NASLJEĐIVANJE JE DOBRO, NAROČITO VIŠESTRUKO - Eiffel, C++, Scala, Java 8

Zlatko Sirotić, univ.spec.inf.

Istra informatički inženjering d.o.o.

Pula

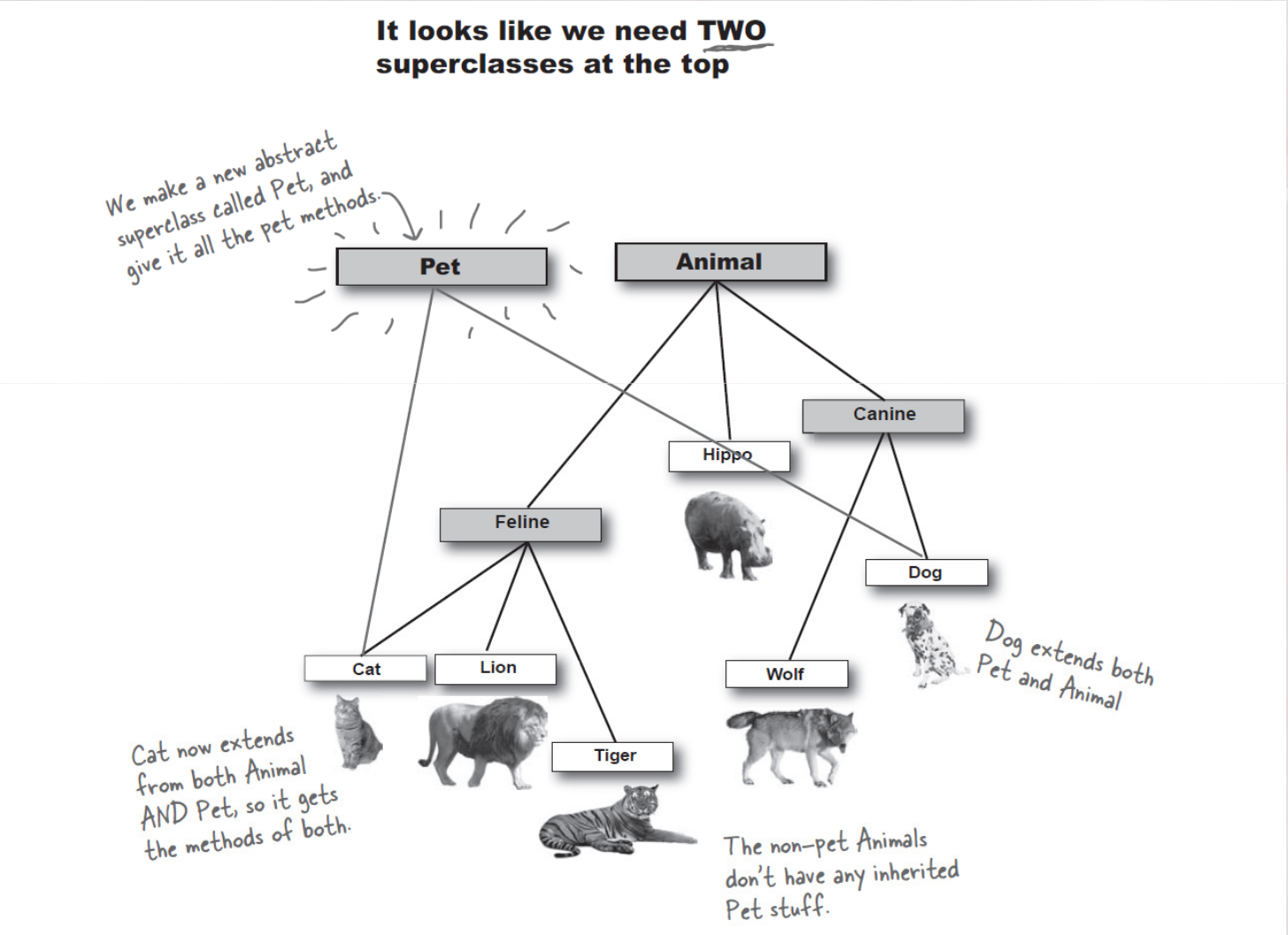
Autor je (bar neko vrijeme) radio s programskim jezicima / alatima:

- "Assembler" za Texas Instruments TI-58 kalkulator (1981.)
- Fortran, BASIC (1982.)
- Pascal, Assembler za Zilog Z80 (ZX Spectrum) (1983.)
- Cobol, dBASE III, Prolog (1984.)
- ADS (Application Development System (1985. – 1999.)
- **Oracle Database, Designer, Forms, Reports (1995. -)**
- Eiffel, C, C++ (1998. -)
- **Java (2003. -)**
- Scala (2012. -)
- **Oracle ADF (2013. -)**

Neki autorovi radovi zadnjih par godina

- CASE 2014: Trebaju li nam distribuirane (SQL) baze u vrijeme oblaka?
- JavaCro 2014: Da li postoji samo jedna "ispravna" arhitektura web poslovnih aplikacija?
- HrOUG 2013: Transakcije i Oracle - baza, Forms, ADF
- CASE 2013: Što poslije Pascala? Pa … Scala!
- HrOUG 2012 a: Visoka konkurentnost na JVM-u
- HrOUG 2012 b: Ima neka loša veza (priča o in-doubt distribuiranim transakcijama)
- CASE 2012 a: Utjecaj razvoja mikroprocesora na programiranje
- CASE 2012 b: Konkurentno programiranje u Javi i Eiffelu

Višestruko nasljeđivanje nam treba? ("Head First Java", Sierra K., Bates B., 2005.)



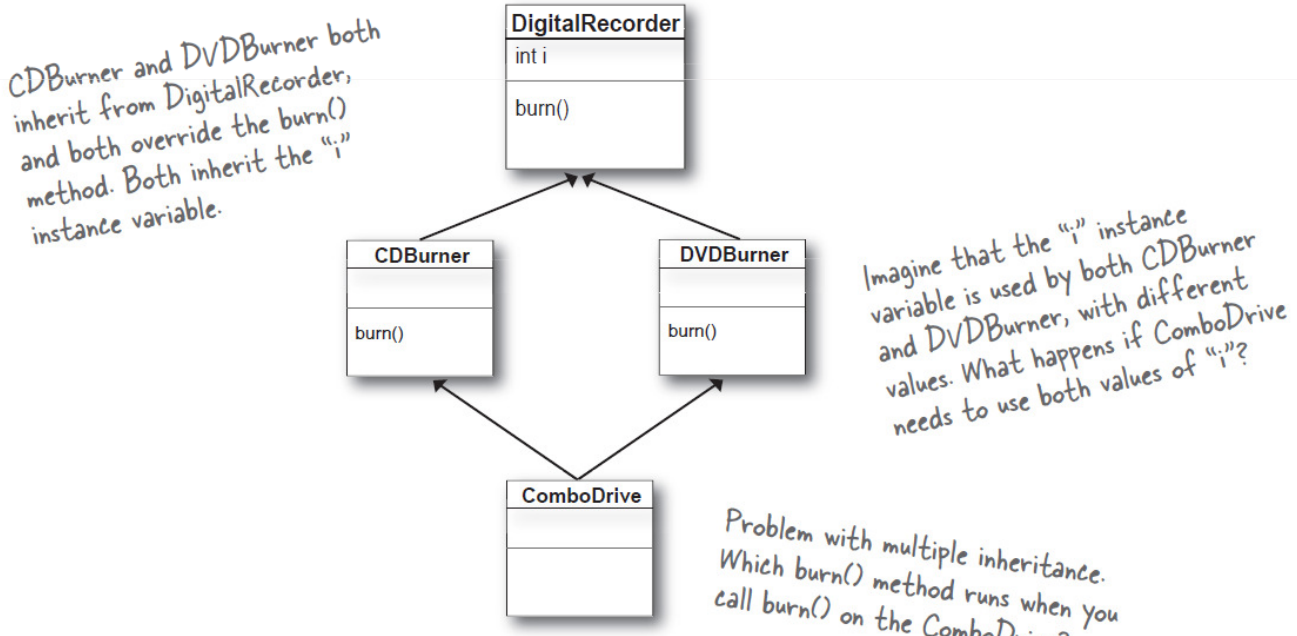
Višestruko nasljeđivanje je loše, zbog "smrtonosnog dijamanta"? (HFJ)

It's called "multiple inheritance" and it can be a Really Bad Thing.

That is, if it were possible to do in Java.

But it isn't, because multiple inheritance has a problem known as The Deadly Diamond of Death.

Deadly Diamond of Death



Što kažu neki drugi znanstvenici - Luca Cardelli u "A Semantic of Multiple Inheritance" (1988.)

- A class can sometimes be considered a subclass of two incompatible superclasses; then an arbitrary decision has to be made to determine which superclass to use. This problem leads naturally to the idea of multiple inheritance.

