Jedu li krave travu?

Zlatko Sirotić, univ.spec.inf.
ISTRA TECH d.o.o., Pula

ISTRA TECH d.o.o. & me

- ISTRA TECH je novo ime (od 2015.) poduzeća **Istra informatički inženjering**, osnovanog 1990. godine.
- Radim na informatičkim poslovima od 1984. godine.
- Oracle softverske alate (baza, Designer CASE, Forms 4GL, Reports, Java) koristim oko 25 godina.
- Objavljivao sam stručne radove na kongresima / konferencijama HrOUG, JavaCro, CASE, KOM, "Hotelska kuća", te u časopisima "Mreža", "InfoTrend" i "UT".
- Neka moja programska rješenja objavljivana su na web stranicama firmi Oracle i Quest (danas dio firme Dell).
- Od 2012. sam vanjski suradnik na Fakultetu informatike Pula.
- 2020. godine nije se održao HrOUG, pa mi je najznačajniji osobni događaj bila uspješna operacija srca (2 premosnice). **Veliko hvala svima u KBC Rijeka!**

HrOUG konferencija & me

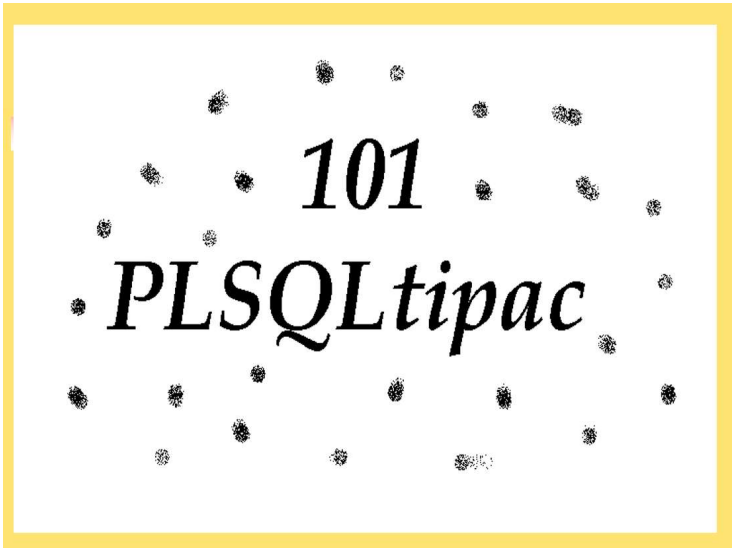
- Prvi put predavač na HrOUG 2002. (jedino održano u Puli).
- Sudjelovao sam 17 puta do sada (uključujući ovu godinu; 2016. i 2017. izostao zbog bolesti) i održao 22 predavanja.
- Uz 22 prezentacije, za HrOUG sam napisao 15 radova, ukupno preko 300 str. teksta.
- Moje prvo predavanje iz 2002. zvalo se **TRI IZ III**

(tri PL/SQL rješenja iz poduzeća Istra informatički inženjering):

1. Rješavanje "COMMIT poslovnih pravila" na Oracle bazi
2. Odgođena deklarativna ograničenja baze i Forms
3. Simulacija ROLLBACK TO SAVEPOINT ponašanja u okidaču baze

HrOUG konferencija & me

- 2003. godine imao sam predavanje **ČETIRI IZ III**, a PL/SQL tip broj 3 **"Mogući problemi kod istovremenog korištenja preopterećenja metoda i polimorfizma"** bio je začetak današnje teme.
- 2004. sam odlučio da predavanje ne nazovem **PET IZ III**, već **101 PLSQLtipac** (101 je binarno 5):



- Desilo se da je moje predavanje dobilo redni broj R101. U Zborniku radova konferencije je onda greškom, umjesto R101 101 PLSQLtipac, tiskano R101 **PLSQLtipac** ☺

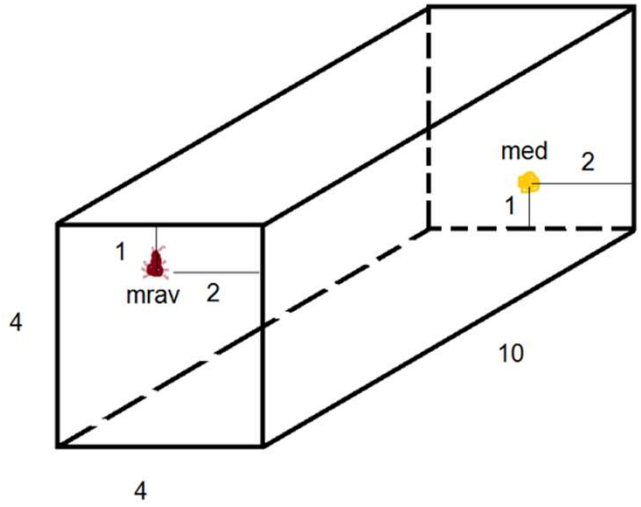
HrOUG predavanja iz prošlog desetljeća

- **2011. a) Konkurentno programiranje – Oracle baza, Java, Eiffel (najbolje ocijenjeno predavanje 16. konferencije)**
- 2011. b) Kriptografija u Oracle bazi
- 2012. a) Ima neka loša veza - priča o in-doubt distribuiranim transakcijama
- 2012. b) Visoka konkurentnost na JVM-u
- **2013. Transakcije i Oracle - baza, Forms, ADF (63 stranice – od tada sam odustao od pisanja radova :)**
- 2014. Nasljeđivanje je dobro, naročito višestruko - Eiffel, C++, Scala, Java 8
- 2015.a) Povratak u Prolog - verzija 1
- 2015 b) Kada Oracle naredba nije serijabilna?
- 2018. Testiranje konkurentnih transakcija
- 2019. Funkcijska paradigma i baze podataka

