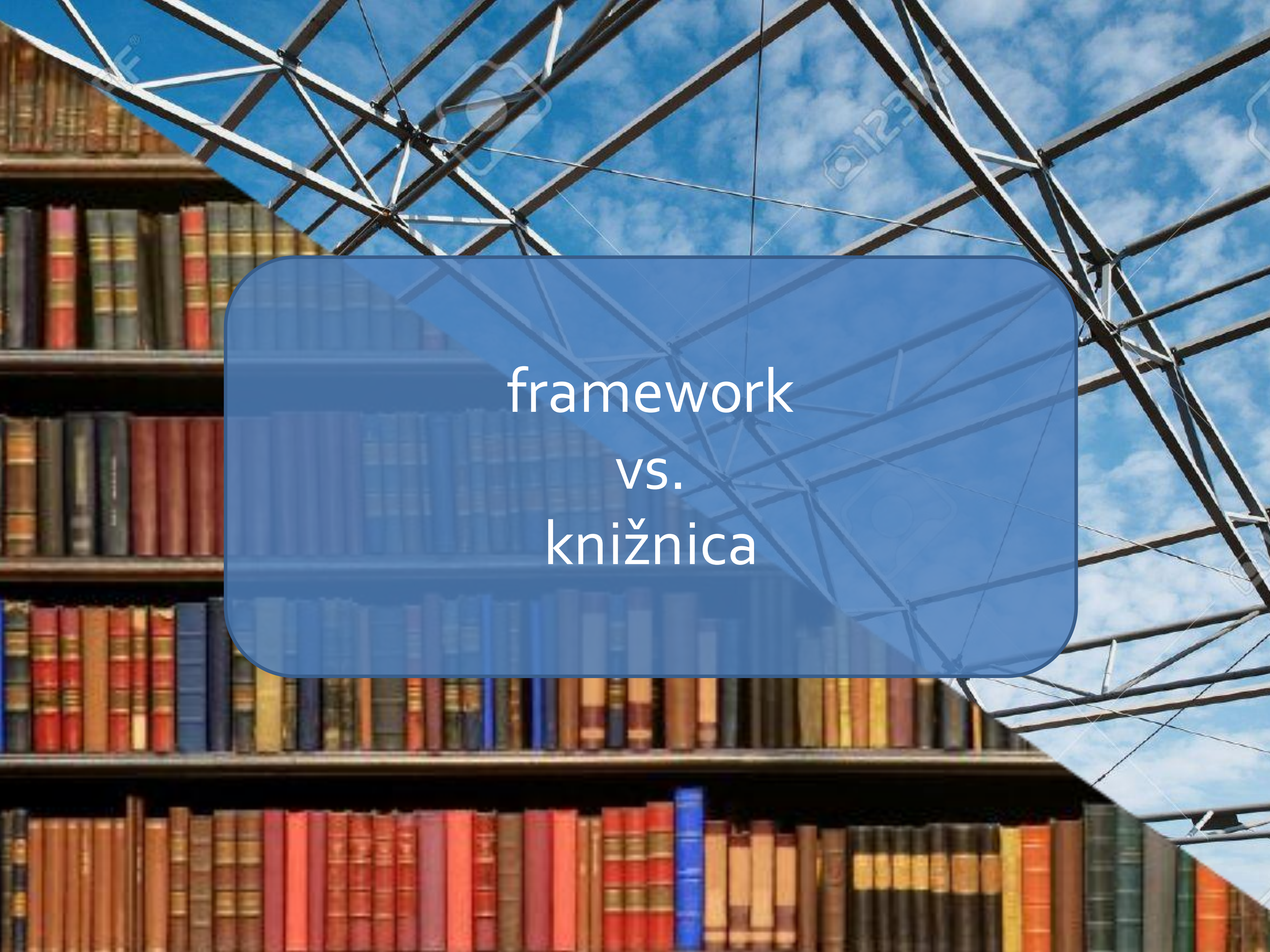
@
λ
<T>
CANNABIS
Metaprogramovanie, lambdy, typové parametre
UINF/PAZ1c
epizóda 8
LSD


framework
vs.
knižnica

Knižnica vs. Framework

- Knižnica – množina tried a ich metód, **ktoré vytvoríme/zavoláme** a oni pre nás niečo spravia
- Framework – program, ktorému podhodíme množinu tried a metód a **on ich vytvorí/zavolá**
  - To, čo mu podhadzujeme, musí byť v tvare, ktorému framework rozumie
  - Typicky označíme tie triedy a metódy, ktoré má použiť, cez anotácie

Metaprogramovanie

- Metaprogram
  - pracuje s iným programom
  - pracuje sám so sebou (reflexia)
- Prekladače/interpretery, Lombok, ObjectInspector z JPAZ2

- Reflexia je zakorenená v Jave
  - JavaFX, Spring
  - Prakticky každá aplikácia používa reflexiu