  Multiple inheritance occurs when an object can belong to several incomparable superclasses: the subclass relation is no longer constrained to form a tree, but can form a dag.

  Multiple inheritance is more elegant than simple inheritance in describing class hierarchies, but is more difficult to implement.

Što kažu neki drugi znanstvenici - Bertrand Meyer u "Object Oriented Software Construction" (1997.)

- Multiple inheritance is indispensable when you want to state explicitly that a certain class possesses some properties beyond the basic abstraction that it represents. Consider for example a mechanism that makes object structures persistent (storable on long-term storage)

  …

  The discussion of inheritance methodology will define it as inheritance of the structural kind. Without multiple inheritance, there would be no way to specify that a certain abstraction must possess two structural properties — numeric and storable, comparable and hashable.

  Selecting one of them as the parent would be like having to choose between your father and your mother.

Kratka povijest promatranih jezika - (pra)roditelji

- Sva četiri jezika imaju zajedničkog pretka – **Algol** (ALGOrithmic Language), kojeg su napravili Backus i Naur daleke (za informatiku) 1958. godine.
- Algol je ostavio ogroman utjecaj na programske jezike koji su došli nakon njega.
- Poznati informatičar C.A.R. Hoare duhovito je rekao: "Algol je bio toliko ispred svog vremena da je predstavljao poboljšanje ne samo u odnosu na sve svoje prethodnike, nego i u odnosu na skoro sve svoje nasljednike".
- Jedan od "Algol-oidnih" jezika bio je i Simula 1, koji je bio namijenjen uglavnom za simulacije.
  Kasnija verzija, **Simula 67** (nastala 1967. godine, a autori su Kristen Nygaard i Ole-Johan Dahl), je jezik opće namjene i to **prvi OOPL u povijesti!**

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,
  takav da 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). Zbog jednostavnije usporedbe četiri OOPL jezika, koristit ćemo pojmove "atribut" i "metoda".
- Svaki jezik ima svoj stil za pisanje imena klasa, atributa, metoda, parametara i lokalnih varijabli.
  Eiffel nije osjetljiv na velika/mala slova,
  dok C++, Java i Scala jesu (case-sensitive).

Nasljeđivanje klasa i nadjačavanje (overriding) metoda

- Nasljeđivanje je jedna od tri osnovne operacije računa klasa (class calculus). Ostale dvije su **agregacija i generičnost**.
- Klasa od koje se nasljeđuje je nadklasa, a klasa koja nasljeđuje je podklasa. Podklasa može imati nove metode i atribute (koje nadklasa nema), ali može i **nadjačati (overriding)** metode iz nadklase.
- Eiffel ima ključnu riječ **redefine,** a Scala **override,** kojom programer eksplicitno kaže da nadjačava određenu metodu.
- C++ i Java ne koriste ključne riječi (Java ima anotaciju), nego se nadjačavanje izražava tako da se u podklasi deklarira metoda sa istom signaturom kao što je metoda u nadklasi.
- Eiffel i Scala (eksplicitan) način izražavanja je bolji, jer smanjuje mogućnost programerske greške.

Objekti na gomili (heap) i objekti na stogu (stack)

- **Stog (stack) i gomila ili hrpa (heap)** su vrlo važne strukture podataka i za ne-OOPL i za OOPL.
- I kod ne-OOPL i kod OOPL se stog koristi za **spremanje lokalnih varijabli i argumenata** (neko kaže parametara) funkcija / procedura.
- **C++ i Eiffel mogu na stogu imati i objekte**, koji nestaju nakon što završi funkcija / procedura koja ih je kreirala. Naravno, mogu imati objekte i na gomili.
- **Java i Scala mogu imati objekte samo na gomili**, a na stogu Java može držati samo varijable primitivnog tipa, ili reference na objekte koji su na gomili (Java i Scala).
- C++ objekti na gomili se ne brišu automatski kad više nema referenci na njih, za razliku od Eiffela, Jave i Scale, gdje ih automatski briše **garbage collector.**

Polimorfizam i dinamičko povezivanje metoda (dynamic binding)

- "Pjesnički rečeno", polimorfizam omogućava da objektima različitog tipa pošaljemo istu poruku i da oni na tu poruku reagiraju na odgovarajući način. Riječ je o tome da možemo varijabli određenog tipa pridružiti ne samo varijablu istog tipa, nego i varijablu podtipa (**polimorfično pridruživanje**).
- Što će se desiti ako imamo neku metodu koja je nadjačana u podklasi i ako nakon polimorfičnog pridruživanja pozovemo tu metodu naredbom "varijabla_nadklase.metoda" - da li će se izvršiti verzija metode iz nadklase ili verzija iz podklase?
- Odgovor je – metoda iz podklase, zahvaljujući tzv. dinamičkom povezivanju - **dynamic binding** (ili dynamic method dispatch).
- Za C++ to vrijedi samo ako je metoda u nadklasi označena sa **virtual** i ako koristimo dinamičke objekte (na heapu).

Promjena tipa parametra kod nadjačavanja metoda

- Eiffel kod nadjačavanja metoda podržava tzv. **covariance** i za parametre metoda i za povratnu (return) vrijednost funkcije i za atribute, tj. oni ne moraju biti istog tipa kao u izvornoj metodi, već mogu biti podtipa.
- C++ ima **novariance** pristup za parametre, ali po Standardu od 1997. podržava **covariance za povratne vrijednosti funkcije**. Java se ponaša kao C++ od verzije Java 5. Scala ima isto ponašanje.
- Nekad je logičan izbor covariance, a nekad **contravariance,** pogotovo kod tzv. nasljeđivanja implementacije.
- Međutim, covariance i contravariance nad parametrima mogu dovesti i do određenih problema, koje kompajler treba pokušati spriječiti (npr. Eiffel kompajler).

1. Eiffel
- kratka povijest

- Eiffel je 1985. godine dizajnirao (a 1986. je napravljen prvi compiler) **Bertrand Meyer**, jedan od najvećih 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 danas najboljim OOPL jezikom. Eiffel je od 2005. godine ECMA standardiziran, a od 2006. ISO standardiziran.

1. Eiffel klasa – konstruktor ne mora imati isto ime kao klasa (u primjeru ipak ima)

```
class ZIVOTINJA
create zivotinja
feature {ANY}
  ime:         STRING
  visina_cm: DOUBLE
  zivotinja (p_ime: STRING; p_visina_cm: REAL) is
    do ime := p_ime; visina_cm := p_visina_cm end
  prikazi_podatke is
    do
      print ("  Ime: "); print (ime); ...
    end
  visina_inch: REAL is
    do Result := visina_cm / 2.54 end
end
```

1. Design by Contract metoda

- Autor **Design by Contract (DBC) metoda** je također Meyer.
- Pojednostavljeno rečeno, DBC se zasniva na ideji da svaka **metoda** (procedura ili funkcija), uz "standardni" programski kod, treba imati još dva dodatna dijela - **pretkondiciju** (precondition) i **postkondiciju** (postcondition).
- **Klasa** treba imati još **invarijantu** (invariant).
- Ugovor se zasniva na tome da metoda "traži" od svog pozivatelja (neke druge metode) da zadovolji **uvjete definirane u pretkondiciji plus uvjete definirane u invarijanti**, a ona (pozvana metoda) se tada "obvezuje" da će na kraju zadovoljiti **uvjete definirane u postkondiciji plus uvjete definirane u invarijanti**.

1. Eiffel i DBC

- U Eiffelu metoda ne može obraditi iznimku koja se desila zbog nezadovoljavanja pretkondicije – jednostavno, **pozvana metoda nema što raditi ako se druga metoda (pozivatelj) ne drži ugovora**!
- Za pojavu iznimke u vrijeme izvršavanja programskog koda pretkondicije "krivac" je metoda-pozivatelj, dok je **za pojavu iznimke u vrijeme izvršavanja programskog koda postkondicije "krivac" metoda-izvršavatelj**.
- Kod nasljeđivanja, **nadjačana metoda mora imati jednaku ili slabiju pretkondiciju** (tj. može zahtijevati od metode koja ju je pozvala ili isto što i metoda nadklase, ili manje od toga) i mora imati **jednaku ili jaču postkondiciju** (tj. mora osigurati barem ono što je osiguravala metoda nadklase).