Mrav i med na prizmi (i valjku)

- Na HrOUG 2015. prvi put sam spomenuo zadatak Mrav i med na prizmi (verzija 0), unutar predavanja Povratak u Prolog.

  Spomenuo sam ga (verzija 1) i 2019. na početku predavanja Funkcijska paradigma i baze podataka.



- Na stranici http://www.istratech.hr/category/blog/

nalazi se najnovija verzija (2), uz još neka predavanja:

- Mrav i med na prizmi - verzija 2
- Povratak u Prolog - verzija 2
- Strukturna složenost algoritama

Teme

- OOPL je (tiho, tiho) slavio 50 godišnjicu prije 4 godine!
- Kratko o povijesti programskih jezika
  C++,
  Eiffel,
  PL/SQL,
  Java,
  Scala
- Počnimo sa **PL/SQL++** (naziv predavanja s HrOUG 2005.)
- PL/SQL primjer paralelne hijerarhije klasa
- Java primjer
- C++ primjer
- Eiffel primjer
- Scala primjer

Klasa

- **Klasa**, a ne **objekt**, je osnovni element objektno-orijentiranih programskih jezika.
  Možda bi se trebali zvati COPL, umjesto OOPL ☺
- Klasa je dio programskog koda, ima osobine i modula i tipa.
  Pojednostavljeno se može reći da vrijedi formula:
  KLASA = MODUL + TIP
- Klasa definira podatke - **atribute** i ponašanje – **metode** (rutine, funkcije).
- Na temelju klase mogu se napraviti podklase, koje **nasljeđuje** (nad)klasu.
- Podklasa može imati nove atribute i metode, a može i **nadjačati** (overiding) nasljeđenu metodu.
- Jedan od "Algol-oidnih" jezika bio je Simula 1, koji je bio namijenjen uglavnom za simulacije. Kasnija verzija, **Simula 67** nastala je 1967. godine u Norveškoj, a autori su Kristen Nygaard i Ole-Johan Dahl - **prvi OOPL u povijesti!**

C++ (kratka povijest)

- C (autor je Dennis Ritchie), također Algol-ov potomak, nastao je 1970. kao jezik za sistemsko programiranje operativnog sustava UNIX. U isto vrijeme nastao je i Pascal, isto potomak Algol-a. Za većinu kasnijih programskih jezika možemo reći da (barem po sintaksi) pripadaju C ili Pascal "struji".
- Nadograđujući C sa objektno orijentiranim mogućnostima (uz zadržavanje kompatibilnosti), Bjarne Stroustrup je 1983. godine napravio C++. 1986. godine je objavio knjigu "The C++ Programming Language".
- Tokom vremena je C++ dobivao neke vrlo značajne mogućnosti, **koje na početku nije imao: višestruko nasljeđivanje, generičke klase (predloške), obradu iznimaka** (exceptions) i dr.
- 1997. godine donesen je ISO standard.
  U standardu C++11 uvedene su npr. i lambda funkcije.
  C++14, C++17 i C++20 donose dodatna poboljšanja.

Eiffel (kratka povijest)

- Eiffel je 1985. godine dizajnirao (a 1986. je napravljen prvi compiler) **Bertrand Meyer**, jedan od autoriteta na području OOPL-a.
- Eiffel je od početka je podržavao **višestruko nasljeđivanje, generičke klase, obradu iznimaka, garbage collection i metodu Design by Contract (DBC).**
- Kasnije su mu dodani **agenti, nasljeđivanje implementacije** (uz nasljeđivanje tipa) i metoda za konkurentno programiranje Simple Concurrent Object-Oriented Programming (**SCOOP**).
- U široj je javnosti daleko manje poznat nego C++ i Java, ali ga mnogi autoriteti smatraju najboljim OOPL jezikom. Eiffel je od 2005. godine ECMA standardiziran, a od 2006. ISO standardiziran.

