PL/SQL++
(objektne osobine PL/SQL jezika - usporedba sa OOPL jezicima Java i Eiffel)

Zlatko Sirotić
Istra informatički inženjering d.o.o., Pula
e-mail: zlatko.sirotic@iii.hr

SAŽETAK

Oracle korporacija je 1997. godine objavila Oracle 8.0 verziju baze i nazvala ju objektno-relacijskom. U skladu s tim nadopunjen je i programski jezik PL/SQL (ali i 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. Te je mogućnosti Oracle uveo 2001. u bazi 9i. U radu se prikazuju neke objektne osobine Oracle PL/SQL 9i R2 jezika, ali bez prikaza objektnih osobina SQL jezika (prikazuje se rad sa tranzijentnim objektima, a ne sa perzistentnim objektima), čime je omogućena usporedba PL/SQL jezika sa OOPL jezicima Java i Eiffel.

1. UVOD

Oracle korporacija je 1988.godine objavila verziju 6.0 svog relacijskog DBMS-a. U sklopu baze 6.0 pojavio se 1991. godine novi programski jezik - PL/SQL (Procedural Language extensions to the Structured Query Language), koji je napravljen kao proceduralna nadopuna (deklarativnog) programskog jezika SQL, kako bi se olakšalo programiranje "mission-critical" poslovnih aplikacija. Oracle je napravio PL/SQL na temelju programskog jezika ADA 83. Budući da ADA 83 nije bila objektno-orijentirani programski jezik (ADA 95 je dobila neke objektne mogućnosti), niti PL/SQL nije bio OOPL.

Međutim, 1997. je Oracle objavio 8.0 verziju baze i nazvao ju objektno-relacijskom. U skladu s tim nadopunjen je i programski jezik PL/SQL (naravno, i 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. Te je mogućnosti Oracle uveo 2001. godine u bazi 9i. U radu se prikazuju neke objektne osobine Oracle PL/SQL 9i R2 (baza 10g nije donijela nekih novosti na tom području), ali bez prikaza objektnih osobina SQL jezika, tj. prikazuje se rad sa tranzijentnim (prolaznim) objektima, a ne sa perzistentnim (stalnim) objektima, čime je omogućena usporedba specijaliziranog PL/SQL jezika sa OOPL jezicima opće namjene Java i Eiffel. Naravno, prilikom kritike PL/SQL objektnih (ne)mogućnosti svakako trebamo uzeti u obzir to da njegovu "prirodnu okolinu" čini baza podataka (sa svojim perzistentnim objektima i SQL jezikom).

2. KRATKA POVIJEST PROMATRANIH JEZIKA

Redoslijed pojavljivanja navedena tri jezika je sljedeći: Eiffel (1985.), PL/SQL (1991.), Java (1995.). Kako je već navedeno, PL/SQL je tek 1997. dobio neke objektne mogućnosti, a značajno ih proširio 2001. godine, pa možemo reći da je PL/SQL najmlađi po stažu u objektno-orijentiranom svijetu.

Sva tri jezika imaju zajedničkog pretka – Algol (ALGOrithmic Language), kojeg su napravili Backus i Naur daleke (za informatiku) 1958. godine, a ubrzo su se pojavili i prvi prevodioci. Zanimljivo je [Joyner 1999] da je jedan od prvih prevodioca za Algol 1960. godine napisao (za tadašnju firmu Burroughs) slavni Donald Knuth, pisac serije kapitalnih knjiga "The Art of Computer Programming". Algol je ostavio ogroman utjecaj na programske jezike koji su došli nakon njega. Poznati informatičar C.A.R. Hoare duhovito je rekao kako je Algol 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! Algol je bio i prvi formalno dizajnirani jezik, jer je njegova sintaksa formalno definirana pomoću posebne notacije BNF (Backus-Naur Form). Kasnije je Niklaus Wirth (tvorac jezika Algol W, Pascal, Modula-2 i Oberon) proširio BNF notaciju u EBNF (Extended BNF), koja se danas široko upotrebljava.