1. Eiffel i DBC

```
class interface STACK [G] ...
remove is
   require not_empty: not empty
   ensure not_full: not full
          one_fewer: el_count = old el_count-1
end
invariant
   count_non_negative: 0 <= el_count
   count_bounded: el_count <= capacity
   empty_if_no_elements: empty = (el_count = 0)
end -- class interface STACK [G]
```

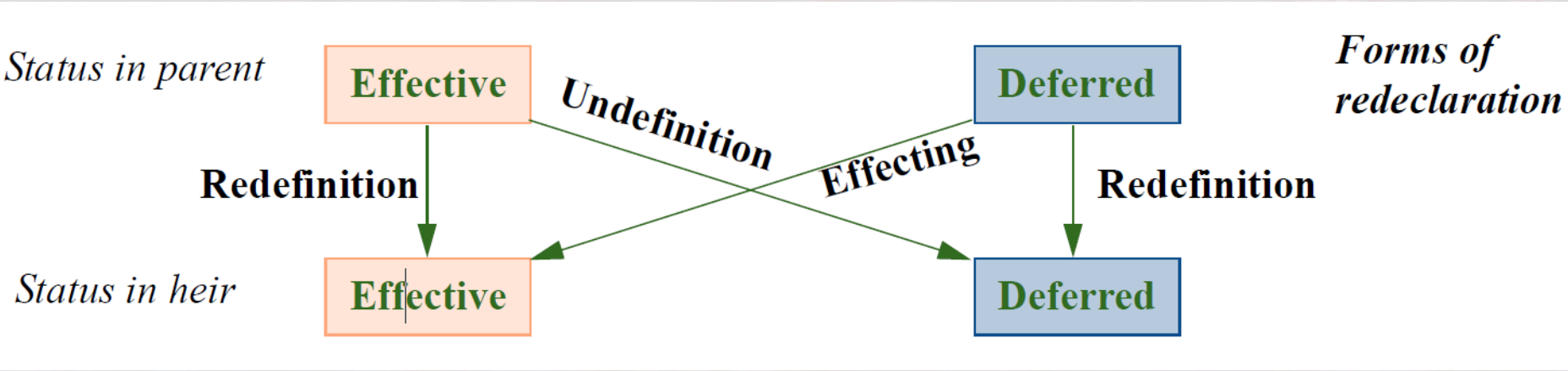
1. Eiffel i SCOOP metoda

- Osnove SCOOP metode Meyer je prikazao još 1990. godine, detaljan prikaz dao je 1997. Eksperimentalno je realizirana i poboljšavana na ETH Zurich.
- Nedavno je (lipanj 2011.) firma Eiffel Software uključila SCOOP metodu u svoj proizvod EiffelStudio v.6.8.
- Moglo bi se reći da je **SCOOP zaseban (mini) jezik za konkurentno programiranje**, jer eksperimentalne implementacije postoje npr. i za jezik Java.
- SCOOP se i dalje razvija, npr. istražuje se:

- prevencija i detekcija deadlocka;
- uvođenje softverske transakcijske memorije;
- distribuirani SCOOP.

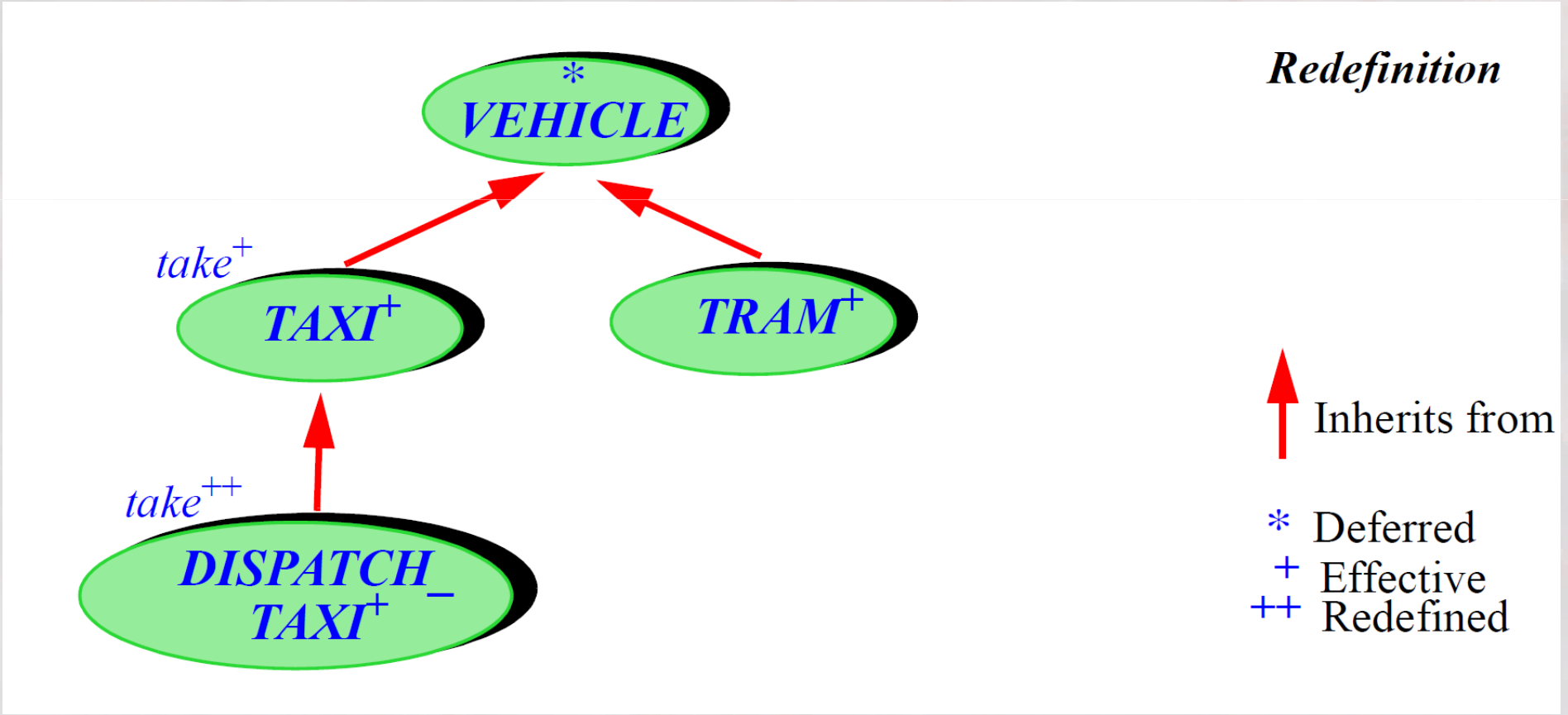
1. Eiffel - različiti načini redeklariranja nadjačane metode u podklasi

- I efektivna (effective) i apstraktna (deferred) metoda nadklase se može **redefinirati (redefinition)** u podklasi.
- Apstraktna metoda iz nadklase može se u podklasi **učiniti efektivnom (effecting)**.
- Suprotno tome, efektivna metoda iz nadklase može se u podklasi **učiniti apstraktnom (undefinition)**.



1. Eiffel – BON grafički prikaz različitih načini redeklariranja nadjačane metode u podklasi

- **BON** (Bussines Object Notation; Jean-Marc Nerson i Kim Waldén, 1989.) grafička notacija (kao UML, ali jednostavnija):

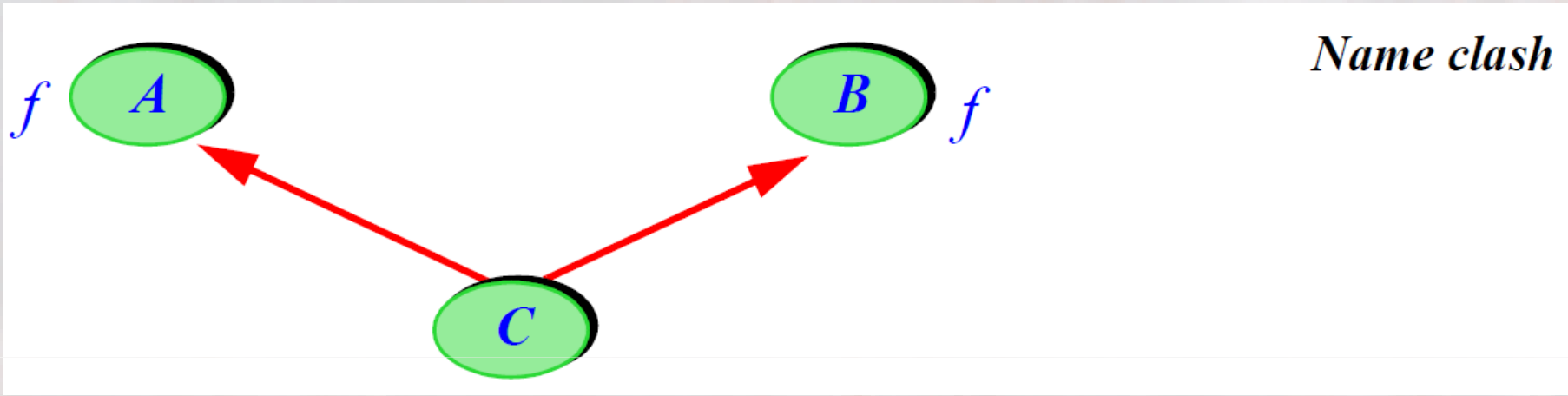


1. Eiffel
- primjer višestrukog nasljeđivanja

- U primjeru klasa TROLLEY nasljeđuje klase TRAM i BUS, pri čemu redefinira metode add_station i remove_station iz klase TRAM:

```
class TROLLEY inherit
   TRAM
      redefine add_station, remove_station end
   BUS
feature
   ...
end
```