PL/SQL (kratka povijest)

- U sklopu Oracle baze 6.0 pojavio se 1991. novi programski jezik PL/SQL, koji je napravljen kao proceduralna nadopuna (deklarativnog) programskog jezika SQL
- PL/SQL je napravljen na temelju programskog jezika **ADA 83**, koja nije bila Object Oriented Programming Language (OOPL). **ADA 95** je dobila neke OOPL osobine.
- Oracle je 1997. objavio 8.0 verziju baze i nazvao ju objektno-relacijskom. U skladu s tim, nadopunjen je i PL/SQL.
- Ta verzija baze, kao niti sljedeća verzija 8i (sa Javom unutar baze), nije imala neke važne objektne mogućnosti kao što su npr. nasljeđivanje, nadjačavanje metoda, polimorfizam.
- Oracle je to uveo 2001. godine u bazi 9i.
  Od tada nema značajnih novih OOPL mogućnosti
  u Oracle PL/SQL-u.

Java (kratka povijest)

- Java 1.0 se pojavila 1996. i (u pravo vrijeme!) reklamirana je kao jezik za Internet, čime je odmah stekla ogromnu slavu.
- Ironija sudbine je da su Java apleti, koji su 1996. godine uzdigli Javu na pijedestal, danas mrtvi.
- Počeci Jave sežu u 1992., kada se zvala Oak i bila namijenjena za upravljanje uređajima za kabelsku televiziju i slične uređaje.
- **Sami autori su rekli da je Java = C++--**, tj. da je to pojednostavljeni (u pozitivnom smislu) C++.
- Nije stoga čudno da Java i C++ imaju sličnu sintaksu. Međutim, Java nije podskup C++ jezika.
- Također, iako jednostavniji nego C++, Java nije baš jednostavan jezik.

Scala (kratka povijest)

- Programski jezik Scala kreirao je **Martin Odersky**, profesor na Ecole Polytechnique Fédérale de Lausanne (EPFL).
- Krajem 80-ih doktorirao je na ETH Zürich kod profesora Niklausa Wirtha (kreatora Pascala i Module-2).
- Nakon toga naročito se **bavio istraživanjima u području funkcijskih jezika**, zajedno sa kolegom Philom Wadlerom (jednim od dva glavna kreatora funkcijskog jezika Haskell).
- Kada je izašla Java, Odersky i Wadler su 1996. napravili jezik **Pizza** nad JVM-om. Na temelju projekta Pizza, napravili su 1997./98. **Generic Java (GJ)**, koji je uveden u Javu 5.
- Odersky je 2002. počeo raditi novi jezik Scala. Tako je nazvana kako bi se naglasila njena **skalabilnost**.
- Prva javna verzija izašla je 2003. Trenutačna verzija je 3.0, (izašla ove godine), značajan skok u odnosu na verziju 2.

PL/SQL klasa- specifikacija (deklaracija)

- PL/SQL klasa se može kreirati samo kao objekt baze (u određenoj shemi).
- PL/SQL klasa ima dva dijela: specifikaciju i tijelo. U specifikaciji se deklariraju atributi i metode, a u tijelu se definiraju metode.

```
CREATE OR REPLACE TYPE Zivotinja AS OBJECT (
  ime    VARCHAR2(20),
  visina NUMBER,
  -- konstruktor nismo niti trebali raditi,
  -- jer PL/SQL ima default konstruktor s parametrima
  CONSTRUCTOR FUNCTION Zivotinja
    (p_visina NUMBER, p_ime VARCHAR2) -- namjerno obrnuto
    RETURN SELF AS RESULT,
  MEMBER PROCEDURE prikazi_podatke
) NOT FINAL;
```

PL/SQL klasa - tijelo (definicija)


```
CREATE OR REPLACE TYPE BODY Zivotinja AS
  CONSTRUCTOR FUNCTION Zivotinja
    (p_visina NUMBER, p_ime VARCHAR2)
    RETURN SELF AS RESULT IS
  BEGIN
    ime := p_ime;
    visina := p_visina;
    RETURN;
  END;

  MEMBER PROCEDURE prikazi_podatke IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ('Ime: ' || ime || ' Visina: ' || visina);
  END;
END;
```

PL/SQL
- programski kod može biti i izvan klase

- PL/SQL nije "čisti" OOPL, pa može postojati programski kod koji se ne nalaze niti u jednoj klasi; npr. prethodnu klasu možemo koristiti u sljedećem tzv. neimenovanom bloku:

```
DECLARE
  l_z Zivotinja := NEW Zivotinja (20, 'Fifi');
BEGIN
  l_z.prikazi_podatke;
END;
Ime: Fifi Visina: 20
```

- Naredba za kreiranje PL/SQL klase dosta podsjeća na naredbu za kreiranje PL/SQL paketa.
- PL/SQL paket nema skoro nikakvih sličnosti sa Java paketom, npr. PL/SQL paket ne može sadržavati klase.
- Za razliku od PL/SQL paketa, PL/SQL klasa ne može imati privatne atribute ili metode, tj. svi atributi i metode su javni.