Jedan od "Algol-oidnih" jezika bio je i Simula 1, koji je bio namijenjen uglavnom za simulacije. Kasnija verzija, Simula 67 (nastala je 1967. godine, a autori su Kristen Nygaard i Ole-Johan Dahl), je jezik opće namjene i to prvi OOPL u povijesti, iako mnogi misle da je prvi OOPL jezik Smalltalk-72. Autor Smalltalk-a, Alan Kay, bio je (kao i autori jezika C++ i Eiffel) inspiriran jezikom Simula 67. Autor jezika ADA, Jean Ichbiah, napisao je jedan od prvih Simula 67 prevodioca. Simula 67 je i dan-danas živi jezik (pod imenom Simula).

C (autor je Dennis Ritchie), također Algol-ov potomak, nastao je 1970. godine (kao kraj lanca jezika CPL-BCPL-B-C), 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 jednoj od dvije struje: C struji 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. Godine 1997. donesen je i ISO standard.

Eiffel je 1985. dizajnirao Bertrand Meyer, jedan od najvećih autoriteta na OOPL području (1986. napravljen je prvi compiler). Prvo izdanje (1988.) njegove knjige OOSC, kapitalno je djelo informatičke literature (aktualno je 2. izdanje iz 1997.). Ta je knjiga upoznala širu javnost sa jezikom Eiffel (tada verzije 2). Od početka je podržavao višestruko nasljeđivanje, generičke klase, obradu iznimaka, ali i Design by Contract. U široj je javnosti daleko manje poznat nego C++ i Java, ali ga mnogi autoriteti smatraju danas najboljim OOPL jezikom. Ove godine (2005) donesen je ECMA standard, kao priprema za ISO standard.

Jezik Java se pojavio 1995. i (u pravo vrijeme!) reklamiran je kao jezik za Internet, čime je odmah stekao ogromnu slavu. To je jezik opće namjene i vrlo portabilan, a počeci mu sežu u 1992., kada se zvao Oak i bio namijenjen za upravljanje uređajima za kabelsku televiziju i slične namjene. 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 i PL/SQL imaju donekle sličnu sintaksu, jer su oba inspirirana jezikom ADA 83. Eiffel i Java su "čisti" OOPL jezici, a C++ i PL/SQL nisu (jer su morali zadržati kompatibilnost). Zajedničko im je svima da su to jezici sa statičkom provjerom tipova (za razliku od npr. jezika Smalltalk).

3. KLASE, UPRAVLJANJE PRISTUPOM

Klasa je dio programskog koda, takav da ima osobine i modula i tipa (pojednostavljeno se može reći da vrijedi formula: KLASA = MODUL + TIP). Klasa, a ne objekt, je osnovni element objektno-orijentiranih programskih jezika (možda bi se trebali zvati COPL, umjesto OOPL). Oracle koristi pojam "object type" (što nije loš naziv). Zbog jednostavnije usporedbe tri OOPL jezika, koristit ćemo uvijek pojmove "klasa", "atribut" i "metoda". Kod imenovanja klasa, atributa, metoda, parametara i lokalnih varijabli uglavnom ćemo koristi uobičajeni PL/SQL stil (donja crta za razdvajanje riječi u imenu). Jedino ćemo imena klasa malo prilagoditi pojedinom jeziku, pa ćemo imena PL/SQL klasa pisati malim slovima, imena Java klasa sa početnim velikim slovima, a imena Eiffel klasa sa svim velikim slovima. Treba napomenuti da PL/SQL i Eiffel nisu osjetljivi na velika/mala slova, dok Java jeste.

PL/SQL klasa se (barem za sada) može definirati 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 klase se metode (deklarirane u specifikaciji) u potpunosti definiraju. U sljedećem primjeru prikazana je PL/SQL klasa koja ima dva atributa i tri metode, od kojih je jedna konstruktor:

```
CREATE OR REPLACE TYPE zivotinja AS OBJECT (
   ime       VARCHAR2 (20),
   visina_cm NUMBER,
   CONSTRUCTOR FUNCTION zivotinja
      (p_visina_cm NUMBER, p_ime VARCHAR2) RETURN SELF AS RESULT,
   MEMBER PROCEDURE prikazi_podatke,
   MEMBER FUNCTION visina_inch RETURN NUMBER
)  NOT FINAL;
/
```

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

   MEMBER PROCEDURE prikazi_podatke IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE (
         ' Ime:' || ime || ' Visina (cm):' || visina_cm ||
         ' Visina (inch):' || visina_inch);
   END;

