Konstruktory

Table of Contents

Co jsou konstruktory

- Konstruktory jsou speciální *metody* volané při vytváření nových objektů (=instancí) dané třídy.
- V konstruktoru se typicky *inicializují atributy (proměnné) objektu*.
- Konstruktory lze volat jen ve spojení s operátorem **new** k vytvoření nového objektu.

Konstruktory v Pythonu a Javě

- V Pythonu jsou to metody `def init(self):`
- V Javě se jmenují přesně stejně jako jejich třída (a bez návratového typu) `public Person(String name) { }`.
- Konstruktorů v jedné třídě může být více, musí se pak lišit počtem, evt. typem parametrů.

Příklad třídy s konstruktorem v Pythonu

```
class Person:
```

```
    # default constructor
    def __init__(self, name):
        self.name = name
```

Příklad třídy s konstruktorem v Javě

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
}
```

Rozdíly Python vs Java

Java

identifikátor `this` znamená, že se přistupuje k atributům objektu. *Není nutné* používat tam, kde se neshoduje jméno atributu a parametru. Někdo `this` systematicky používá, někdo nepoužívá. Tendence je spíše nepoužívat.

Python

identifikátor `self` znamená totéž, ale musí se používat *vždy při přístupu* k atributu nebo metodě objektu.

Konstruktory — použití

Python

```
pepa = Person("Pepa from Hongkong")
```

Java

```
Person pepa = new Person("Pepa from Hongkong");
```

- Toto volání vytvoří objekt `pepa` a naplní ho jménem.
- Následně je možné získávat hodnoty proměnných objektů pomocí tečkové notace, např. `pepa.name`.
- V tomto případě by nebylo možné volat `Person pepa = new Person()`,
- ! protože existující konstruktor má jeden parametr.

Více konstruktorů v jedné třídě

- Je dost typické, že třída má dva či více konstruktorů
- Jedná se o tzv. přetěžování konstruktorů (overloading)

```java
public class Person {
  private String name;
  public Person() {
    this.name = "<UNKNOWN>";
  }
  public Person(String name) {
    this.name = name;
  }
}
```

Výběr konstruktoru

- Při spuštění pak musíme řešit, který konstruktor se má použít
- To je však intuitivní: použije se ten, který se hodí dle zadaných parametrů

Se jménem nebo bez?

```java
public class Person {
  private String name;
  public Person() {
    this.name = "<UNKNOWN>";
  }
  public Person(String name) {
    this.name = name;
  }
}
```

Lze si vybrat

```java
public class Demo {
  public static void main(String[] args) {
    Person noname = new Person(); // <UNKNOWN>
    Person jan = new Person("Jan Fabián"); // Jan Fabián
  }
}
```

Výchozí (default) konstruktor

- Co když třída nemá definovaný žádný konstruktor?
- Vytvoří se automaticky *výchozí (default)* konstruktor:

```java
public Person() { }
```

Použití výchozího konstruktoru

```
Person p = new Person();
```

Výchozí (default) konstruktor se vytvoří pouze v případě, že žádný jiný konstruktor v třídě neexistuje.

Pouhá deklarace proměnné

```
Person p;
System.out.println(p.getName());
```

- Výrazný rozdíl oproti C++: v Javě vůbec nepůjde přeložit.
- Nevytvoří žádný objekt a překladač ví, že proměnná `p` neukazuje nikam.
- Tudíž veškerá volání `p.getName()` a podobně by byla nefunkční.

Nevytvoření objektu

- Když do odkazu přiřadím `null`, přeložit půjde.
- Nicméně, co se stane, když zavolám nad odkazem `null` metody?

```
Person p = null;
System.out.println(p.getName());
```

- Kód po spuštění "spadne", neboli zhavaruje, předčasně skončí.
- Java sa snaží pád programu popsat pomocí *výjimek* (exceptions).
- Výjimky mají své *jméno*, obvykle i určitý textový popis dokumentující příčinu havárie.

```
Exception in thread "main" java.lang.NullPointerException
```

Návratový typ konstruktoru?

- "prázdný" typ `void`? NIKOLI!
- konstruktory vracejí odkaz na vytvořený objekt
- *návratový typ nepíšeme*, typem je fakticky odkaz na nově vytvořený objekt

Proměnná objektového typu

- Bavíme se o proměnných *lokálních* ve kódu metod.

- Proměnná objektového typu se deklaruje např. `Person p;`
- Deklarace proměnné objektového typu sama o sobě *žádný objekt nevytváří*.
- Takové proměnné jsou pouze *odkazy* na *dynamicky vytvářené objekty*.
- Vytvoření objektu se děje až operátorem `new` dynamicky, instance se vytvoří až za běhu programu.
- V Javě se celé objekty do proměnné *neukládají*, jde vždy o uložení pouze *odkazu* (adresy) na objekt.

Přiřazení proměnné objektového typu

- Přiřazením takové proměnné pouze *zkopírujeme odkaz*.
- Na jeden objekt se odkazujeme nadále ze dvou míst.
- **Nezduplikujeme** tím objekt.

Příklad kopie odkazu na objekt

Proměnné `jan` a `janCopy` ukazují na ten tentýž objekt ⇒ změna objektu se projeví v obou:

```
public static void main(String[] args) {
  Person jan = new Person("Jan");
  Person janCopy = jan;
  janCopy.name = "Janko"; // modifies jan too
  System.out.println(jan.name); // prints "Janko"
}
```

Přiřazení `janCopy.name = "Janko"` bude možné jen tehdy, nebude-li atribut `name` privátní, jinak bychom museli mít něco jako `janCopy.setName("Janko")`.

Konstruktory — shrnutí

Jak je psát a co s nimi lze dělat?

- neuvádí se *návratový typ*
- mohou a nemusí mít *parametry*
- když třída nemá žádný konstruktor, automaticky se vytvoří *výchozí*
- může jich být *více* v jedné třídě, reálně se používá

Návrhové vzory

- Návrhové vzory jsou osvědčené způsoby objektové dekompozice v jasně popsaných situacích
- Jsou použitelné pro libovolný objektově orientovaný jazyk

- Jejich aplikace ale vyžaduje návrhová rozhodnutí, která mohou být ovlivněna vlastnostmi programovacího jazyka
- Mnohé si postupně stručně představíme s cílem
  - demonstrovat, že objektová dekompozice Java Core API není náhodná,
  - motivovat vás k používání vozrů při dekompozici vašeho kódu.

Vytvářecí návrhové vzory

Speciální podskupina návrhových vzorů, která nabízí alternativní způsoby k vytváření objektů, než je prosté volání konstruktoru

- Singleton: Vytvoření jediné instance třídy, kterou všichni sdílí a snadno k ní přistupují odkudkoli.
- Builder: Konstrukce složitého objektu po kouscích (např. vytvoření grafu přidáváním uzlů a hran).
- Prototype: Namísto vytváření nového objektu naklonuj existující objekt.
- Abstract Factory: Jednotné místo pro vytváření vzájemně kompatibilních instancí různých tříd.
- Factory Method: Přenechání konstrukce objektu podtřídě.
- Dobrá praxe dle Josh Bloch: Effective Java