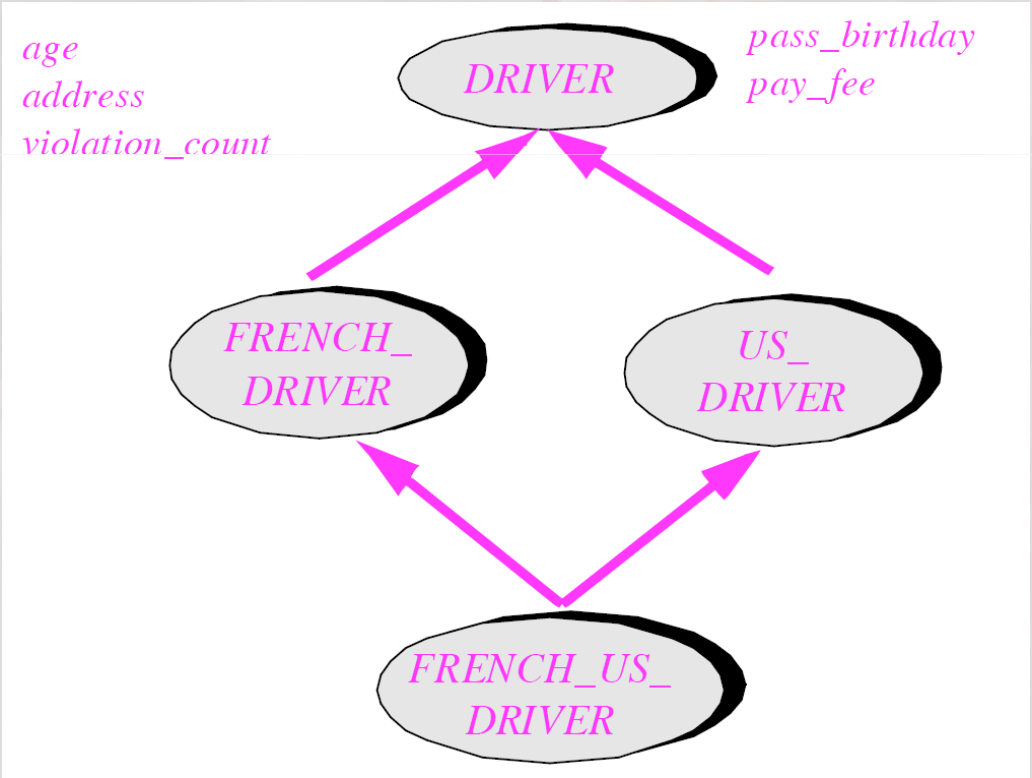
1. Eiffel – primjer nasljeđivanja dvije klase koje imaju metodu istog imena; koristi se rename




```
class C inherit
   A
      rename f as first_f end
   B
feature
   ...
end
```

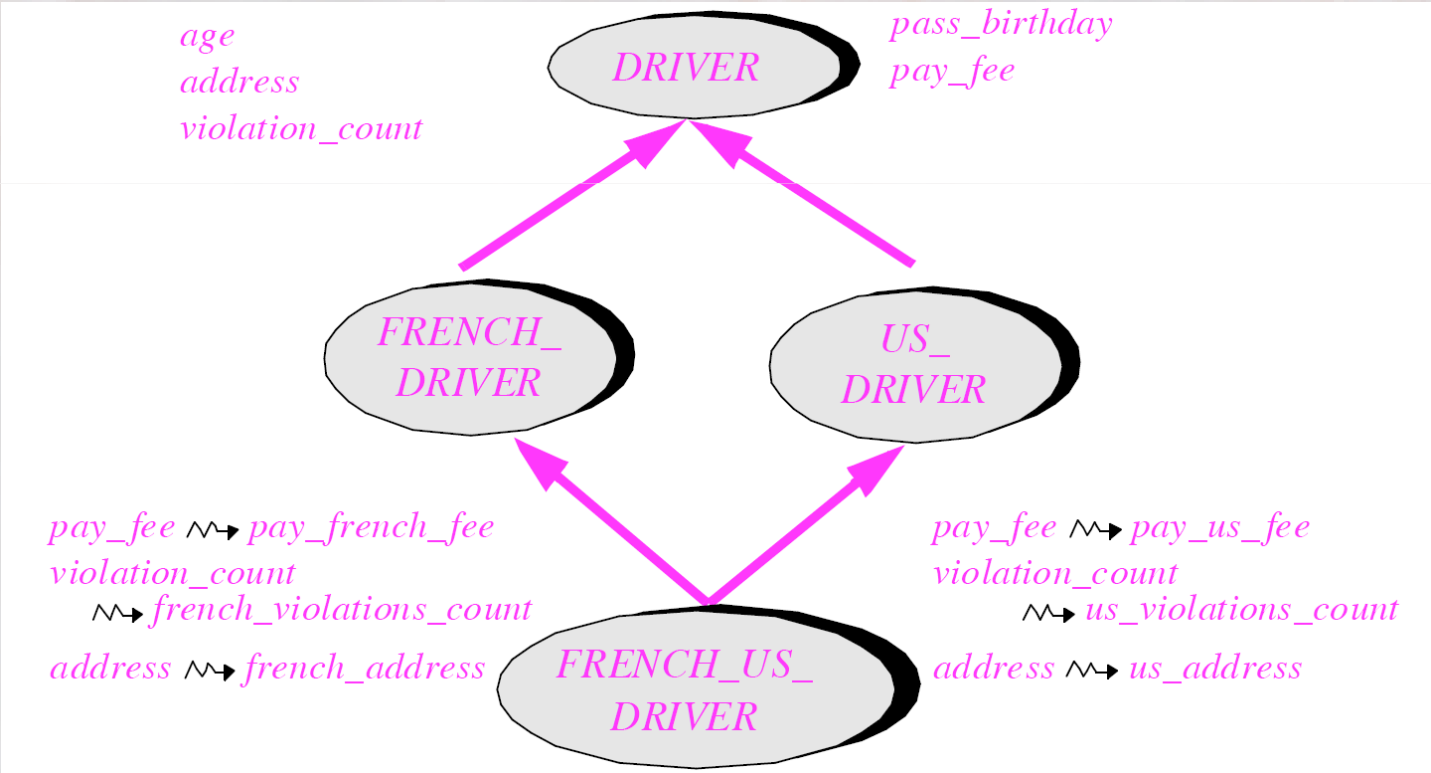
1. Eiffel – primjer ponovljenog nasljeđivanja (repeated inheritance)

- Klasa FRENCH_US_DRIVER nasljeđuje klase FRENCH_DRIVER i US_DRIVER, koje obje nasljeđuju klasu DRIVER. **Hoće li se desiti "dijamantna smrt"?**



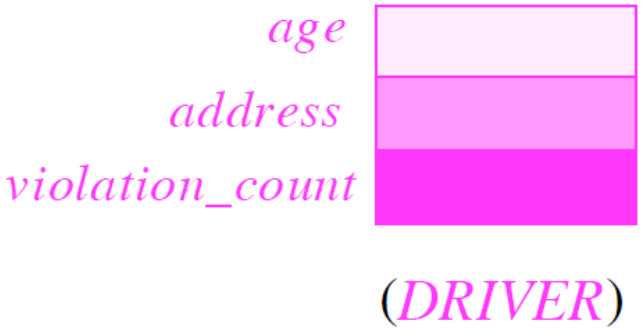
1. Eiffel – primjer ponovljenog nasljeđivanja (nastavak)

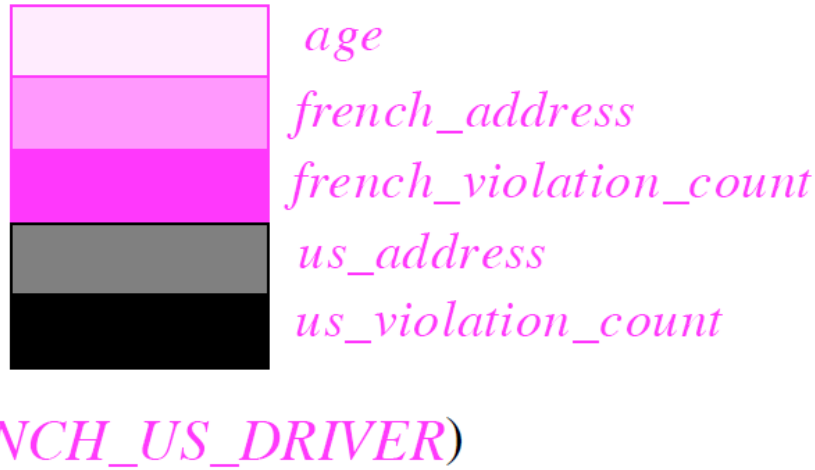
- Pretpostavimo da klase FRENCH_DRIVER i US_DRIVER, preimenuju nasljeđene atribute address i violation_count, te metodu pay_fee. To je tzv. **replikacija atributa / metoda**.