   MEMBER FUNCTION visina_inch RETURN NUMBER IS
   BEGIN
      RETURN visina_cm / 2.54;
   END;
END;
/
```

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

```
DECLARE l_dz zivotinja := NEW zivotinja (20, 'FIFI');
BEGIN l_dz.prikazi_podatke; END;
```

Poznavatelji PL/SQL jezika odmah će uočiti da naredba za kreiranje klase dosta podsjeća na naredbu za kreiranje PL/SQL paketa (treba napomenuti da PL/SQL paket nema skoro nikakvih sličnosti sa Java paketom, npr. PL/SQL paket ne može sadržavati klase). Međutim, za razliku od PL/SQL paketa, PL/SQL klasa za sada ne može imati privatne atribute ili metode, tj. svi atributi i metode su javni. Da bismo postigli što veću sličnost primjera u svim jezicima, sljedeća Java klasa ima javne atribute i metode, iako bi u Java stilu bilo bolje napraviti privatne atribute i odgovarajuće "get/set" metode za čitanje/postavljanje atributa. Vidimo da PL/SQL ima odvojenu deklaraciju od definicije i programski kod (PL/SQL neimenovani blok) koji nije dio nijedne klase. Za razliku od toga, Java klasa nema odvojenu deklaraciju od definicije, a "main" procedura mora biti dio neke 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 + " Visina (cm):" + visina_cm +
         " Visina (inch):" + visina_inch ());
   }
   public double visina_inch () {
      return visina_cm / 2.54;
   }
   public static void main (String[] args) {
      Zivotinja l_dz = new Zivotinja ("FIFI", 20);
      l_dz.prikazi_podatke ();
   }
}
```

Slijedi primjer ekvivalentne Eiffel klase. Napomenimo da konstruktor metoda u Eiffel-u ne mora imati isto ime kao klasa u kojoj se nalazi, ali smo namjerno zadržali isto ime:

```
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)
         print (", Visina (cm): ");   print (visina_cm)
         print (", Visina (inch): "); print (visina_inch)
      end
   visina_inch: REAL is
      do
         Result := visina_cm / 2.54
      end
end
```

Vidimo da se u Eiffel-u ne mora koristiti znak za odvajanje naredbi " ; ", ali je preporuka da se koristi ako imamo više naredbi u istom retku (zbog čitljivosti). Primijetimo da u funkciji ne postoji ključna riječ **return**, već Eiffel ima posebnu predefiniranu varijablu **Result**. Također, primijetimo da PL/SQL i Eiffel omogućavaju tzv. uniformni pristup (uniform acess), tj. poziv atributa ne razlikuje se od poziva funkcije bez parametara. Npr. u proceduri "prikazi_podatke" funkciju pozivamo sa "visina_inch", dok Java traži da se koriste zagrade i kad metoda nema parametara, pa se mora pisati "visina_inch ()". Uniformni pristup se smatra značajnom mogućnošću OOPL jezika.

U Eiffel-u, kao i u Javi, programski kod mora uvijek biti dio (neke) klase. Procedura "pokreni", koja ima istu funkciju kao neimenovani PL/SQL blok, odnosno Java procedura "main" u prethodnim primjerima, nalazi se u unutar klase TEST:

```
class TEST
create pokreni
feature {ANY}
   pokreni is
      local
         l_dz: ZIVOTINJA
      do
         create l_dz.zivotinja ("FIFI", 20)
         l_dz.prikazi_podatke
      end
end
```

Naglasimo opet da smo mogli definirati Java i Eiffel klase tako da neki atributi i/ili metode ne budu javni, ali PL/SQL klase to ne omogućavaju. To je dosta čudno, jer PL/SQL od svojih početaka ima pakete, u kojima postoje javne (ili globalne) varijable, procedure i funkcije (deklarirane u specifikaciji paketa) i privatne varijable, procedure i funkcije (deklarirane u tijelu paketa). Vjerojatno možemo očekivati da će Oracle ubrzo omogućiti privatne metode (ako ne i atribute).