ANOTÁCIE, REFLEXIA

Anotácie a.k.a. zavináče

- = Metadáta o programe
- Niektoré z nich ste už videli
- @Override
  - označuje prekrývajúcu metódu, resp. metódu ktorá implementuje abstraktnú metódu
- @Test
  - Metóda, predstavujúca unit test
- @SuppressWarnings("unchecked")
  - označuje metódu, v ktorej sa nemá upozorňovať napr. na
    - neuvedenie vnútorného typu generickej triedy, alebo
    - neisté pretypovania

```
List zoznam = new ArrayList();
```

Použitie anotácií

- Informácia pre IDE
  - IDE nás upozorňuje, ak si myslíme, že prekrývame (@Override) a pritom neprekrývame
  - IDE potlačí upozorňovania (warnings)
- Spracovanie počas kompilácie / nasadzovania
  - Rôzne nástroje môžu generovať nový kód, súbory, dokumentáciu... - napr. Lombok
- Spracovanie počas behu programu - **reflexia**
  - Schopnosť programu manipulovať sám so sebou
  - Program si nájde anotované triedy, metódy, parametre metód a môže
    - vyrábať ich objekty,
    - volať na nich metódy (až počas behu zistí, aké majú meno)

Vypisovač entít cez anotácie

- Vytvoríme si anotáciu **@DaoGetter** na metódy, ktoré vracajú Dao objekty
- Vytvoríme si anotáciu **@EntityLister** na metódy, ktoré vracajú všetky entity
- Uvedieme tieto anotácie nad metódy, ktoré chceme poskytnúť vypisovaču
- Vypisovač si vo Factory nájde nejaké gettery Dao objektov a na každom Dao objekte si zavolá metódy na vrátenie entít a vypíše ich na konzolu

Anotácie a frameworky

- Prakticky všetky frejmworky používajú metaprogramovanie
- JavaFX - @FXML
- Spring – webové služby
  - ukážeme si o týždeň
  - hlavná téma predmetu Projekt1
  - @JpaRepository, @Service, @Controller, @Autowired, ...
  - ~90% Java aplikácií vo firmách používa Spring

To ako už nebudem vytvárať objekty a spúšťať na nich metódy?
With it
You shall deal

λ

LAMBDA VÝRAZY

Lambda výrazy

- Lambda kalkulus: formálny výpočtový model od r. 1936
  - Iné modely: čiastočne rekurzívne funkcie, Turingove stroje
- Základný rozdiel je v uvažovaní
  - procedurálny prístup : funkcii podhodíme dáta cez vstupné parametre a spustíme ju a vráti výsledok
  - Lambdy = funkcionálny prístup : dátam podhodíme funkciu a tá sa na nich vykoná a vráti nejaké dáta
- Dátam podhadzujeme pomenované alebo anonymné funkcie

```
For študent: študenti
  študent.dajÁčko.potľapkajPoPleci.pošliDomov
```

```
študenti.(s -> s.setZnámka('A')).(s -> potľapkajPoPleci)...
```

Lambdy v jave

- Rozhranie vieme implementovať triedou

```
public interface Printer {
   void print(String s);
}
```

```
public class TerrorPrinter implements Printer {
   @Override
   public void print(String s) {
      System.out.println(s + " is a crime against humanity!");
   }
}
```

Lambdy v jave

- Rozhranie vieme implementovať aj **anonymnou** triedou

```
public interface Printer {
    void print(String s);
}
```

```
var evilPrinter = new Printer() {
    @Override
    public void print(String s) {
        System.out.println(s + " is evil");
    }
};

evilPrinter.print("Wearing socks with sandals");
```

Lambdy v jave

- Rozhranie s **jednou metódou** vieme implementovať aj **lamba výrazom**

```
public interface Printer {
   void print(String s);
}
```

```
Printer goodPrinter = s -> System.out.println(s + " is good");

goodPrinter.print("Bacon with eggs");
```

Closure (uzáver)

- lambda výraz, ktorý používa premennú z "vonku"

```
int counter = 0;
Printer countedPrinter = s -> {
    counter++;
    System.out.println(s +
        " (This method was called " + counter + " times)");
};
```

Closure (uzáver)

- lambda výraz, ktorý používa premennú z "vonku"

```
int counter = 0;
Printer countedPrinter = s -> {
    counter++;
```

```
);
```

Tento príklad nefunguje, pretože Java má iba obmedzenú podporu uzáverov

"Premenné z vonku sa nesmú v lambde ani vo vonkajšej metóde meniť. (*effective final*)"

Je to obmedzenie typového systému Javy.
Iné jazyky to majú urobené lepšie.

