Enum

Spis treści

Zapiski prowadzącego Karola Rogowskiego i uczestnika Bootcampu Zajavka Bartek Borowczyk aka Samuraj Programowania.

Czym jest enum?

Enum jest specjalnego rodzaju "klasą", która zawiera w sobie grupę stałych (stałych zmiennych 😉). Czyli takich wartości, których nie możemy zmienić. Tworząc `enum`, na etapie kompilacji określamy już, jakie wartości są stałymi dla naszego `enum`a.

Najprostszym sposobem na stworzenie `enum`a jest utworzenie pliku z takim zapisem:

```
public enum Color {
    RED,
    BLUE,
    GREEN
}
```

W ten sposób określiliśmy enum `Color`, który ma zdefiniowane 3 konkretne wartości: `RED`, `BLUE`, `GREEN`. Nie ma np. wartości `YELLOW`, więc nie możemy jej użyć. Wartości oddzielane są przecinkami, po ostatniej może pojawić się (ale w przykładzie jak wyżej nie musi) średnik.

Natomiast żeby odwołać się do użycia enuma, należy napisać to w taki sposób:

```
Color color = Color.BLUE;
```

W tym momencie określamy, że pod zmienną kolor, znajduje się enum `Color` o wartości `BLUE`.

Na pewno też już widzisz, że panuje taka konwencja, że jeżeli coś jest stałą wartością, to piszemy tę wartość w całości wielką literą.

Enum a instrukcje warunkowe

Enum w wyrażeniach if-else

Enum jest bardzo dobrym kandydatem do używania go w wyrażeniach warunkowych. Konstrukcja taka jak poniżej jest często spotykana, gdyż popularne jest porównywanie stałych w zastosowaniach na co dzień. Przykład wykorzystania enuma w konstrukcji `if-else`:

```
Color color = Color.BLUE;

if (color == Color.BLUE) {
    System.out.println("PRINT BLUE");
} else if (color == Color.RED) {
    System.out.println("PRINT RED");
} else if (color == Color.GREEN) {
    System.out.println("PRINT GREEN");
}
```

Enum w wyrażeniach switch

Pokazaliśmy przykład wykorzystania enuma w wyrażeniach `if-else`, może być on również stosowany w konstrukcji `switch`. Często spotyka się konstrukcje enuma w `switch`, gdy mamy do obsłużenia dużą ilość stałych. Przykładowo:

```
Color color = Color.BLUE;

switch (color) {
    case BLUE:
        System.out.println("PRINT BLUE");
        break;
    case RED:
        System.out.println("PRINT RED");
        break;
    case GREEN:
        System.out.println("PRINT GREEN");
        break;
    default:
        System.out.println("NOTHING");
        break;
}
```

Popularne metody

Najpopularniejsze metody, jakie dostarcza nam enum to:

valueOf(String value)

Metoda ta pozwala nam stworzyć enum na podstawie przekazanego Stringa. Jest case-sensitive. Czyli ma to znaczenie czy wartość przekazana jako argument metody jest litera po literze taka sama i czy jest to wielką, czy małą literą. Przykładowo:

```
public enum Day {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
// Gdzieś w kodzie
Day monday1 = Day.valueOf("MONDAY"); // wykona się poprawnie
Day monday2 = Day.valueOf("Monday"); // podczas działania programu zostanie wyrzucony błąd
```

name()

Zwraca nam String z wartością enuma, czyli:

```
System.out.println(Planet.EARTH.name()); // wydrukuje EARTH
```

Efekt byłby ten sam, jeżeli wydrukowalibyśmy coś takiego:

```
System.out.println(Planet.EARTH); // wydrukuje EARTH
```

Oznacza to, że domyślna implementacja metody `toString()` tak naprawdę zwraca nam to samo co `name()`.

ordinal()

Metoda ta podaje nam miejsce, na którym dana wartość enuma się znajduje. Zwróć uwagę, że wartość ta jest podawana od 0. Przykładowo:

```
public enum Animal {
    DOG,
    CAT,
    BIRD,
    FISH
}

// Gdzieś w kodzie
System.out.println(Animal.DOG.ordinal()); // 0
System.out.println(Animal.BIRD.ordinal()); // 2
```

Iterowanie po wartościach

Czyli w jaki sposób można pobrać wszystkie wartości enuma np. w tablicy? Zwróć uwagę, że metoda `.values()` zwraca tablicę ze wszystkimi wartościami enuma. Zobaczysz czemu to jest istotne jak poruszymy kolekcje ☺. Można to zrobić tak:

```
Animal[] values = Animal.values();
for (Animal value : values) {
    System.out.println("Printing value: " + value);
}
```

Definiowanie własnych pól i metod w enum

Enum może mieć zdefiniowane pola, czyli możemy w przypadku danego modelu samochodu zapisać, jaki może on mieć kolor, albo jego inne cechy.

```
public enum Volkswagen {
    PASSAT("white", 2020),
    GOLF("red", 1020),
    ARTEON("green", 3040),
    TIGUAN("blue", 2021),
    TOUAREG("brown", 3020);

    private String color;

    private int productionYear;

    Volkswagen(final String color, final int productionYear) {
      this.color = color;
      this.productionYear = productionYear;
    }
}
```