Java omogućava definiranje javnih, zaštićenih i privatnih atributa/metoda, pomoću specifikatora pristupa **public**, **protected** i **private**. Javni atributi/metode pristupačni su svim klasama, zaštićeni atributi/metode pristupačni su samo klasi u kojoj su deklarirani i podklasama te klase, a privatni atributi/metode nisu pristupačni niti podklasama. Java također ima četvrtu vrstu pristupa, koja se često naziva "paketni pristup" i za koju ne postoji ključna riječ. Java klase koje se nalaze u istom paketu mogu pristupati atributima/metodama drugih klasa iz istog paketa, osim onima koji imaju specifikator pristupa **private**, a oni atributi/metode koji nemaju uz sebe specifikator **public**, **protected** ili **private**, klasama izvan paketa "izgledaju" kao da imaju specifikator **privat**e. Osim atributa/metoda u Javi se i cijela klasa može označiti kao javna (ali, u jednoj .java datoteci može postojati najviše jedna takva klasa). Javna klasa vidljiva je klasama izvan paketa, a inače (ako nije označena sa **public**) vidljiva je samo klasama iz istog paketa.

Eiffel ima vrlo jednostavno i vrlo fleksibilno definiranje pristupa, jer za svaki atribut/metodu možemo definirati koja mu klasa može pristupati. Npr, ako ne želimo dati niti jednoj klasi pristup atributu "visina_cm" i funkciji "visina_inch", pristup konstruktoru "zivotinja" želimo dozvoliti samo klasama "TEST1" i "TEST2", a pristup atributu "ime" i proceduri "prikazi_podatke" želimo dozvoliti svim klasama, napisat ćemo:

```
feature {NONE} visina_cm ...; visina_inch ...;
feature {TEST1, TEST2} zivotinja ...;
feature {ANY} ime ...; prikazi_podatke ...;
```

Važno je naglasiti da u Eiffel-u dozvola pristupa atributu znači dozvolu čitanja atributa, ne i dozvolu pisanja. Vrijednosti atributa može mijenjati samo metoda iz klase u kojoj je atribut definiran (ili iz njene podklase), pomoću "set" procedura. Zbog toga u Eiffel-u ne treba raditi "get" funkcije. Java mora imati "get" funkcije ako želimo atribute zaštiti od upisa izvan klase. Zanimljivo je da Eiffel može sakriti atribute nekog objekta čak i od drugih objekata (instanci) iste klase, pomoću **{NONE}** ili, što je isto, pomoću **{}**. Eiffel ne poznaje sakrivanje atributa/metoda od podklase (nema nešto kao Java **private**), jer Meyer drži da to nije u skladu sa OO modelom.

4. OBJEKTI, STATIČKI ATRIBUTI I METODE, APSTRAKTNE KLASE

Za razliku od klasa, čija je priroda statička (to je programski kod), objekti su dinamičke prirode. Objekti nastaju i nestaju za vrijeme izvođenja programa (zapravo, perzistentni objekti mogu "preživjeti" kraj izvođenja programa, za razliku od tranzijentnih objekata). Objekt je napravljen na temelju određene klase, pa se zove i instanca klase. Pojednostavljeno rečeno, objekt je record struktura koja se sastoji od polja (fields) koja (polja) odgovaraju atributima klase (na temelju koje je objekt nastao).

Vidjeli smo u prethodnoj točci da se za kreiranje objekata koristi posebna metoda – konstruktor, koja se u jezicima PL/SQL i Java mora zvati isto kao klasa u kojoj se nalazi i koja se poziva pomoću riječi **new** (PL/SQL ne traži da napišemo riječ **NEW**, ali to je preporučljivo, zbog jasnoće). U Eiffel-u se konstruktor ne mora zvati isto kao klasa, a poziva se sa **create**. Jedna klasa može imati i više konstruktora. Budući da se svi PL/SQL i Java konstruktori unutar iste klase moraju isto zvati, moraju se ipak po nečemu razlikovati – po signaturi, tj. broju i tipu parametara. Nije nužno da programer deklarira konstruktor, jer sva tri jezika imaju default konstruktor. Eiffel i Java default konstruktori su bez parametara, što znači da će se prilikom kreiranja objekta njegova polja napuniti default vrijednostima (ovisnima o tipu polja). PL/SQL default konstruktor ima onoliko parametara koliko klasa ima atributa. To je bio razlog zašto smo u prethodnoj točci "obrnuli" redoslijed parametara u PL/SQL konstruktoru (u odnosu na atribute), jer bi se inače signatura našeg konstruktora podudarala sa signaturom default konstruktora (zapravo, eksplicitni konstruktor smo napravili samo radi ilustracije).