1. Eiffel – primjer ponovljenog nasljeđivanja (nastavak)

- Za razliku od atributa address i violation_count, atribut age se ne replicira, pa je u klasi FRENCH_US_DRIVER on prisutan samo jednom. To je tzv. **zajednički (shared) atribut**. Zajednička je i metoda pass_birthday, za razliku od replicirane metode pay_fee (naglasimo da se u stvarnosti programski kod ipak ne duplicira, ako nije nadjačan).





1. Eiffel – primjer ponovljenog nasljeđivanja (nastavak)

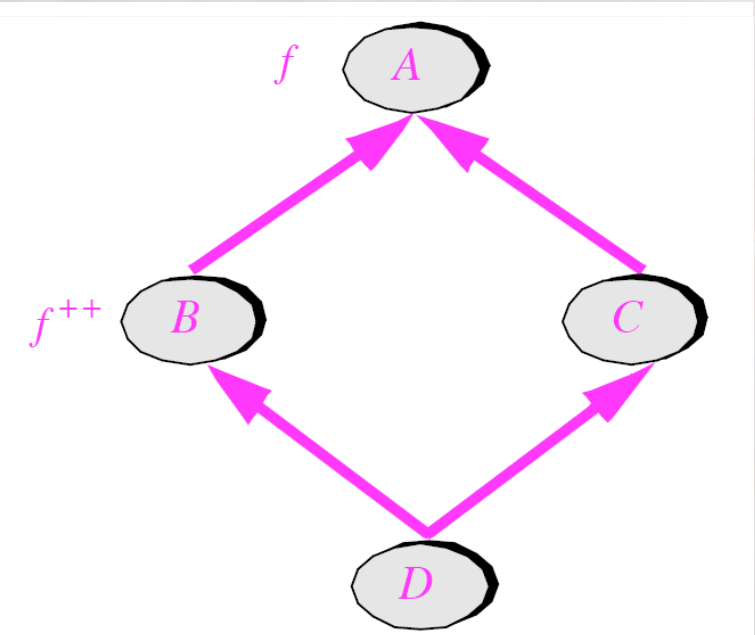
❖ Programski kod:

```
class FRENCH_US_DRIVER inherit
  FRENCH_DRIVER
    rename
      address as french_address,
      violation_count as french_violation_count,
      pay_fee as pay_french_fee
    end
  US_DRIVER
    rename
      address as us_address,
      violation_count as us_violation_count,
      pay_fee as pay_us_fee
    end
feature ... end -- class FRENCH_US_DRIVER
```

1. Eiffel – primjer ponovljenog nasljeđivanja, sa redeklariranjem i bez replikacije

- Pretpostavimo da klase B i C nasljeđuju klasu A, te klasa D nasljeđuje obje klase.
- Ako je metoda f iz A **redeklarirana** u (npr.) klasi B, a obje su varijante efektivne (nisu deferred, tj. apstraktne), moramo koristiti ključnu riječ **undefine** da jednu od njih učinimo apstraktnom (ovdje onu iz C):

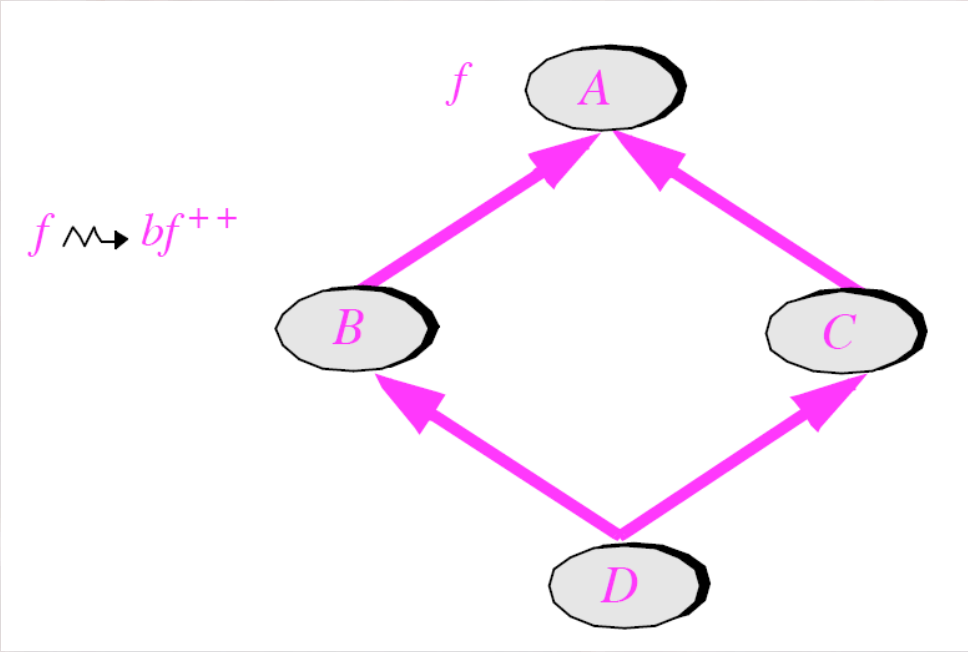
```
class D inherit
   B
   C
      undefine f end
feature ... end
```



1. Eiffel – primjer ponovljenog nasljeđivanja, sa redeklariranjem i sa replikacijom

- Slično kao u prethodnom slučaju, ali je sada metoda f iz A istovremeno i **redeklarirana** u klasi B, i mijenjano joj je ime iz f u bf.
- Sada imamo replikaciju (te metode), pa u klasi D moramo odabrati koju od te dvije varijante želimo koristiti, pomoću ključne riječi **select** (ovdje onu iz C):

```
class D inherit
   B
   C
      select f end
feature ... end
```



2. 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. Najnoviji je C++14 (koji donosi manja poboljšanja).

2. C++ klasa - deklaracija

- Da bismo postigli što veću sličnost primjera u svim jezicima, sljedeća C++ klasa ima javne atribute i metode, iako bi u C++ stilu bilo bolje napraviti privatne atribute i odgovarajuće "get/set" metode za čitanje/postavljanje atributa:

```
#include <string>
#include <iostream>
using namespace std;
class Zivotinja {
public:
  string ime;
  double visina_cm;
  Zivotinja(string p_ime, double p_visina_cm);
  virtual void prikazi_podatke();
  virtual double visina_inch();
};
```

2. C++ klasa
- deklaracija i definicija su razdvojene

```
// definicija klase
Zivotinja::Zivotinja(string p_ime,double p_visina_cm) {
  ime = p_ime; visina_cm = p_visina_cm;
}
void Zivotinja::prikazi_podatke() {
   cout << "Ime: " << ime << ...;
}
double Zivotinja::visina_inch() {
   return visina_cm / 2.54;
}
// ovaj kod nije dio nijedne klase
void main() {
  Zivotinja* l_dz = new Zivotinja("FIFI", 20);
  l_dz->prikazi_podatke();
  delete l_dz;
}
```

2. C++ podržava višestruko nasljeđivanje

- C++ mehanizam višestrukog nasljeđivanja u C++ nije tako fleksibilan kao Eiffel mehanizam.
- Između ostalog, C++ nema preimenovanje (rename) metoda. Ako se od dvije klase naslijede dvije funkcije istog imena, mora se koristiti **scope resolution operator** za rješavanje dvosmislenosti:

```
class A {void f() {…}};
class B {void f() {…}};
class C : public A, public B {…};
void test()
{
    C* p = new C();
    // p->f(); This call would be ambiguous - invalid;
    p->A::f();
    p->B::f();
}
```

2. C++ i ponovljeno nasljeđivanje (repeated inheritance)

- Ako se kod ponovljenog nasljeđivanja želi raditi **replikacija** člana vršne klase, opet se koristi scope resolution operator:

```
class D {int n;};
class E : public D {};
class F : public D {};
class G : public E, public F {};
void f()
{
    G* p = new G();
    // p->n = 0; // This would be invalid: which n?
    // Valid, assigns to version replicated in E
    p->E::D::n = 0;
    // Valid, assigns to version replicated in F
    p->F::D::n = 0;
}
```

- Međutim, ako želimo da se oba člana naslijeđena iz vršne klase **spoje u jedan**, onda se javljaju dodatne komplikacije.