Jednostruko nasljeđivanje

- **Nasljeđivanje** je jedna od tri osnovne operacije računa klasa. Ostale dvije su agregacija i generičnost.
- **Podklasa** nasljeđuje atribute/metode od nadklase.

```
CREATE OR REPLACE TYPE Krava UNDER Zivotinja (
  daje_dnevno_mlijeka NUMBER,

  CONSTRUCTOR FUNCTION Krava
    (p_visina NUMBER, p_ime VARCHAR2,
     p_daje_dnevno_mlijeka NUMBER)
    RETURN SELF AS RESULT,

  OVERRIDING MEMBER PROCEDURE prikazi_podatke
) NOT FINAL;
```

Jednostruko nasljeđivanje

```
CREATE OR REPLACE TYPE BODY Krava AS
  CONSTRUCTOR FUNCTION Krava
    (p_visina NUMBER, p_ime VARCHAR2,
     p_daje_dnevno_mlijeka NUMBER)
    RETURN SELF AS RESULT IS
  BEGIN
    ime := p_ime; visina := p_visina;
    daje_dnevno_mlijeka := p_daje_dnevno_mlijeka;
    RETURN;
  END;

  OVERRIDING MEMBER PROCEDURE prikazi_podatke IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ('Ime: ' || ime || ' Visina: ' || visina ||
       ' Daje mlijeka: ' || daje_dnevno_mlijeka);
  END;
END;
```

Nadjačavanje (overriding) metoda

- PL/SQL ima ključnu riječ **OVERRIDING**, kojom programer eksplicitno kaže da **nadjačava** određenu metodu.
- Eiffel ima za to riječ **redefine**, Scala i C++11 imaju **override**.
- C++ prije C++11 i Java prije verzije 5 nisu koristili za to ključne riječi ili anotacije, nego se nadjačavanje izražavalo isključivo tako da se u podklasi deklarira metoda sa istom signaturom kao što je metoda u nadklasi.
- Ako je signatura različita, tada se radi o **preopterećenju (overloading) metoda**, a ne o nadjačavanju.
- PL/SQL, Eiffel, Scala, C++1 (ključne riječi) i Java 5 (anotacija) eksplicitan način izražavanja nadjačavanja smanjuje mogućnost programerske greške.

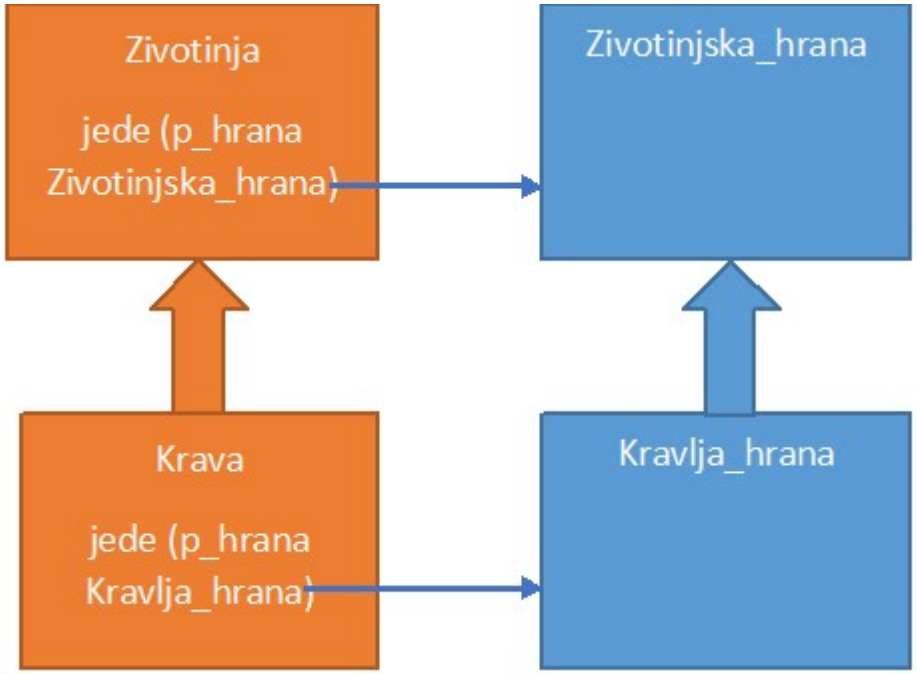
Primjer polimorfizma i dinamičkog (po)vezivanja metoda

```
DECLARE
  v_z Zivotinja := NEW Zivotinja (20, 'Fifi');
  v_k Krava := NEW Krava (150, 'Milka', 30);
BEGIN
  v_z.prikazi_podatke; -- poziva metodu iz Zivotinja
  v_k.prikazi_podatke; -- poziva metodu podklase Krava

/*v_k := v_z; -- compiler bi javio grešku */
  v_z := v_k; -- polimorfna dodjela
  v_z.prikazi_podatke; -- poziva metodu podklase!
END;
Ime: Fifi Visina: 20
Ime: Milka Visina: 150 Daje mlijeka: 30
Ime: Milka Visina: 150 Daje mlijeka: 30
```

Problem paralelne hijerarhije (povezanih) klasa

- Pretpostavimo da imamo klase **Zivotinja** i **Zivotinjska_hrana**, a klasa Zivotinja ima i proceduru **jede**, koja ima parametar tipa Zivotinjska_hrana.
- Sad želimo napraviti podklase **Krava** i **Kravlja_hrana** i u klasi Krava **promijeniti tip parametra** u proceduri jede iz Zivotinjska_hrana u Kravlja_hrana.