Možemo reći da postoje statički i dinamički objekti. Kada u programu koristimo neku varijablu (atribut, parametar ili lokalnu varijablu), možemo reći da se statički objekt nalazi unutar varijable, dok se dinamički objekt ne nalazi unutar varijable, već varijabla sadrži samo referencu na objekt. Dinamički objekti se uvijek kreiraju u dinamičkom dijelu memorije (heap), a statički objekti se najčešće kreiraju u (relativno) statičkom dijelu memorije (stack), osim ako su podobjekti dinamičkog objekta. Samo Eiffel (ali i C++) ima oba tipa objekata (statičke i dinamičke). U Eiffel-u se deklaracija statičkih objekata radi sa **expanded** (Eiffel statički objekti ne koriste konstruktore, a C++ da). Java nema statičkih objekata - osim objekata koji pripadaju određenim predefiniranim tipovima (tzv. primitivnim tipovima, koji se u Javi ne smatraju klasama), sve ostalo su dinamički objekti. Suprotno od Jave, PL/SQL nema reference, barem ne na tranzijentne objekte (postoje tzv. REF reference između objekata u bazi, tj. između perzistentnih objekata).

Problem sa dinamičkim objektima je da njih treba eksplicitno brisati kada više nisu potrebni, dok se statički objekti automatski eliminiraju prilikom završetka određenog dijela programa (npr. izlaska iz **BEGIN ... END** bloka). Ako se dinamički objekti ne brišu kada više nisu potrebni, nastaje problem (nepotrebnog) zauzeća memorije. Ako se pak brišu, a još uvijek su potrebni, nastaje još veći problem, jer će zbog toga program vjerojatno "puknuti" (ili će se čudno ponašati). Eiffel i Java (za razliku od C++) imaju automatsko sakupljanje smeća (garbage collection), tj. sistem sam briše objekte koji više nisu potrebni i zato nemaju naredbu **delete** i destruktore. PL/SQL nema dinamičkih objekata, pa nema niti automatsko sakupljanje smeća, niti naredbu za brisanje objekata i destruktore.

Osim statičkih objekata, postoje i statički atributi i statičke metode. Od tri promatrana jezika, statičke atribute ima samo Java (ključna riječ je **static**). Za razliku od običnih atributa, koji postoje za svaki objekt, statički atributi postoje samo na razini klase. Može se reći da su to globalne varijable za OOPL jezike. Statičke metode imaju PL/SQL i Java. Kod poziva statičkih metoda navodi se ime klase, a ne ime objekta. Statička metoda ne može pozivati ne-statičke atribute/metode i ne može sadržavati riječ **this** (Java) ili **SELF** (PL/SQL) kojom se označava tekući objekt (Eiffel za tekući objekt koristi riječ **Current**).

Apstraktne klase su takve klase iz kojih se ne mogu generirati objekti. PL/SQL označava takve klase sa **NOT INSTANTIABLE** (default je **INSTANTIABLE**). Apstraktna klasa u Eiffel-u označava se sa **deferred**, a u Javi sa **abstract**. I metode se mogu označiti kao apstraktne, što znači da nisu definirane (nego samo deklarirane) i da njihova definicija mora biti napravljena u (nekoj) podklasi. Ključne riječi za označavanje apstraktnih metoda su iste kao one za označavanje apstraktnih klasa. Java osim apstraktnih klasa može imati i **interface**, za koji se može reći da predstavlja potpuno apstraktnu klasu (u kojoj su sve metode apstraktne).

