Výčtové typy

Table of Contents

Motivace k výčtovému typu

Chceme reprezentovat dny v týdnu.

Klasika jako v C

```
public static final int MONDAY = 0;
public static final int TUESDAY = 1;
public static final int WEDNESDAY = 2;
```

Ale nemáme žádnou kontrolu

typovou

metoda přijímající *den v týdnu* má parametr typu `int`, takže bere libovolné číslo, třeba `2000`, a to nebude fungovat.

hodnotovou

dva dny v týdnu mohou omylem mít stejnou hodnotu a překladač nám to taky neodchytí.

Výčtový typ `enum`

- Typově bezpečná cesta, jak vyjmenovat a používat pojmenované konečné výčty prvků.
- Proměnná určitého výčtového typu může pak nabývat vždy jedné hodnoty z daného výčtu.
- Definice výčtového typu "den v týdnu":

```
public enum Day {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Jak to bylo v Pythonu?

Zde bylo třeba podědit ze třídy `Enum` a ručně ohodnotit jednotlivé prvky výčtu:

```
class State(Enum):
    INIT = 1
    RUNNING = 2
    STOPPED = 3
```

Příklad použití výčtu

Velmi příjemné použití ve větvení `switch`

```
public String tellItLikeItIs(Day day) {
  switch (day) {
    case MONDAY:
      return "Mondays are bad.";
    case FRIDAY:
      return "Fridays are better.";
    case SATURDAY, SUNDAY:
      return "Weekends are best.";
    default:
      return "Midweek days are so-so.";
  }
}
```

Klíčové slovo `break` může být vynecháno, protože `return` způsobí okamžitý návrat z funkce.

Porovnání `enum`

Lze snadno a bezpečně testovat rovnost pomocí `==`:

Porovnávání

- `day == MONDAY`
- `day.equals(MONDAY)` - funguje stejně, pouze selže při `day == null`
- `MONDAY.equals(day)` - bezpečněji, ale stejně lépe psát `day == MONDAY`

Uspořádání

- Díky tomu, že každý výčet implementuje `Comparable<E>`, lze prvky i řadit.
- Platí tedy, že `MONDAY` je před `TUESDAY` atd.

Výčty vs. třídy

- Výčtový typ se koncepčně velmi podobá *třídě*, de facto *je to třída*.
- Výčet má však jen *pevně daný počet prvků* (instancí).
- Pro každý prvek daného typu `enum` je získatelné jeho ordinální číslo (pořadí) metodou `int ordinal()`.
- Každý námi definovaný výčtový typ je potomkem třídy `java.lang.Enum`.
- Podobně jako jiné třídy má vestavěné metody a může mít *další metody*, *konstruktory* apod.

Výčtový typ s dalšími vlastnostmi

- Využijeme, že `enum` je de facto třída.
- Přidáme *atributy*, případně *metody* nebo i *konstruktor*.
- Dobře se využije možnost definovat jednotlivé prvky výčtu s inicializací jeho atributů.
- Lze i překrýt metody jako `toString` a vracet jiné textové reprezentace
- *Nelze* však překrýt `equals`, porovnání proto zůstává takové, že *x.equals(y) ~ x == y*

Příklad "Order"

```
enum OrderStatus {
    // 3 instances and no more ever
    // they might by initialized with parameters
    ORDERED(5), PREPARING(2), READY(0);
    // enum may have attributes (not necessarily final)
    int timeToReady;
    OrderStatus(int timeToReady) {
        this.timeToReady = timeToReady;
    }
    ...
}
```

```
    // may override toString
    @Override public String toString() {
      return "ready in " + timeToReady;
    }
    // this would not compile! must NOT override equals
    @Override public boolean equals(Object o) {
        if(o instanceof OrderStatus status)
            return timeToReady == status.timeToReady;
        return false;
    }
```

Příklad (2)

```
OrderStatus status = OrderStatus.PREPARING;
System.out.println(status);
// may even directly modify status properties
OrderStatus.PREPARING.timeToReady = 1;
System.out.println(status);
// status is still == (though modified inside)
System.out.println(status == OrderStatus.PREPARING);
```

Není úplně jasné, zda modifikovatelnost prvků výčtu je či není stylisticky OK.

Výčty mají předem přesně daný počet instancí

- Toto se nezkompiluje, **duplicitní identifikátor** `ORDERED`.
- Že pokaždé jinak inicializovaný, nehraje roli.
- Parametry v závorce nezpůsobí vytvoření nových prvků jako `new`.

```
enum OrderStatus {
  ORDERED(5), ORDERED(15), PREPARING(2), READY(0);
}
```

Prvky jsou uspořádané

- Jsou uspořádané dle pořadí, v jakém jsou uvedeny, aniž bychom cokoli definovali.
- Nelze ovšem psát `OrderStatus.ORDERED < OrderStatus.PREPARING`, ale je nutno využít `compareTo`:
- `OrderStatus.ORDERED.compareTo(OrderStatus.PREPARING)`

Množina prvků výčtu EnumSet

- Jelikož prvků výčtu je konečný (a většinou nevelký) počet, je dobré, že pro ně máme speciální typ množiny — `java.util.EnumSet`.
- `EnumSet` je abstraktní podtřídou `AbstractSet` a má dvě neabstraktní implementace:

  RegularEnumSet
  : vnitřně implementován jako bitové pole do velikosti 64 bitů, a to pomocí jednoho `long`

  JumboEnumSet
  : jako bitový vektor libovolné, tzn. i větší velikosti

Příklad EnumSet

```
EnumSet<OrderStatus> set = EnumSet.of(OrderStatus.ORDERED, OrderStatus.READY);
// prints [ready in 5, ready in 0]:
System.out.println(set);
// prints [ready in 2]:
System.out.println(EnumSet.complementOf(set));
// prints class java.util.RegularEnumSet:
System.out.println(set.getClass());
```

Další použitelné (statické) metody EnumSet:

EnumSet.noneOf()
: vytvoří prázdnou množinu

EnumSet.range(LOWER, UPPER)
: vytvoří množinu prvků od *LOWER* po *UPPER*

Mapa s klíči prvky výčtu EnumMap

- Z obdobného důvodu se nabízí rovněž mapa `EnumMap`.
- Klíčem jsou prvky výčtu (tzn. jsou pevně předem dány a většinou jich není moc).

```
EnumMap<OrderStatus, Integer> countOrders
  = new EnumMap<OrderStatus, Integer>(OrderStatus.class);
countOrders.put(OrderStatus.ORDERED, 4);
countOrders.put(OrderStatus.READY, 2);
System.out.println(countOrders);
```

Replit.com demo k výčtovým typům

- https://replit.com/@tpitner/PB162-Java-Lecture-03-enum

Další zdroje

- Hezký příklad najdete na The Java™ Tutorials — Enum Types