3. 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 (malo ga je nadopunio Gilad Bracha, sa wildcardsima).

3. Scala
- kratka povijest (nastavak)

- Dok je za primjenu GJ-a Sun čekao skoro 6 godina, odmah su preuzeli **Java kompajler koji je Odersky napravio za GJ**. Taj se kompajler koristi od Jave 1.3.
- 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 2.11, a sljedeća verzija 2.12 radit će isključivo na JVM 8 !
- Od tada, Scala se sve više koristi u praksi. Došla je među prvih 50 najkorištenijih jezika, sa tendencijom da se probije među prvih 20. Zanimanje za Scalu naročito se povećalo kada je **Twitter** prebacio glavne dijelove svojih programa iz jezika Ruby u Scalu.

3. Mišljenja drugih kreatora programskih jezika o Scali

- "If I were to pick a language to use today other than Java, **it would be Scala**."

  - **James Gosling, creator of Java**

- "I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 **I'd probably have never created Groovy**."

  - **James Strachan, creator of Groovy.**

3. Osnovne osobine Scale

- **Scala je čisti objektno-orijentirani jezik** (sa statičkom provjerom stipova). Osim toga, na temelju objektno-orijentiranih mogućnosti izgrađene su i brojne funkcijske mogućnosti, **tako da je Scala i funkcijski jezik (ali nije čisti)**.
- Sa funkcijskim osobinama došle su i neke osobine koje su vrlo pogodne za **konkurentno programiranje**.
- **Scala je izvrstan jezik i za pisanje DSL-ova** (Domain-Specific Language), jezika za specifičnu problemsku domenu.
- No, važno je da se može programirati u Scali bez da se napusti Java, jer se **Java i Scala programski kod mogu jako dobro upotpunjavati**.

3. Osobine funkcijskih jezika

- Funkcije višeg reda (higher-order functions)
- Leksičko zatvaranje (lexical closure)
- Podudaranje (sparivanje) uzorka (pattern matching)
- Jednokratno pridruživanje (single assignment)
- Lijena evaluacija (lazy evaluation)
- Zaključivanje o tipovima (type inference)
- Optimizacija repnog poziva (tail call optimization)
- List comprehension: kompaktan i ekspresivan način definiranja listi kao osnovnih podatkovnih struktura funkcijskog programa
- Monadički efekti (monadic effects)

3. Osobine funkcijskih jezika - dodatak

- ❖ Neki dodaju još npr:
- Funkcije kao vrijednosti, "građani prvog reda" ("first-class value")
- Anonimne funkcije
- Currying
- Sakupljanje smeća (garbage collection)
- ❖ Funkcije su u Scali vrijednosti. Kao i svaka druga vrijednost, mogu biti pridružene nekoj varijabli, poslane kao parametri nekoj drugoj funkciji, ili vraćene kao rezultat funkcije.
- ❖ Budući da su u Scali funkcije vrijednosti, a istovremeno su u Scali sve vrijednosti objekti, **slijedi da su u Scali sve funkcije objekti**.

3. Scala funkcije višeg reda

- Funkcije koje kao parametar ili povratnu vrijednost imaju neku drugu funkciju, zovu se funkcije višeg reda (ovdje je to sum):

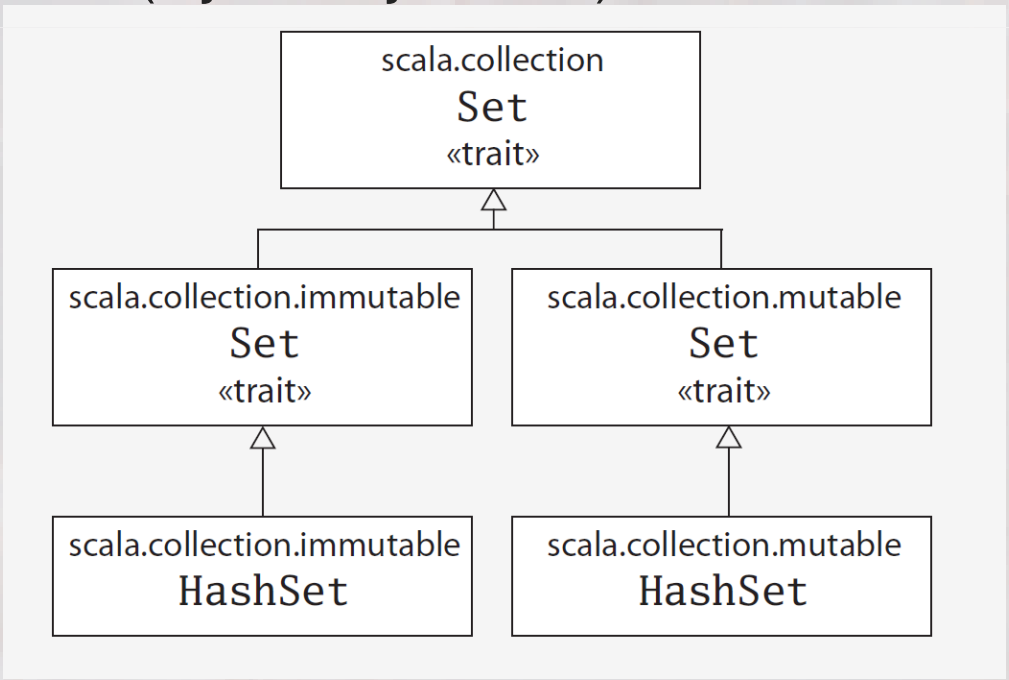
```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

- Sada definiramo dvije funkcije i koristimo ih (za punjenje vrijednosti val varijabli) kao 1. parametar funkcije sum:

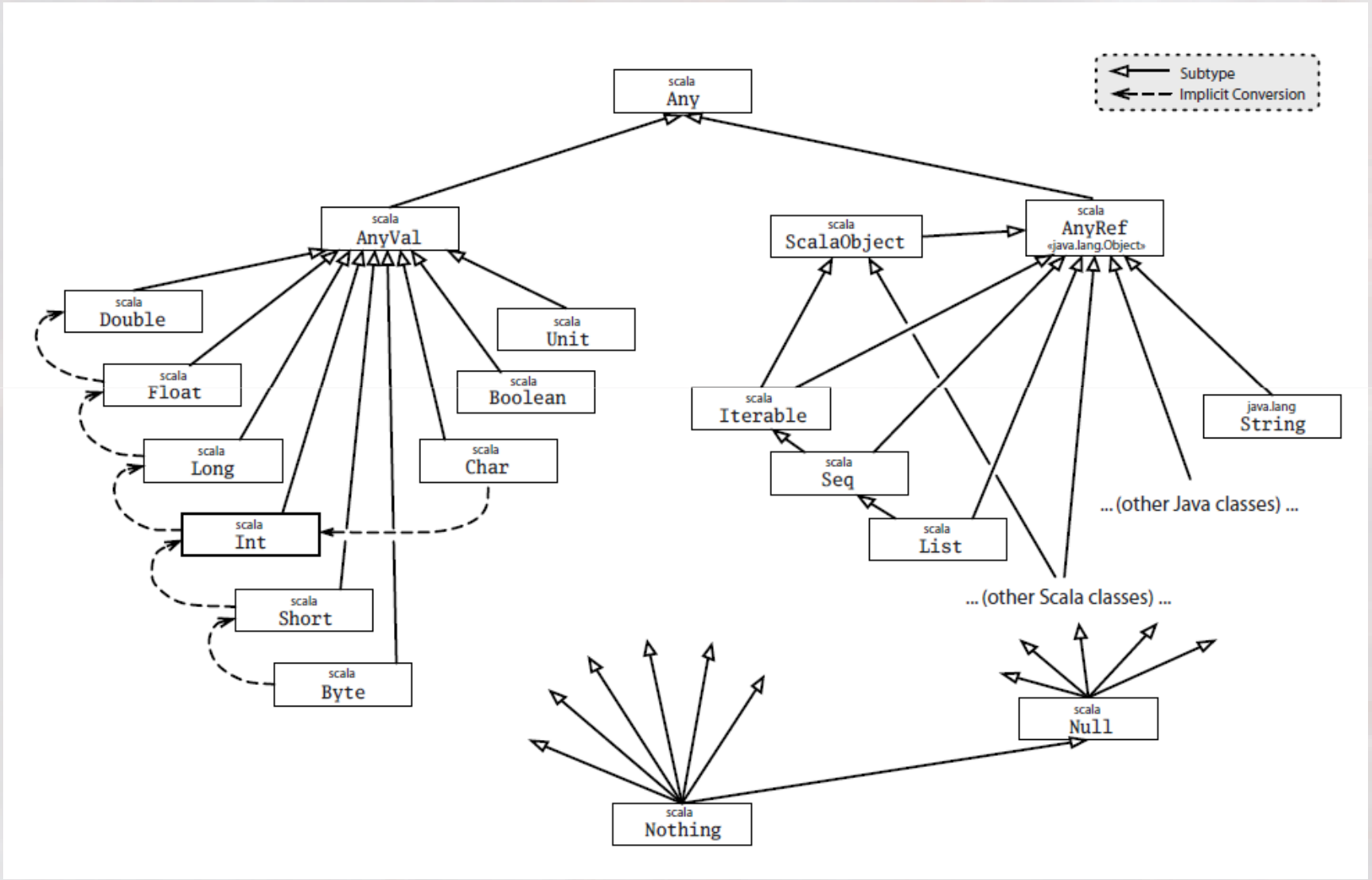
```
def square(x: Int): Int = x * x
def powerOfTwo(x: Int): Int =
  if (x == 0) 1 else 2 * powerOfTwo(x - 1)
val sumSquares = sum(square, 1, 5) // 1+4+9…=55
val sumPowersOfTwo = sum(powerOfTwo, 1, 5) //62
```

3. Scala kolekcije - većinom postoje u dvije varijante, imutabilna i mutabilna

- Funkcijski jezici poznati su po tome da imaju izvrstan način rada sa kolekcijama, naročito sa listama (list).
- Glavne Scala kolekcije su **List, Set i Map**. List je uređena kolekcija objekata, Set je neuređena kolekcija objekata, a Map je skup parova (ključ, vrijednost).



3. Hijerarhija Scala klasa



3. Scala i višestruko nasljeđivanje - Scala trait

- Može izgledati da je trait nešto kao Java sučelje sa konkretnim metodama. No, trait može imati skoro sve što ima i klasa, npr. **može imati i polja, a ne samo metode**. Trait se, zapravo, kompajlira u Java sučelje i pripadajuće pomoćne klase koje sadrže implementaciju metoda i atributa.
- U Scali, klasa može naslijediti samo jednu (direktnu) nadklasu, ali zato **može naslijediti više traitova**.
- Iako traitovi sliče na klase, razlikuju se u dvije važne stvari. Prvo, trait nema parametre klasa.
- Drugo, kod višestrukog nasljeđivanja klasa, metoda koja se poziva sa super može se odrediti tamo gdje se poziv nalazi. Kod traitova se to, međutim, **određuje metodom koja se zove linearizacija** (linearization) klase i traitova koji su miksani s tom klasom.

3. Primjeri korištenja traitova

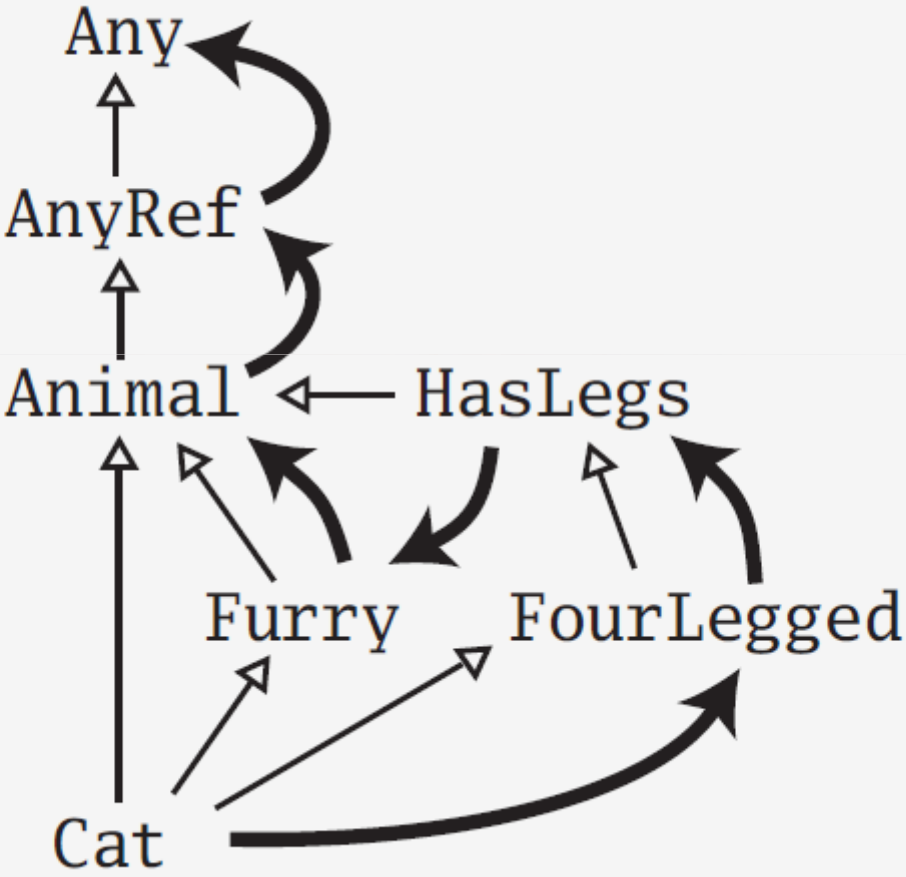
❖ Primjer za prikaz **linearizacije**:

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal
    with Furry with FourLegged
```