5. JEDNOSTRUKO NASLJEĐIVANJE, NADJAČAVANJE (OVERRIDING) METODA, POLIMORFIZAM

Nasljeđivanje je jedna od tri osnovne operacije računa klasa (class calculus). Ostale dvije su agregacija i generičnost. Klasu od koje se nasljeđuje u nastavku ćemo nazivati nadklasa, a klasu koja nasljeđuje podklasa. Naravno, to su relativni pojmovi, jer nadklasa može biti podklasa neke treće klase, a podklasa može biti nadklasa nekoj četvrtoj klasi. Inače, uobičajeni nazivi u promatranim jezicima su: PL/SQL nadtip / podtip, Java bazna klasa / derivirana klasa, Eiffel klasa roditelj / klasa potomak (ili nasljednik). Slijedi PL/SQL primjer klase "kucni_ljubimac" (samo specifikacija, bez tijela), koja nasljeđuje klasu "zivotinja":

```
CREATE OR REPLACE TYPE kucni_ljubimac UNDER zivotinja (
   omiljena_hrana VARCHAR2 (20),
   CONSTRUCTOR FUNCTION kucni_ljubimac
      (p_visina_cm NUMBER, p_ime VARCHAR2, p_omiljena_hrana VARCHAR2)
      RETURN SELF AS RESULT,
   OVERRIDING MEMBER PROCEDURE prikazi_podatke,
   FINAL MEMBER PROCEDURE pocisti
)  NOT FINAL; ...
```

Klasa "kucni_ljubimac" ima, uz naslijeđene atribute, novi atribut "omiljena_hrana". Klasa je naslijedila funkciju "visina_inch" i proceduru "prikazi_podatke" (ali ju je nadjačala) i dodana joj je nova procedura "pocisti". Napomenimo da se konstruktori ne nasljeđuju - svaka klasa ima svoj vlastiti (a može poslužiti i default). Vidimo da PL/SQL ima ključnu riječ **OVERRIDING**, kojom programer eksplicitno kaže da nadjačava određenu metodu. Eiffel ima za to riječ **redefine**. Java do verzije 1.5 nije koristila ključne riječi niti anotacije, nego se nadjačavanje izražavalo 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 i Java 1.5 (pomoću anotacije) eksplicitan način izražavanja je bolji, jer smanjuje mogućnost programerske greške.

Osvrnimo se sada i na PL/SQL ključnu riječ **NOT FINAL**. Njome označavamo da klasa nije finalna (krajnja) u hijerarhiji nasljeđivanja, tj. da može imati podklase (inače, PL/SQL default je **FINAL**). Eiffel je tek nedavno uveo tu mogućnost, a finalna klasa označava se sa **frozen**. U Javi sa finalna klasa označava sa **final, a** default u Javi je da klasa nije finalna. Vidimo da se i u deklaraciji (nove) procedure "pocisti" nalazi riječ **FINAL**. Primijenjena na metodu, ona označava da se ta metoda u podklasama ne može nadjačati (PL/SQL default za metodu je da nije finalna). Eiffel ima ekvivalentnu riječ **frozen**, Java ima **final** (Java **private** metode su implicitno **final**).

Vidjeli smo da se metoda može nadjačati (drugom) metodom. Možemo postaviti pitanja: da li se atribut može nadjačati atributom; da li se atribut može nadjačati metodom, ili obrnuto? PL/SQL odgovor na sva ova pitanja je – ne. U Javi se "običan" atribut može redefinirati kao statički atribut i obrnuto (ali, "obična" metoda se ne može nadjačati statičkom metodom, ili obrnuto, a isto vrijedi i za PL/SQL). Eiffel omogućava da se atribut nadjača atributom (koji mora biti ili istog tipa ili mora biti podtipa - covariance) i da se funkcija bez parametara nadjača atributom (isto mora vrijediti covariance).

Nadjačavanje metoda ne bi bilo naročito korisno kada ne bi postojao još jedan važan element objektnog modela – polimorfizam. "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, tj. u programu možemo napisati naredbu poput sljedeće: "varijabla_nadklase := varijabla_podklase" (polimorfično pridruživanje). Pitanje je sada 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 – iz podklase, zahvaljujući tzv. dinamičkom povezivanju (dynamic binding, a koriste se i nazivi dynamic method dispatch i virtual method dispatch).

PL/SQL i Java nemaju mogućnost razlikovanja između dvije različite vrste nasljeđivanja: nasljeđivanja tipa (zove se i semantičko nasljeđivanje, ili nasljeđivanje sučelja) i nasljeđivanja implementacije (zove se i sintaktičko nasljeđivanje, ili nasljeđivanje programskog koda). Prva vrsta nasljeđivanja je "pravo" nasljeđivanje, u kojem podklasa predstavlja podtip nadklase (klase "zivotinja" i "kucni_ljubimac" su primjer takvog nasljeđivanja). Druga vrsta nasljeđivanja je "praktično" nasljeđivanje, gdje se želi iskoristiti neki postojeći programski kod, ali se ne može koristiti polimorfizam. C++ ima obje vrste nasljeđivanja već odavno, a Eiffel je dobio nasljeđivanje implementacije tek nedavno.