W przykładzie powyżej mamy określone modele samochodu marki Volkswagen, gdzie każdy z nich ma podany swój kolor oraz rok produkcji. Zmienne te definiowane są tak samo, jak pola w klasie, przy czym ważne jest, że jeżeli zdefiniujemy takie pole, to w enumie musi się pojawić konstruktor, w którym takie pole jest inicjowane. Konstruktor ten nie może być ani `public`, ani `protected`, bo dostaniemy wtedy błąd kompilacji. Może być on natomiast `package-private` lub `private`.

Tu uczulam, nawet jeżeli jest on `package-private`, to nie próbuj tworzyć z niego obiektu w innej klasie w tej samej paczce. Ten konstruktor ma być wykorzystany tylko i wyłącznie w enumie, w którym jest napisany do zainicjowania określonych przez Ciebie pól.

Jeżeli po ostatniej wartości enuma pojawia się jeszcze coś, tzn. pola, metody - za ostatnią wartością musi się wtedy pojawić średnik. W przeciwnym wypadku, po ostatniej wartości enuma, nie musimy nic pisać, czyli możemy to zrobić tak jak w pierwszym przykładzie enuma `Color`.

Skoro mamy już pola, to żeby móc z nich w pełni korzystać, musimy określić też gettery.

```
public String getColor() {
    return color;
}

public int getProductionYear() {
    return productionYear;
}
```

Są one jednocześnie przykładem na to, w jaki sposób zdefiniować w enumie metody. Możemy teraz spokojnie odwołać się do tych metod w taki sposób:

```
System.out.println(Volkswagen.PASSAT.getColor()); // white
System.out.println(Volkswagen.PASSAT.getProductionYear()); // 2020
```

Uczulam tutaj, żeby nie definiować w enumie setterów. Technicznie można to zrobić, ale nie robi się takich potworków w życiu codziennym.

Enum i metody abstrakcyjne

Możemy definiować metody abstrakcyjne w enumie, tak jak są one definiowane w klasie abstrakcyjnej. Wymusi to wtedy na nas implementację tych metod oddzielnie przez każdą wartość enuma, jaką zdefiniujemy:

```
public enum Volkswagen {
    PASSAT("white", 2020) {
        @Override
        public String findAnotherColor() {
          return "blue";
        }
    },
    GOLF("red", 1020) {
        @Override
        public String findAnotherColor() {
          return "black";
        }
    };

    private String color;

    private int productionYear;

    Volkswagen(String color, int productionYear) {
        this.color = color;
        this.productionYear = productionYear;
    }

    public abstract String findAnotherColor();

    public String getColor() {
        return color;
    }

    public int getProductionYear() {
        return productionYear;
    }

}
```

Mamy w przykładzie zdefiniowaną metodę abstrakcyjną `findAnotherColor()`, która musi zostać nadpisana przez każdą wartość enuma, jaką zdefiniujemy. Musi zostać nadpisana, czyli określona, bo nie ma ona ciała, jedynie jest to zapis, że takie zachowanie w enumie wystąpi, ale nie wiadomo co to zachowanie robi. Jeżeli nie nadpiszemy tej metody, Java wyrzuci nam błąd kompilacji. W tym przypadku metoda abstrakcyjna `findAnotherColor()` ma nam zwrócić alternatywny kolor, na który możemy pomalować samochód, jeżeli jego pierwotny kolor nie jest dostępny. Jednocześnie też należy pamiętać, że enum nie może być klasą abstrakcyjną, więc po dodaniu metody abstract, musimy ją zaimplementować w konkretnych wartościach enuma. Dopisanie słówka abstract przez słówkiem enum nie rozwiąże problemu.

Enum a implementowanie interfejsów

Enum może implementować interfejsy tak samo, jak normalna klasa. Wystarczy dodać słówko implements oraz interesujący nas interface.

Enum Day

```
public enum Day implements WorkingDay, Weekend {
    MONDAY(true),
    TUESDAY(true),
    WEDNESDAY(true),
    THURSDAY(true),
    FRIDAY(true),
    SATURDAY(false),
    SUNDAY(false);

    private final boolean workingDay;

    Day(boolean workingDay) {
        this.workingDay = workingDay;
    }

    @Override
    public boolean isWeekend() {
        return !workingDay;
    }

    @Override
    public boolean isWorkingDay() {
        return workingDay;
    }
}
```

Interface Weekend

```
public interface Weekend {
  boolean isWeekend();
}
```

Interface WorkingDay

```
public interface WorkingDay {
  boolean isWorkingDay();
}
```

Warto pamiętać wtedy o adnotacji `@Override`, dzięki niej możemy uniknąć wielu problemów na etapie kompilacji, które były omawiane wcześniej.