Problem paralelne hijerarhije (povezanih) klasa

- To je **promjena signature u nadjačanoj metodi**.

- Općenito, kod OOPL jezika postoje tri mogućnosti:

1. tip parametra se ne može mijenjati
   - **bez varijance (no variance)**
2. tip parametra može se mijenjati tako da bude podtip u odnosu na bazni tip – **kovarijanca (covariance)**
3. tip parametra može se mijenjati tako da bude nadtip u odnosu na bazni tip - **kontravarijanca (contravariance)**.

- Dakle, mi bismo htjeli primijeniti **kovarijancu**.

PL/SQL primjer paralelne hijerarhije

- Napravimo prvo **apstraktne** nadklase Zivotinjska_hrana i Zivotinja:

```
CREATE OR REPLACE TYPE Zivotinjska_hrana AS OBJECT (
  naziv VARCHAR2 (20),
  NOT INSTANTIABLE MEMBER FUNCTION naziv_hrane
    RETURN VARCHAR2
)
NOT INSTANTIABLE
NOT FINAL;

CREATE OR REPLACE TYPE Zivotinja AS OBJECT (
  ime   VARCHAR2 (20),
  NOT INSTANTIABLE MEMBER PROCEDURE jede
     (p_hrana Zivotinjska_hrana)
)
NOT INSTANTIABLE
NOT FINAL;
```

PL/SQL primjer paralelne hijerarhije

- Zatim napravimo podklase od Zivotinjska hrana
  – prvo podklasa Kravlja_hrana:

```
CREATE OR REPLACE TYPE Kravlja_hrana UNDER Zivotinjska_hrana (
  OVERRIDING MEMBER FUNCTION naziv_hrane RETURN VARCHAR2
)
NOT FINAL;

CREATE OR REPLACE TYPE BODY Kravlja_hrana AS
  OVERRIDING MEMBER FUNCTION naziv_hrane RETURN VARCHAR2 IS
  BEGIN
    RETURN 'Kravlja hrana: ' || naziv;
  END;
END;
```

PL/SQL primjer paralelne hijerarhije

- Zatim napravimo podklasu Riba (napomena: ovdje i u nastavku ćemo promatrati ribu isključivo kao vrstu hrane, iako je riba i životinja):

```
CREATE OR REPLACE TYPE Riba UNDER Zivotinjska_hrana (
  OVERRIDING MEMBER FUNCTION naziv_hrane RETURN VARCHAR2
)
NOT FINAL;

CREATE OR REPLACE TYPE BODY Riba AS
  OVERRIDING MEMBER FUNCTION naziv_hrane RETURN VARCHAR2 IS
  BEGIN
    RETURN 'Riba: ' || naziv;
  END;
END;
```

PL/SQL primjer paralelne hijerarhije

- Na kraju napravimo podklasu Krava, ali tako da u proceduri jede **ne pokušamo primijeniti kovarijancu** kod parametra p_hrana:

```
CREATE OR REPLACE TYPE Krava UNDER Zivotinja (
  OVERRIDING MEMBER PROCEDURE jede (p_hrana Zivotinjska_hrana)
)
NOT FINAL;

CREATE OR REPLACE TYPE BODY Krava AS
  OVERRIDING MEMBER PROCEDURE jede (p_hrana Zivotinjska_hrana)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ('Krava: ' || ime || ' jede ' || p_hrana.naziv_hrane);
  END;
END;
```

PL/SQL primjer paralelne hijerarhije

- Sljedeći kod pokazuje da kravi možemo dati kravlju hranu (npr. travu), ali i ribu (npr. srdelu), a onda je samo korak do kravljeg ludila :)

```
DECLARE
  v_krava  Krava          := NEW Krava ('Milka');
  v_khrana Kravlja_hrana  := NEW Kravlja_hrana ('Trava');
  v_riba   Riba           := NEW Riba ('Srdela');
BEGIN
  v_krava.jede (v_khrana);
  v_krava.jede (v_riba);
END;
```

```
Krava: Milka jede Kravlja hrana: Trava
Krava: Milka jede Kravlja hrana: Srdela
```

PL/SQL primjer paralelne hijerarhije

- Naravno, nismo zadovoljni prethodnim ponašanjem, pa **pokušavamo primijeniti kovarijancu**:

```
CREATE OR REPLACE TYPE Krava UNDER Zivotinja (
  OVERRIDING MEMBER PROCEDURE jede (p_hrana kravlja_hrana)
)
NOT FINAL;

0/0       PL/SQL: Compilation unit analysis terminated
2/21      PLS-00635: method does not override
```

- Dakle, **PL/SQL ne podržava kovarijancu**, a ne podržava niti kontravarijancu. Parametri u nadjačanoj proceduri moraju zadržati isti tip, tj. **PL/SQL ponašanje je bez varijance**.
- Preciznije, PL/SQL podržava kovarijancu samo kod povratne vrijednosti funkcije.

