méthode `compareTo`, on devrait pouvoir être sûr que la valeur reçue en argument (`that`) est une énumération, afin de pouvoir utiliser la méthode `ordinal` pour faire la comparaison...

Conclusion : il faut borner le paramètre de type `E` !

Enum v3

La méthode ordinal étant définie par la classe `Enum`, le paramètre de type doit être borné par `Enum`.
Mais `Enum` prend un paramètre de type, qui représente le type (précis) de l'énumération. Quel est ce type ici ? `Enum<E>` !
Conclusion : la borne est `Enum<E>`.

Enum v3

On obtient finalement la définition suivante de `Enum` :

```
class Enum<E extends Enum<E>>
      implements Comparable<E> {
   …
   public int ordinal() { … }
   public int compareTo(E that) {
      return Integer.compare(this.ordinal(),
                             that.ordinal());
   }
}
```

valide uniquement grâce à la borne

Intérêt de la borne

Observez que la borne interdit la comparaison de valeurs provenant d'énumérations différentes.

Par exemple, un appel comme :

```
Month.JANUARY.compareTo(Order.ASCENDING);
```

est invalide, car la méthode `compareTo` de `JANUARY` attend un argument de type `Enum<Month>`. Or `ASCENDING` a le type `Order`, sous-type de `Enum<Order>`, qui est incompatible avec `Enum<Month>`.

Comme dit, la première version de notre classe `Enum` aurait permis ce genre de comparaisons, clairement invalides !

Ajout d'attributs aux énumérations

Ajout d'attributs

Les énumérations étant traduites en classe, il est autorisé de leur ajouter des champs et des méthodes, y compris un constructeur – mais ce dernier *doit* être privé.

Exemple :

```
public enum Month {
  JANUARY("Janvier"),
  FEBRUARY("Février"),
  …;

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

arg. du constructeur

Ajout d'attributs (bis)

Il est même possible d'ajouter des attributs aux *valeurs* individuelles d'une énumération. Exemple :

```
public enum Month {
  JANUARY {
    public String frenchName() {
      return "Janvier";
    }
  },
  FEBRUARY { … },
  …;
  abstract String frenchName();
};
```

Une sous-classe anonyme de celle de l'énumération (`Month` ici) est créée pour chaque valeur possédant des attributs.

Enumérations et collections

Collections d'énumérations

Il est souvent utile de stocker des valeurs d'une énumération dans des collections.
Pour ce faire, il est bien entendu possible d'utiliser les collections normales (p.ex. les mises en œuvre de `List`, `Set` et `Map`) mais la nature des énumérations autorise parfois des mises en œuvre plus efficaces.

L'API Java offre donc deux collections spécialisées pour les énumérations :

- `EnumSet` pour les ensembles de valeurs d'énumération, et
- `EnumMap` pour les tables associatives dont les clefs sont des valeurs d'énumération.

EnumSet

`EnumSet` représente un ensemble de valeurs d'énumération par un tableau de bits. Le n^e bit de ce tableau informe quant à la présence de la n^e valeur de l'énumération dans l'ensemble. Ce tableau de bits est de plus encodé dans un ou plusieurs entiers de type `long` capables de stocker 64 bits chacun.

Par exemple, l'ensemble des mois de 30 jours – `APRIL`, `JUNE`, `SEPTEMBER` et `NOVEMBER` – est encodé par le tableau de bits [0,0,0,1,0,1,0,0,1,0,1,0], lui-même encodé dans un entier `long` valant $0\ldots010100101000_2$, ce qui est très efficace.

EnumSet

La mise en œuvre de `EnumSet` utilise, pour des raisons d'efficacité, une technique qu'il est intéressant de connaître.
La classe `EnumSet` elle-même est abstraite, mais elle possède deux sous-classes privées :

- la première ne stocke qu'un seul entier de type `long` et ne fonctionne donc que pour les énumérations possédant au plus 64 valeurs,
- la seconde stocke un tableau d'entiers de type `long` et fonctionne pour toutes les énumérations.

Toutefois, le choix de la sous-classe est fait à l'exécution en fonction de la taille de l'énumération utilisée.

Méthodes de création

Pour ce faire, ni `EnumSet` ni ses sous-classes n'offrent de constructeur public. Au lieu de cela, `EnumSet` propose des méthodes de création statiques qui examinent l'énumération utilisée et choisissent la bonne mise en œuvre en fonction.
Une de ces méthodes est p.ex. la méthode `of` qui accepte un nombre variables d'arguments (au moins un) et construit un ensemble contenant ces valeurs. Exemple :

```
EnumSet.of(Month.APRIL,
           Month.JUNE,
           Month.SEPTEMBER,
           Month.NOVEMBER);
```

EnumMap

`EnumMap` représente une table associative dont les clefs proviennent d'une énumération et les valeurs ont le type V par un tableau de V. La n^e entrée de ce tableau donne la valeur de la n^e valeur d'énumération.

Par exemple, la table donnant le nom français des mois est représentée par le tableau de chaînes à 12 entrées contenant `"Janvier"` à l'index 0, `"Février"` à l'index 1, et ainsi de suite. Là aussi, cette représentation est très compacte et efficace.