❖ Trait se može koristiti i selektivno na razini objekta. Npr., pretpostavimo da klasa Macka ne nasljeđuje trait Programer. **Instanca (objekt) ipak može naslijediti taj trait**:

```
val jakoPametnaMacka =
    new Macka("Mica maca") with Programer
```

3. Hijerarhija nasljeđivanja i linearizacija klase Cat



3. Trait – rezolucija konflikta kod nasljeđivanja dvije metode istog imena

- Primjer rezolucije (rješavanja) konflikta kod nasljeđivanja dvije metode istog imena – **programer rješava eksplicitno** :

```
trait Hodor {
  def speak = "Hodor!"
}
trait Stark {
  def speak = "Winter is coming"
}
class StarkHodor extends Hodor with Stark {
  override def speak = "Hodoris coming!"
  // we can also use super[Hodor].speak
}
```

4. 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.
- 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. Eiffel sintaksa je inspirirana jezikom ADA 83. Scala je negdje između.
- Međutim, Java nije podskup C++ jezika. Također, iako jednostavniji nego C++, Java nije baš jednostavan jezik.
- Eiffel i Scala su "čisti" OOPL jezici. C++ nije, jer je morao zadržati (potpunu) kompatibilnost sa jezikom C.
  Java je negdje između.

4. Što je nakon pojave Jave rekao Bertrand Meyer u "Object Oriented Software Construction" (1997.)

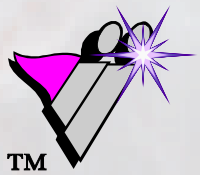
- Java is one of the most innovative developments in the software field, and there are many reasons to be excited about it. Java's language is not the main one.

  As an O-O extension of C, it has missed some of the lessons learned since 1985 by the C++ community; **as in the very first version of C++, there is no genericity and only single inheritance is supported.**

  Correcting these early oversights in C++ was a long and painful process, creating years of havoc as compilers never quite supported the same language, books never quite gave accurate information, trainers never quite taught the right stuff, and programmers never quite knew what to think.

4. Java je već odavno mogla imati nešto kao lambda - Bertrand Meyer u "Touch of Class" (2009.)

- Java has no equivalent to the notion of agent as used in this book or to similar mechanisms in other languages, such as C#'s "delegates" studied in the next appendix.
- ... Java of course offers alternatives for such needs … for most other cases the recommended **Java solution is to use inner classes** (as detailed, for the example of GUI programming, in the next section).
- For a long time, the Java community denied that anything else was necessary …
- **… which (delegates) were then being proposed for Java, but only made their way into the future C# …**
- Almost a decade and a half later the designers finally relented: it has been announced that Java 7 will include an agent-like mechanism known as **closures**. (zapravo, Java 8)



4. Java klasa (za razliku od C++ klase) nema odvojenu deklaraciju od definicije i programski kod je uvijek dio klase

```
public class Zivotinja {
  public String ime;
  public double visina_cm;
  public Zivotinja(String p_ime, double p_visina_cm) {
    ime = p_ime; visina_cm = p_visina_cm;
  }
  public void prikazi_podatke() {
    System.out.println("  Ime: " + ime ...);
  }
  public double visina_inch() {...}
  public static void main(String[] args) {
    Zivotinja l_dz = new Zivotinja("FIFI", 20);
    l_dz.prikazi_podatke();
  }
}
```

