

Rozhraní

Table of Contents

Motivace	1
Motivační příklad	1
Rozhraní v Pythonu	2
Rozhraní v Pythonu	2
Příklad jednoduchého rozhraní	2
Přesněji k rozhraní	3
Deklarace rozhraní	3
Implementace rozhraní	3
Typ rozhraní namísto třídy	4
"Přetypování" obecně	4
Typová kontrola odkazů na objekty	4
Příklad - typová kontrola odkazů	4
Zjištění typu	5
Přetypování u primitivních typů	5
Proměnná deklarovaná jako rozhraní	5
Proměnná deklarovaná jako rozhraní	5
Příklad z reálného světa	6
Vytvoříme pro všechny jedno uniformní rozhraní	6
Anotace <code>@Override</code>	6
Replit demo k rozhraním	6

Motivace

- Jsou objektové jazyky, kde vůbec rozhraní nemáme.
- Jsou dokonce objektové jazyky, kde nemáme ani třídy (JavaScript).
- Většinou ale *třídy jsou*, u většiny OO jazyků: Java, C++, C#, Python, Ruby...
- Přímočaré řešení je pak použít na implementaci nějaké potřebné funkčnosti třídu.
- Problém je, že někdy máme takový požadavek na funkčnost, který se jednou třídou špatně implementuje, resp. evidentně potřebujeme více *zcela různorodých tříd*, které budou tento požadavek plnit.

Motivační příklad

- Příklad: požadavkem je, aby objekty uměly o sobě sdělit informace, tedy aby měly třeba metodu `retrieveInfo()`.

- Zcela jistě existuje mnoho typů objektů, které o sobě umějí informovat — od psa až po laserovou tiskárnu
- Tudíž ani není možné všechny třídy objektů, které o sobě umějí informovat, chápat jako podtřídy jedné třídy "UmímOSoběInformovat".
- V Javě navíc třída smí mít pouze jednu rodičovskou třídu/předka (=dědit jen z jedné rodičovské), tzn. bychom tím "spotřebovali" možného předka a žádného jiného předka už bychom nikde nemohli mít.

Rozhraní v Pythonu

- Zde máme tzv. *neformální rozhraní* (informal interfaces).
- Jedná se o dynamický jazyk, při překladu nekontroluje, zda jsou všechny metody implementované.
- Napíšeme do *docstringů*, co má metoda dělat a uvedeme do ní `pass`.
- V podtřídě pak metodu/y překryjeme.

Rozhraní v Pythonu

```
class InformalParserInterface:
    def load_data_source(self, path: str, file_name: str) -> str:
        """Load in the file for extracting text."""
        pass

    def extract_text(self, full_file_name: str) -> dict:
        """Extract text from the currently loaded file."""
        pass
```

- [Implementing an Interface in Python](#)

Příklad jednoduchého rozhraní

- V Javě musíme trochu formálněji.
- Velmi jednoduché rozhraní s jedinou metodou:

```
// Informing = can describe information about itself
public interface Informing {
    // this method is used for describing
    String retrieveInfo();
}
```

Přesněji k rozhraní

- Rozhraní (**interface**) je *seznam metod* (budoucích) tříd objektů.
- *Neobsahují vlastní kód* těchto metod.
- Je to tedy *seznam hlaviček metod* s popisem, co přesně mají metody dělat, typicky dokumentačním komentářem.
- Rozhraní by nemělo příliš smyslu, kdybychom neměli třídy, které jej naplňují, realizují, přesněji *implementují*.
- Říkáme, že třída *implementuje* rozhraní, pokud obsahuje *všechny metody*, které jsou daným rozhraním předepsány. tedy umí to, co rozhraní předepisuje.

Deklarace rozhraní

- Stejně jako třída, jedině namísto slova **class** je **interface**.
- Všechny *metody v rozhraní přebírají viditelnost* z celého rozhraní
- viditelnost hlaviček metod není nutno psát;
- rozhraní v našem kurzu budou pouze **public** (není to vůbec velká chyba tak dělat stále).
- Těla metod v deklaraci rozhraní se nepíší vůbec, ani složené závorky, jen středník za hlavičkou.

```
// Informing = can describe information about itself
public interface Informing {
    // this method is used for describing
    String retrieveInfo();
}
```

Implementace rozhraní

Třídy **Person** a **Donkey** implementují rozhraní **Informing**:

```
public class Person implements Informing {
    public String retrieveInfo() {
        return "People are clever.";
    }
}

public class Donkey implements Informing {
    public String retrieveInfo() {
        return "Donkeys are simple.";
    }
}
```



Když třída implementuje rozhraní, musí implementovat *všechny* jeho metody!