Java primjer paralelne hijerarhije

- Napravimo prvo apstraktne nadklase Zivotinjska_hrana i Zivotinja:

```
abstract class Zivotinjska_hrana {
  String naziv;
  abstract String naziv_hrane();
}

abstract class Zivotinja {
  String ime;
  abstract void jede(Zivotinjska_hrana p_hrana);
}
```

Java primjer paralelne hijerarhije

- Zatim napravimo podklase od Zivotinjska hrana – prvo podklasa Kravlja_hrana:
- Napomenimo da anotacija **@Override** postoji od Jave 5.

```
class Kravlja_hrana extends Zivotinjska_hrana {
  Kravlja_hrana(String p_naziv) {
    naziv = p_naziv;
  };

  @Override
  String naziv_hrane() {
    return "Kravlja hrana: " + naziv;
  }
}
```

Java primjer paralelne hijerarhije

- Zatim napravimo podklasu Riba:

```
class Riba extends Zivotinjska_hrana {
  Riba(String p_naziv) {
    naziv = p_naziv;
  };

  @Override
  String naziv_hrane() {
    return "Riba: " + naziv;
  }
}
```

Java primjer paralelne hijerarhije

- Na kraju napravimo podklasu Krava, ali tako da u proceduri jede **ne pokušamo primijeniti kovarijancu** kod parametra p_hrana:

```
class Krava extends Zivotinja {
  Krava(String p_ime) {
    ime = p_ime;
  };

  @Override
  void jede(Zivotinjska_hrana p_hrana) {
    System.out.println
      ("Krava: " + ime + " jede " +
       p_hrana.naziv_hrane());
  }
}
```

Java primjer paralelne hijerarhije

- Dobijemo isto ponašanje kao u PL/SQL-u. Sljedeći kod pokazuje da kravi možemo dati kravlju hranu (npr. travu), ali i ribu (npr. srdelu):

```
public class Test {
  public static void main(String[] args) {
    Krava          v_krava  = new Krava("Milka");
    Kravlja_hrana v_khrana = new Kravlja_hrana("Trava");
    Riba           v_riba   = new Riba("Srdela");

    v_krava.jede(v_khrana);
    v_krava.jede(v_riba);
  }
}
```

```
Krava: Milka jede Kravlja hrana: Trava
Krava: Milka jede Riba: Srdela
```

Java primjer paralelne hijerarhije

- Naravno, nismo zadovoljni prethodnim ponašanjem, pa **pokušavamo primijeniti kovarijancu, što ne uspijeva:**

```
class Krava extends Zivotinja {
  Krava(String p_ime) {
    ime = p_ime;
  };

  @Override
  void jede(Kravlja_hrana p_hrana) {
    System.out.println("Krava: " + ime + " jede " +
  p_hrana.naziv_hrane());
  }
}
Test.java:39: error: method does not override or
  implement a method from a supertype
  @Override
  ^
```

C++ primjer paralelne hijerarhije

- Napravimo prvo apstraktne nadklase Zivotinjska_hrana i Zivotinja:

```
#include <string>
#include <iostream>
using namespace std;

class Zivotinjska_hrana {
public:
  string naziv;
  virtual string naziv_hrane() = 0; // abstract method
};

class Zivotinja {
public:
  string ime;
  virtual void jede(Zivotinjska_hrana* p_hrana) = 0;
};
```

C++ primjer paralelne hijerarhije

- Primijetimo da u prethodnim klasama C++ traži ključnu riječ **virtual** za one metode koje u nastavku želimo nadjačati. U podklasama više nije nužno navoditi virtual, jer se to podrazumijeva, ali ga ovdje ipak eksplicitno pišemo.
- Zatim napravimo podklase od Zivotinjska hrana – prvo podklasa Kravlja_hrana.

```
class Kravlja_hrana : public Zivotinjska_hrana {
public:
  Kravlja_hrana(string p_naziv) {
    naziv = p_naziv;
  };

  virtual string naziv_hrane() override {
    return "Kravlja hrana: " + naziv;
  }
};
```

C++ primjer paralelne hijerarhije

- Ključna riječ **override** postoji od C++11 (2011.). Zatim napravimo podklasu Riba:

```
class Riba : public Zivotinjska_hrana {
public:
  Riba(string p_naziv) {
    naziv = p_naziv;
  };

  virtual string naziv_hrane() override {
    return "Riba: " + naziv;
  }
};
```

C++ primjer paralelne hijerarhije

- Na kraju napravimo podklasu Krava, ali tako da u proceduri jede **ne pokušamo primijeniti kovarijancu** kod parametra p_hrana:

```
class Krava : public Zivotinja {
public:
  Krava(string p_ime) {
    ime = p_ime;
  };

  virtual void jede(Zivotinjska_hrana* p_hrana) override
  {
    cout << "Krava: " << ime << " jede " <<
      p_hrana->naziv_hrane() << "\n";
  }
};
```