4. Java 8
- najvažnije nove mogućnosti:
lambda izrazi, default metode, Streams

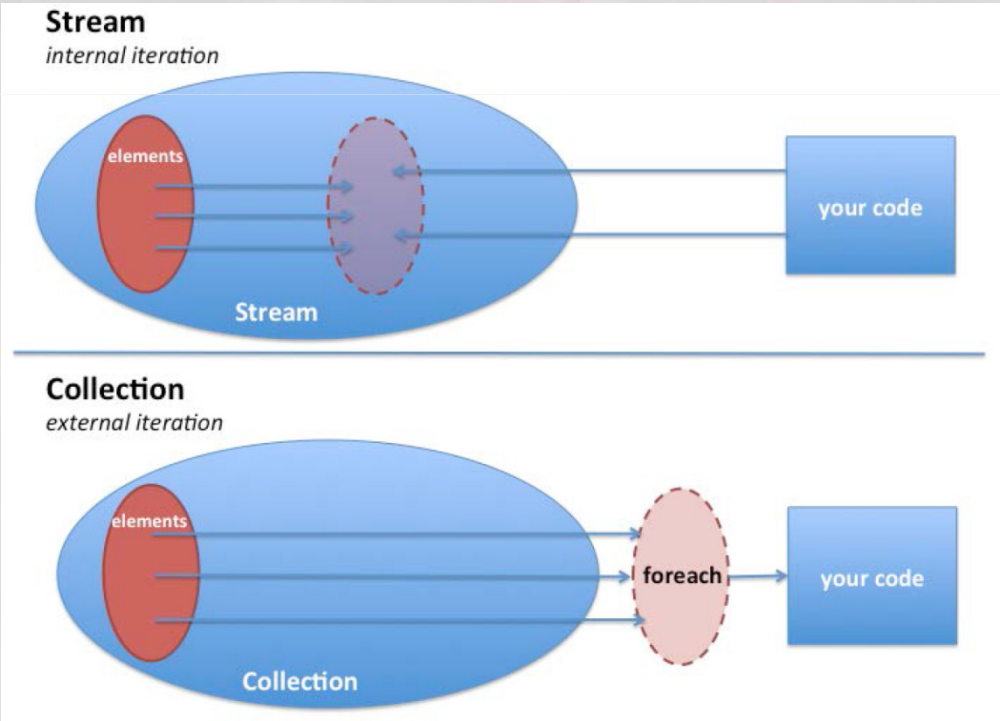
- Na temelju onoga što čitamo i čujemo, mogli bismo zaključiti da je najvažnija nova mogućnost u Javi 8 **lambda izraz** (ili kraće, **lambda**).
- Inače, lambda izraz je (u Javi) naziv za metodu bez imena. U pravilu je ta metoda funkcija, a ne procedura. Zato možemo reći i da je **lambda izraz anonimna funkcija**, koja se može javiti kao parametar (ili povratna vrijednost) druge funkcije (koja je, onda, funkcija višeg reda).
- U Javi 8 pojavile su se i tzv. **default metode** u Java sučeljima (interfaces). One, zapravo, predstavljaju uvođenje **višestrukog nasljeđivanja implementacije** u Javu.
- Međutim, lambda izrazi i default metode su, na neki način, posljedica uvođenja treće važne mogućnosti u Javi 8, a to su **Streamsi**, koji nadograđuju dosadašnje Java kolekcije.

4. Java 8 Streams (java.util.stream) - zašto su potrebni?

- Pojava **masivno višejezgrenih procesora** traži bolji i lakši način izrade paralelnih programa od dosadašnjih načina. Jedan (dobar) način je da se koristi funkcijsko programiranje, a naročito paralelne kolekcije.
- Java nema paralelne kolekcije (collections), ali su zato uvedeni Streamsi, kako bi se postojeće kolekcije "zaogrnule" u (paralelne) Streamse i mogle (indirektno) paralelizirati.
- **Kako bi se olakšalo korištenje Streamsa, bilo je potrebno uvesti i lambda izraze i default metode**. Međutim, lambda izrazi i default metode mogu biti korisni i za ostale svrhe, ne samo tvorcima API-a, već i "običnim" programerima.
- Za razliku od dosadašnjih kolekcija, koje sadrže sve svoje elemente u memoriji, Streamse možemo shvatiti kao **"vremenske kolekcije"**, čiji se elementi (kojih teoretski može biti beskonačan broj) stvaraju po potrebi.

4. Java 8 – razlika između eksterne i interne iteracije

- "Stare" Java kolekcije koriste eksternu (eksplicitnu) iteraciju, koju piše programer. Streamsi imaju internu iteraciju.
- **Streamsima se može poslati naš kod, kao lambda izraz**, za definiranje obrade elemenata:



4. Java 8 Streams programiranje je deklarativno (što, a ne kako) – kao SQL

```
// 1. "klasična" obrada "klasične" kolekcije - for petlja
private static void checkBalance(List<Account> accList) {
  for (Account a : accList)
    if (a.balance() < a.threshold) a.alert();
}
// 2. primjena lambda izraza za obradu "klasične" kolekcije
private static void checkBalance(List<Account> accList) {
  accList.forEach(
   (Account a) -> { if (a.balance() < a.threshold) a.alert(); }
  );
}
// 3. primjena Streamsa za paralelizaciju "klasične" kolekcije
private static void checkBalance(List<Account> accList) {
  accList.parallelStream().forEach(
   (Account a) -> { if (a.balance() < a.threshold) a.alert(); }
  );
}
```

4. Java 8 default metode

- Kako je već rečeno, default metode su u Javi 8 trebale prvenstveno zbog uvođenja Stream API-a, iako su default metode korisne i za ostale svrhe (ne samo tvorcima API-a).
- Kod uvođenja Streamsa, bilo je potrebno nadograđivati brojna postojeća Java sučelja, tj. dodavati im nove metode. Međutim, **kad dodajemo nove metode u sučelje, moramo mijenjati sve klase** koje ga (direktno ili indirektno) nasljeđuju, što može značiti izmjenu milijuna redaka programskog koda.
- Default metode su riješile taj problem. **Sučelje sada može imati i implementaciju metode**, a ne samo deklaraciju.
- Java 8 sučelje sa default metodom **nije tako moćno kao Scala trait**. Scala trait može imati i ne-default metode, može imati stanje (atribute), može se komponirati za vrijeme runtimea, može pristupati instanci klase koja ga nasljeđuje. Ništa od toga Java 8 sučelje (sa default metodama) ne može.

4. Java 8 default metode

```
// 1. ako bismo postojeće sučelje Iterable
// htjeli proširiti sa novom metodom forEach,
// morali bismo mijenjati svaku klasu
// koja (direktno ili indirektno) nasljeđuje to sučelje
public interface Iterable<T> {
  public Iterator<T> iterator();
  public void forEach(Consumer<? super T> consumer);
}

// 2. default metode rješavaju taj problem
public interface Iterable<T> {
  public r<T> iterator();
  public default void forEach(Consumer<? super T> consumer) {
    for (T t : this) {
      consumer.accept(t);
    }
  }
}
```

4. Java 8 u odnosu na Scalu ("Scala" prezentacija, Assaf Israel, Technion, 2013.)

Java 8

- Has primitives
- No operator overloading
- Statements and expressions
- Has static methods and members
- Some type inference
- Library collections are mutable
- `default` methods
- Basic switch functionality
- Basic foreach functionality
- No extension methods
- Has checked exceptions
- Type Erasure (JVM property)
- Little to no syntactic sugar

Scala

- Everything is an object
- No concept of "operators", just methods
- Everything is an expression
- Uses a companion object/Singleton
- Stronger type inference
- Collections are immutable by default
- `traits`
- Powerful pattern matching (a la ML)
- Powerful for comprehensions
- Can define `implicit` coercions
- Doesn't have checked exceptions
- Can get around type erasure
- A lot (too much?) of syntactic sugar

4. Java 8 je tek sada dobila ono što Scala već dugo ima ("Scala" prezentacija)

A note on the "new" features in Java 8

All the new features in Java 8 have been in Scala for years

- Higher order functions
- **`optional`** values
- Lambda Expressions
  - The syntactic sugar of **`Functional Interfaces`** can be implemented using **`implicit`**
- **`default`** methods on **`interfaces`** are actually weak **`traits`**
- Streams and easy parallelization

Zaključak

- U zadnjih desetak godina sve se više govori (i) o tome da bi programiranje trebalo biti multiparadigmatsko, tj. da bismo trebali **koristiti onu jezičnu paradigmu koja je najprirodnija za rješavanje određenog problema**. Dakle, dobro je koristiti i **funkcijsko, a ne samo imperativno programiranje**.
- Osim toga, **masivno višejezgreni procesori** traže bolji i lakši način izrade paralelnih programa. Jedan (dobar) način je da se koristi funkcijsko programiranje.
- U Javi 8 je olakšavanje paralelnog programiranja potaknulo uvođenje **Streamsa**, što je "poguralo" (i) realizaciju **lambda izraza** i **default metoda**, čime je višestruko nasljeđivanje "na mala vrata" ušlo i u Java svijet. Nadamo se da će se te nove mogućnosti pokazati kao dobre, i da će se dalje razvijati.