Enumérations

Philippe Genoud
dernière mise à jour : 01/02/2022 16:40

- définition de type énumérés à l'aide de constantes entières

```
public class Jour {
   public static final int LUNDI = 1;
   public static final int MARDI = 2;
   ...
   public static final int DIMANCHE = 7;
}
```

- Non « type-safe » : un Jour étant simplement un **`int`** il est possible de passer n'importe quelle valeur là où un jour est attendu, ou d'additionner deux jours ensembles
- Fragilité : les valeurs énumérées sont des constantes fixées dans les clients qui les utilisent au moment de la compilation. Si une nouvelle constante est ajoutée entre deux constantes ou que l'ordre est changé les clients doivent être recompilés, sinon ils continueront à fonctionner mais leur comportement pourra être imprévisible
- Non informativité : lorsque vous imprimez une valeur énumérée vous n'obtenez qu'un entier ce qui ne dit pas ce que cette valeur représente ni quel est son type.
- Non objet : pas d'opérations (méthodes) définies sur les valeurs d'un type énuméré

- nécessité de passer par des patterns objets pour répondre à ces problèmes

```
public class Jour {
   public static final Jour LUNDI = new Jour("Lundi",1);
   public static final Jour MARDI = new Jour("Lundi",2);
   ...
   public static final Jour DIMANCHE = new Jour("Dimanche",7);

   private static Jour[] values = { LUNDI, MARDI, ... , DIMANCHE };

   private String nom;
   private int rang;

   private Jour(String nom, int rang) {
      this.nom = nom;
      this.rang = rang;
   }

   public String toString() {
      return nom;
   }

   public int getRang() {
       return rang;
   }

   public Jour suivant() {
       return values[(this.rang + 1) % 7];
   }
}
```

- *Item 21 dans « Effective Java » de Joshua Bloch*
- *Mais ce n'est pas la panacée*
  - *à la charge du programmeur*
  - *très verbeux (-> augmente risque d'erreurs)*
  - *les valeurs de types énumérées ainsi définis ne peuvent être utilisées dans des instructions switch*

- Java 5.0 fournit un type énuméré « type-safe » en standard via le mot clé **enum**

```
public enum Jour {
   LUNDI, MARDI, ..., DIMANCHE ;
}
```

```
public enum Mois {
   JANVIER, FEVRIER, ..., DECEMBRE ;
}
```

```
public class RendezVous {
   private Jour j;
   private Mois m;
   ...

   public RendezVous(Jour j, Mois m, ...) {
      this.j = j;
      this.m = m;
      ...
   }
   ...
   String toString() {
      return "Rendez-vous : " + j + ... ;
}
```

Dans sa forme le plus simple un type énuméré définit un ensemble de valeurs

Un type énuméré est utilisé comme un type objet :

- on peut déclarer des références sur le type énuméré
- les seules valeurs possibles pour ces références sont les constantes définies dans le type énuméré

```
...
RendezVous rv = new RendezVous(
      Jour.LUNDI, Mois.DECEMBRE, ...);
...
System.out.println(rv);
...
```

```
...
Rendez-vous : LUNDI ...
...
```

Enumérations Types énumérés

- Les types énumérés étant définis comme des classes possibilité de définir des types beaucoup plus riches en ajoutant des données et un comportement aux objets d'une énumération

Exemple : Les planètes du système solaire.

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22,  1.137e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    private double mass()   { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

Chaque constante de l'énumération est déclarée avec des paramètres qui seront passés au constructeur

Chaque planète a une masse et un rayon

L'énumération possède un constructeur

Chaque planète peut calculer la gravité à sa surface et le poids d'un objet à sa surface

• Exemple d'utilisation de l'énumération `Planet`

Un programme qui prend votre poids sur terre (dans n'importe quelle unité) , calcule et affiche votre poids (dans la même unité) sur les différentes planètes du système solaire

Boucle « foreach » (java 5.0), simplification d'écriture pour itérer facilement sur les éléments d'un tableau ou d'une collection

```
public class TestPlanets {

  public static void main(String[] args) {
     double earthWeight = Double.parseDouble(args[0]);
     double mass = earthWeight/EARTH.surfaceGravity();
     for (Planet p : Planet.values()) {
           System.out.printf("Your weight on %s is %f%n", p, p.surfaceWeight(mass));
     }
  }
}
```

`values()` : Méthode statique qui pour une énumération retourne un tableau contenant toutes les valeurs énumérées dans l'ordre où elles ont été déclarées

```
Planet[] lesPlanetes = Planet.values();
for (int i = 0; i < lesPlanetes.length; i++) {
    Planet p = lesPlanetes[i];
    System.out.println(…, p,…);
}
```

```
$ java TestPlanets 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
Your weight on PLUTO is 11.703031
```

Records

Philippe Genoud
dernière mise à jour : 01/02/2022 16:41

Motivation

- Nouveau type d'objets introduits en mars 2021 avec Java 16
- Pourquoi ?
  - répondre aux reproches d'excessive verbosité concernant le langage Java
  - en particulier pour des objets immuables dont le rôle consiste uniquement à transporter des données (*immuatable data carriers* ou *data transfert objects*)
- écrire de telles classes implique l'écriture de beaucoup de code à faible valeur ajoutée (boiler plate code) : répétitif et source d'erreurs
  - constructeurs
  - accesseurs
  - equals
  - hashCode
  - toString
  - …