Typ rozhraní namísto třídy

```
public void printInfo(Informing informing) {  
    System.out.println("Now you learn the truth!");  
    System.out.println(informing.retrieveInfo());  
}
```

```
Person p = new Person();  
printInfo(p);  
Donkey d = new Donkey();  
printInfo(d);
```

- Parametr metody je typu rozhraní, můžeme tam tedy použít *všechny třídy, které ho implementují*.
- To je velice časté a užitečné používat jako typ parametru rozhraní.
- Je z toho pak dobře vidět, že daný objekt používáme jako instanci implementující určité rozhraní a konkrétní třída nás u použití nezajímá.

"Přetypování" obecně

- Java má podobně jako většina jiných jazyků operátor pro "přetypování" (*type cast*).
- Píše se buďto a) (*typ*) *hodnota* nebo b) (*typ*) *odkaz_na_objekt* podle toho, co "přetypováváme".
- Jeho fungování se *zásadně* liší podle toho, zda jde o a) o primitivní hodnotu nebo b) odkaz na objekt.

Typová kontrola odkazů na objekty

- U "přetypování" odkazů na objekty se ve skutečnosti nejedná o *změnu obsahu objektu*,
- nýbrž pouze o *potvrzení*, že běhový typ objektu je ten požadovaný, na nějž "přetypováváme".
- Tj. jde o *běhovou typovou kontrolu*.
- Sdělujeme překladači "Hele, já vím, že jde o osobu, tak s tím pracuj jako s osobou".
- Když běhová kontrola případně selže (tedy nikoli v době překladu, ale až při spuštění), je vyvolána výjimka a běh programu přerušen.

Příklad - typová kontrola odkazů

```
Person p = new Person();  
// OK, p je třídy Person a ta implementuje Informing.  
Informing i = (Informing) p;  
// Rovněž OK, protože jakýkoli objekt je
```

```
// (aspoň nepřímým) potomkem Object.  
Object o = (Object) i;
```

Zjištění typu

Nejsme-li si typem jisti, otestujeme ho:

```
if(o instanceof Person p) {  
    // now we know p is a Person, so we can call walk()  
    p.walk();  
}
```

Přetypování u primitivních typů

- U *primitivních typů* se jedná o skutečný převod hodnoty z původního typu na cílový, o *typovou konverzi*.
- Např. `long` přetypujeme na `int` a ořeže se tím rozsah.
- Korektnost a případnou ztrátu informace za nás pohlídá v některých případech překladač, který samozřejmě zná obecné vlastnosti typů. Projeví se zejména při přetypování čísel.

Proměnná deklarovaná jako rozhraní

Můžeme taky vytvořit proměnnou *deklarovaného* typu rozhraní:

```
public class Person implements Informing {  
    public String retrieveInfo() {  
        return "People are clever.";  
    }  
    public void emptyMethod() { }  
}
```

Proměnná deklarovaná jako rozhraní

```
Informing i = new Person();  
i.retrieveInfo(); // ok  
i.emptyMethod(); // cannot be done
```

- Proměnná `i` může používat pouze metody deklarované v rozhraní.
- Nevidíme ostatní metody v třídě `Person`.

Příklad z reálného světa

- máme různé tiskárny, různých značek
- nechceme psát kód, který ošetří všechny značky, všechny typy
- chceme použít i budoucí značky/typy

Vytvoříme pro všechny jedno uniformní rozhraní

```
public interface Printer {  
    void printDocument(File file);  
    File[] getPendingDocuments();  
}
```

- náš kód bude používat tohle rozhraní, každá tiskárna která ho implementuje, bude kompatibilní
- budoucí tiskárny, které rozhraní implementují, ho budou moci používat také

Anotace `@Override`

- Pro jistotu, že *přepisujeme metodu předepsanou rozhraním* a ne jinou, použijeme znovu anotaci `@Override`:
- Používejme to!

```
public class Person implements Informing {  
    @Override  
    public String retrieveInfo() {  
        return "People are clever.";  
    }  
}
```

Replit demo k rozhraním

- <https://replit.com/@tpitner/PB162-Java-Lecture-04-interfaces-Shapes>