C++ primjer paralelne hijerarhije

- Dobijemo isto ponašanje kao u PL/SQL-u i Javi. Sljedeći kod pokazuje da kravi možemo dati kravlju hranu (npr. travu), ali i ribu (npr. srdelu):

```
int main() {
  Krava*          v_krava  = new Krava("Milka");
  Kravlja_hrana* v_khrana = new Kravlja_hrana("Trava");
  Riba*           v_riba   = new Riba("Srdela");

  v_krava->jede(v_khrana);
  v_krava->jede(v_riba);
}
```

```
Krava: Milka jede Kravlja hrana: Trava
Krava: Milka jede Riba: Srdela
```

C++ primjer paralelne hijerarhije

- Kad pokušamo napraviti kovarijancu u parametru procedure, dobijemo kod kompajliranja sličnu grešku kao u PL/SQL-u i Javi.

```
class Krava : public Zivotinja {
public:
  Krava(string p_ime) : Zivotinja(p_ime) {};
  virtual void jede(Kravlja_hrana* p_hrana) override {
    cout << "Krava: " << ime << " jede " <<
      p_hrana->naziv_hrane() << "\n";
  }
};
```

```
[Error] 'virtual void Krava::jede(Kravlja_hrana*)'
  marked override, but does not override
```

Eiffel primjer paralelne hijerarhije

- Za razliku od programskih jezika PL/SQL, Java i C++, koji kod nadjačavanja procedura ne dozvoljavaju kovarijancu kod parametara (dozvoljavaju kovarijancu samo kod povratne vrijednosti funkcije), Eiffel to dozvoljava.
- I ne samo kod parametara procedure – **Eiffel dozvoljava kovarijancu i kod nadjačanih atributa (u podklasama)**.
- U nastavku ćemo vidjeti primjer koji je sličan dosadašnjem, a napravljen je na temelju primjera iz 17. poglavlja knjige **Object-Oriented Software Construction** (Bertrand Meyer, 2. izdanje iz 1997., poznata pod skraćenicom OOSC2).
- Taj primjer, zajedno s primjerom iz Scale (koji će biti prikazan na kraju), bili su inspiracija za ovu prezentaciju.
- Scala primjer napravljen je prema knjizi **Programming in Scala** (Martin Odersky i dr., 3. izdanje iz 2016.).

Eiffel primjer paralelne hijerarhije

- Prvo napravimo dvije klase koje imaju sličnu ulogu kao što su prije imale klase Zivotinjska_hrana i Zivotinja:

```
class SOBA
end

class SKIJAS
feature
  soba: SOBA

  smjesti_u (p_soba: SOBA) is
    soba := p_soba
  end
end
```

- Vidimo da klasa SKIJAS ima atribut soba i parametar p_soba, koji su oba tipa SOBA.

Eiffel primjer paralelne hijerarhije

- Sad napravimo po dvije podklase prethodnih klasa.
  Ključna riječ **redefine** je kao PL/SQL / Java / C++ **override**.
  Primijetimo kovarijancu kod nadjačanih atributa i parametra:

```
class SOBA_ZA_DJEVOJKE inherit SOBA end
class SOBA_ZA_DJECAKE inherit SOBA end

class SKI_DJEVOJKA inherit SKIJAS redefine soba, smjesti_u end
feature
  soba: SOBA_ZA_DJEVOJKE
  smjesti_u (p_soba: SOBA_ZA_DJEVOJKE) is
    soba := p_soba
  end
end
class SKI_DJECAK inherit SKIJAS redefine soba, smjesti_u end
feature
  soba: SOBA_ZA_DJECAKE
  smjesti_u (p_soba: SOBA_ZA_DJECAKE) is
    soba := p_soba
  end
end
```

Eiffel primjer paralelne hijerarhije

- Unatoč kovarijanci, evo programskog koda koji će dovesti do onoga što se htjelo spriječiti (tzv. **catcall** problem):

```
class TEST
feature
  test is
  local
    djevojka: SKI_DJEVOJKA
    zenska_soba: SOBA_ZA_DJEVOJKE
    djecak: SKI_DJECAK
    skijas: SKIJAS
  do
    create djevojka
    create zenska_soba
    djevojka.smjesti_u (zenska_soba) -- djevojka u zenskoj sobi
    create djecak
 -- djecak.smjesti_u (zenska_soba) -- ne kompajlira, naravno
    skijas := djecak; -- polimorfna dodjela
    skijas.smjesti_u (zenska_soba) -- djecak je u zenskoj sobi
  end
end
```

Eiffel primjer paralelne hijerarhije

- Kako navodi Bertrand Meyer u OOSC2, općeniti primjeri s paralelnom hijerarhijom klasa mogli bi se rješavati primjenom **generičkih klasa**, koje imaju i Java (generics) i C++ (templates). No Meyer smatra da je primjena generičkih klasa za paralelne hijerarhije "preteško oružje", jer traži savršeno predviđanje kod modeliranja klasa.
- U OOSC2 Meyer daje tri moguća rješenja problema koji je naveden u prethodnom primjeru. Jedno rješenje je **globalna analiza sustava**, gdje bi se analizirao (tj. kompajlirao) cijeli sustav, a ne samo klase koje su mijenjane. Jer, programski kod može biti ispravan na razini pojedinačnih klasa, a neispravan na razini sustava. Naravno, kompajliranje svih klasa nakon promjene u jednoj klasi nije baš jako praktično.
- Na stranicama firme **eiffel.com** (vlasnik je Meyer) našli smo informaciju za EiffelStudio 14.05 (May 2014): **Compiler Covariance fix: new mechanism to avoid catcall at compile time** – ali, ne znamo detalje.