Objet POJO (Plain Old Java Object)

Personne.java

```java
public class Personne {

    private final String nom;
    private final String prenom;
    private final int anneeNaissance;

    public Personne(String nom, String prenom, int anneeNaissance) {
        this.nom = nom;
        this.prenom = prenom;
        this.anneeNaissance = anneeNaissance;
    }

    public String getNom() {
        return nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public int getAnneeNaissance() {
        return anneeNaissance;
    }

    @Override
    public String toString() {
        return "Personne [anneeNaissance=" + anneeNaissance
            + ", nom=" + nom + ", prenom=" + prenom + "]";
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + anneeNaissance;
        result = prime * result + ((nom == null) ? 0 : nom.hashCode());
        result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
        return result;
    }
    @Override
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            Personne other = (Personne) obj;
            if (anneeNaissance != other.anneeNaissance)
                return false;
            if (nom == null) {
                if (other.nom != null)
                    return false;
            } else if (!nom.equals(other.nom))
                return false;
            if (prenom == null) {
                if (other.prenom != null)
                    return false;
            } else if (!prenom.equals(other.prenom))
                return false;
            return true;
        }

}
```

```java
public class AppPersonne{

    public static void main(String[] args) {
        Personne p1 = new Personne("DUPONT", "Jean", 1976);
        Personne p2 = new Personne("DUPONT", "Jean", 1990);
        System.out.println(p1);
        System.out.println(p2);
        System.out.println("p1 == p2 : " + p1.equals(p2));
    }
}
```

Personne [anneeNaissance=1976, nom=DUPONT, prenom=Jean]
Personne [anneeNaissance=1990, nom=DUPONT, prenom=Jean]
p1 == p2 : false

Objet Record (Java 16 +)

PersonneRecord.java

```
public record Personne(String nom, String prenom, int anneeNaissance) {}
```

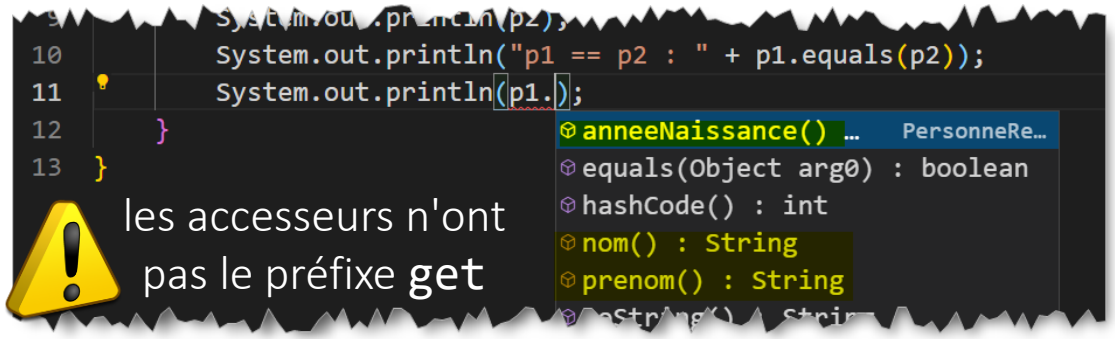
Méthodes accesseurs publiques

Constructeur canonique

Méthodes **equals** et **hashCode**

Méthode **toString**

Code généré automatiquement par le compilateur




```
public class AppPersonne{

    public static void main(String[] args) {
        Personne p1 = new Personne("DUPONT", "Jean", 1976);
        Personne p2 = new Personne("DUPONT", "Jean", 1990);
        System.out.println(p1);
        System.out.println(p2);
        System.out.println("p1 == p2 : " + p1.equals(p2));
    }
}
```

Personne [anneeNaissance=1976, nom=DUPONT, prenom=Jean]
Personne [anneeNaissance=1990, nom=DUPONT, prenom=Jean]
p1 == p2 : false

- Possibilité d'ajouter du code à un Record

rajouter des membres statiques

étendre le constructeur canonique

définir d'autres constructeurs

ajouter des méthodes

```
public record Personne (String nom, String prenom, int anneeNaissance) {

    public static final int ANNEE_MIN = 1900;
    public static final int ANNEE_DEFAUT = 1900;

    public Personne {    // on ne rappelle pas les paramètres
        if (anneeNaissance < ANNEE_MIN) {
            throw new IllegalArgumentException("l'année de naissance doit être >= 1900");
        }
    }

    public Personne(String nom, String prenom) {
        this(nom,prenom,ANNEE_DEFAUT);
    }                                    // doit obligatoirement appeler le constructeur
                                         // canonique en 1ère instruction

    public boolean plusVielleQue(PersonneRecord p) {
        return this.anneeNaissance < p.anneeNaissance;
    }

}
```

- Ce que l'on ne peut pas faire avec un Record

ajouter des attributs (*fields*) non statiques

```
1  package fr.im2ag.m2cci
2
3  public record Personne
4
5      private String civilité;
```

String civilité

User declared non-static fields civilité are not permitted in a record Java(16778949)

View Problem No quick fixes available

sous classer un type record

```
1  package fr.im2ag.m2cci.recordsdemo;
2
3  public class Etudiant extends Personne {
4
5  }
6
```

fr.im2ag.m2cci.recordsdemo.Personne

The record Personne cannot be the superclass of Etudiant; a record is final and cannot be extended Java(16778983)

View Problem No quick fixes available