Closure (uzáver)

- Špinavý trik
  - Premennú v uzávere síce nemôžeme meniť
  - Ale môžeme meniť jej "vnútro"
    - Prvky poľa
    - Inštančné premenné komplexného objektu

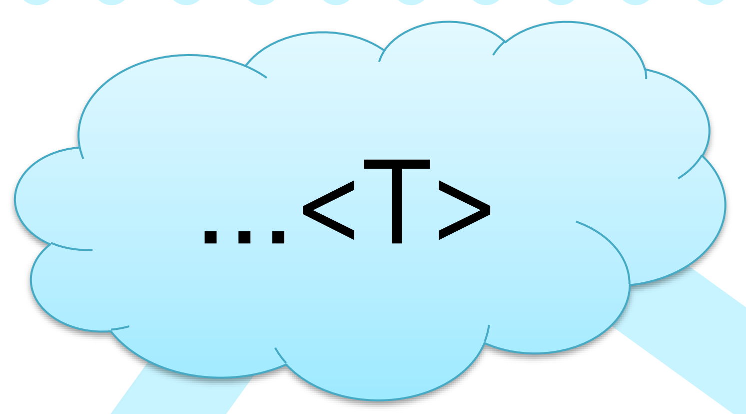
```
int[] counter = {0};

Printer countedPrinter = s -> {
   counter[0]++;
   System.out.println(s +
      " (This method was called " + counter[0] + " times)");
};
```

Java Stream API

Prúdy

- na volanie nejakých funkcií pre každý prvok vyrobíme prúd prvkov: kolekcia**.stream()**

- **.filter(**Predicate p**)** – ak p vráti true prvok ide do výstupného prúdu
- **.map(**Function f**)** – f vracia pre každý prvok prúdu nejaký prvok pre výstupný prúd
- **.reduce(**BinaryOperator bo**)** - vezmeme prvé dva prvky a nahradíme ich jedným, potom ho vezmeme a spolu s ďalším urobíme jeden, ...
- **.sorted(**Comparator c**)** – výstupný prúd bude utriedený
- **.collect(**Collector c**)** – prúd prerobí na výsledný objekt (najčastejšie kolekciu)
- **.forEach(**Consumer c**)** – pre každý prvok vykoná akciu a nevráti nič



GENERICKÉ TRIEDY

Kontajnery, krabice, kolekcie

- Špeciálne prípady asociácie
  - Objekt triedy funguje ako kontajner pre objekty inej triedy
  - Funkcionalita je rovnaká, len vie byť realizovaná nad rozličnými dátovými typmi
- Nechceme zoznam nejakých objektov, ale zoznam používateľov
- Nechceme porovnávať nejaké dva objekty, ale dva reťazce

Generická trieda

- trieda je charakterizovaná dvoma zložkami typ **„vonkajška“** + typ **„vnútra“**

```
List<Zviera> zvierata = new ArrayList<Zviera>();
```

```
class Porovnavac extends Comparator<User> { .. }
```

- Trieda ArrayList sa nazýva generická
- Rozhrania List a Comparator sú tiež generické
- Typ **„vnútra“** = typový parameter

Generická trieda Uzol v strome

- Čo sa môže nachádzať v uzle?
  - niekedy celé číslo,
  - inokedy reťazec,
  - alebo lokalita,
  - alebo XML tag
- V celom strome budeme pracovať iba s jedným dátovým typom (napr. strom lokalít)
  - typ uzlov si vyberie ten, kto ten strom vyrobí
  - podmienka pre to, aby Uzol bol dobrým kandidátom pre generickú triedu

Uzol<T>

```
public class Uzol<T> {
   private T data;
   private Uzol<T> lavy;
   private Uzol<T> pravy;

   public T getData(){
      return data;
   }
...
}
```

T = ľubovoľný dátový typ,
určí sa pri vytváraní inštancie:

```
Uzol<Lokalita> u = new Uzol<Lokalita>();
```

Použitie generických tried

- Ak nepoužijeme generiká, musíme pracovať s typom Object namiesto typom T.

```
String lokalita = (String) uzol.getData();
```

- Vždy, keď máte nutkanie vytvoriť inštančnú premennú typu Object, uvažujte nad generickým typovým parametrom.

Pozor na dedičnosť

- Vieme že ArrayList implemetuje List
  - a ak k tomu aj Mesto dedí od Lokality

~~List<Lokalita> lokality = ArrayList<Mesto>();~~

lokality.add(new Okres("Košice - okolie"));

Okres tiež je lokalita

ArrayList očakáva, že mu do metódy add pripláva mesto

Porovnávajúce sa dáta v uzle

- Niekedy potrebujeme, aby vnútorný typ spĺňal nejaké predpoklady
  - implmentuje dané rozhranie, alebo
  - rozširuje nejakú konkrétnu triedu

```
public class Uzol<T extends Comparable<T>> {
  private T data;

  public boolean isGreaterThan(Uzol<T> other) {
    return (data.compareTo(other.data) > 0);
  }
…
}
```

Vieme zavolať, lebo nech už bude v uzle čokoľvek, bude to mať metódu compareTo

Otázky?