Scala varijanca generičkih parametara

- Scala ne dozvoljava varijancu parametara kod nadjačavanja procedura, ali ima **varijancu kod generičkih parametara u generičkim klasama**, što ima i Kotlin, a Java nema.
- Kod Scale, deklaracija npr. **List[+A]** označava **kovarijancu** (**kontravarijanca** se označava s **List[-A]**), što znači da je npr. **List[String] podtip od List[AnyRef]** - podudara se s time što je **String podtip od AnyRef** (kod kontravarijance, List[String] je nadtip od List[AnyRef]).
- Kotlin označava **kovarijancu s List<out T>**, a **kontravarijancu s List<in T>** (List<T> je bez varijance).
- Takav način deklariranja varijance (generičkih parametara) zove se **declaration-site variance**, što je puno bolje nego Java način, **use-site variance**.
- U Javi moguće probleme oko toga rješava korisnik library-a, a u Scali i Kotlinu pisac library-a, uz pomoć kompajlera.

Scala apstraktni tipovi

- Scala, uz apstraktne metode, ima i **apstraktne tipove** i apstraktne atribute (val i var):

```
abstract class Abstract {
  type T
  def transform(x: T): T
  val initial: T
  var current: T
}

class Concrete extends Abstract {
  type T = String
  def transform(x: String) = x + x
  val initial = "hi"
  var current = initial
}
```

Scala primjer paralelne hijerarhije

- Prvo napravimo apstraktnu klasu Zivotinjska_hrana sa apstraktnom metodom naziv_hrane (pa u podklasama ne trebamo pisati override) i podklase Kravlja_hrana i Riba:

```
abstract class Zivotinjska_hrana(naziv: String) {
  def naziv_hrane: String
}

class Kravlja_hrana(var naziv: String)
extends Zivotinjska_hrana(naziv) {
  def naziv_hrane: String = "Kravlja hrana: " + naziv
}

class Riba(var naziv: String)
extends Zivotinjska_hrana(naziv) {
  def naziv_hrane: String = "Riba: " + naziv
}
```

Scala primjer paralelne hijerarhije

- Onda napravimo apstraktnu klasu Zivotinja sa **apstraktnim tipom HRANA, ali ograničenim na tip Zivotinjska_hrana**.
- Na kraju i njenu podklasu Krava, tako da apstraktni tip HRANA zamijenimo sa konkretnim tipom Zivotinjska_hrana:

```
abstract class Zivotinja(ime: String) {
  type HRANA <: Zivotinjska_hrana
  def jede(p_hrana: HRANA)
}

class Krava(var ime: String) extends Zivotinja(ime) {
  type HRANA = Kravlja_hrana
  def jede(p_hrana: Kravlja_hrana) = // moze i HRANA
    println
      ("Krava: " + ime + " jede " + p_hrana.naziv_hrane)
}
```

Scala primjer paralelne hijerarhije

- Kompajler ne pušta da kravi damo ribu, niti direktno, niti nakon polimorfne dodjele:

```
object Test {
  def main(args: Array[String]) = {
    val v_krava  = new Krava("Milka")
    val v_khrana = new Kravlja_hrana("Trava")
    val v_riba   = new Riba("Srdela")
    v_krava.jede(v_khrana)
    v_krava.jede(v_riba) // greška kod kompajliranja
    val v_zivotinja = v_krava; // polimorfna dodjela
    v_zivotinja.jede(v_khrana)
    v_zivotinja.jede(v_riba) // greška kod kompajliranja
  }
}
... error: type mismatch;
 found   : Riba
 required: Kravlja_hrana
    v_krava.jede(v_riba) i v_zivotinja.jede(v_riba)
                 ^
```

Scala apstraktni tipovi : generičke klase

- Dakle, iako Scala ne dozvoljava kovarijancu parametara kod nadjačavanja procedura, ona ipak rješava problem paralelnih hijerarhija, pomoću apstraktnih tipova.
- Što kažu u knjizi **Programming Scala** (Dean Wampler i Alex Payne, 2. izdanje iz 2014., str. 368-369):

Technically, you could implement almost all the idioms that parameterized types support using abstract types and vice versa. *However, in practice, each feature is a natural fit for different design problems.*

Parameterized types work nicely for containers, like collections, *where there is little connection between the types represented by the type parameter and the container itself. For example, a list works the same if it's a list of strings, a list of doubles, or a list of integers…*

In contrast, abstract types tend to be most useful for type "families", types that are closely linked.

Umjesto zaključka

Za type klučnu reč,
Kaj dala si mi, Scala,
Kaj morem ti neg' reć:
Od vsega srca fala.