Polimorfizam omogućava da objektima šaljemo poruke bez da znamo njihov točan tip (ali znamo da su podtip nekog tipa). Mogućnost da nam program za vrijeme izvođenja daje informaciju o tipu objekta obično se označava kao RTTI (run-time type information). PL/SQL za to ima operator **IS OF**, Eiffel ima sličan operator **?=**, a Java ima najbogatije mogućnosti. "Jača" osobina od RTTI je tzv. refleksivnost, kod koje program u toku izvođenja može dati i još mnogo drugih dinamičkih (meta)informacija o objektima i klasama. Java od svih promatranih jezika najbolje podržava refleksivnost.

Ponekad bismo htjeli u podklasi koristiti metodu (iz nadklase) koju smo nadjačali. Najčešće to želimo raditi upravo u metodi koja je nadjačala metodu nadklase, jer često metoda iz podklase radi nešto slično nadjačanoj metodi. Naravno, ne želimo pisati isti kod dva puta. U Javi to nije problem, jer bilo koja metoda u podklasi može pomoću "super.ime_metode ()" pozvati bilo koju metodu iz nadklase (riječ **super** bez navođenja imena metode može se koristiti samo u konstruktoru).

Eiffel ima za to riječ **Precursor**, ali se ona ne može koristiti slobodno kao što to dopušta Java, već se može koristiti samo u metodi koja je nadjačala metodu iz nadklase i time implicitno pozvati nadjačanu metodu (ne treba navoditi ime nadjačane metode).

PL/SQL nema tih mogućnosti. Međutim, ako imamo pristup programskom kodu nadklase, možemo nadklasu mijenjati (ili je unaprijed učiniti takvom) tako da u njoj napišemo "pomoćnu" **FINAL** metodu koju ćemo pozivati iz metode nadklase i metode podklase. Prikaz tog (našeg) rješenja može se naći na www.quest-pipelines.com/pipelines/plsql/tips03.htm#JUNE (Calling the Parent Object's Version of an Overridden Method).

6. PARAMETRI, PROMJENA SIGNATURE U NADJAČANOJ METODI, DEFAULT PARAMETRI

PL/SQL može uz formalni parametar imati specifikatore **IN**, **OUT** ili **IN OUT**, koji određuju da li pozvana metoda može samo čitati parametar (**IN**, to je default), samo pisati (**OUT**), ili oboje (**IN OUT**). PL/SQL šalje **IN** (aktualne) parametre kao reference, a **OUT** i **IN OUT** šalje kao vrijednosti.

Eiffel šalje parametre referentnog tipa kao reference, a **expanded** parametre šalje kao vrijednosti. Bez obzira kako se parametri šalju, Eiffel ne dozvoljava da se oni u metodi mijenjaju, niti pomoću naredbe pridruživanja "parametar := nesto", niti pomoću naredbe kreiranja objekta "create parametar;". Ali, iako Eiffel metoda ne može mijenjati objekt predstavljen parametrom, može mijenjati njegove atribute (bilo direktno, bilo pozivom drugih metoda). Budući da se PL/SQL **OUT** i **IN OUT** parametri šalju kao vrijednosti, PL/SQL se (sa objektima) ponaša isto tako.

Java samo parametre primitivnog tipa šalje kao vrijednosti, a ostale uvijek kao reference (no, i reference se šalju kao vrijednosti). Ako se uz formalni parametar upotrijebi riječ **final**, tada se parametri u Javi ponašaju isto kao u Eiffel-u. Međutim, i ako se ne upotrijebi riječ **final**, parametar se u stvarnosti ne mijenja, jer promjena ima samo lokalni karakter (vrijedi samo unutar metode). U Javi se navođenjem **final** ispred formalnog parametra zabranjuje promjena tog parametra u metodi. Međutim, ako se i ne upotrijebi riječ final, parametar se u stvarnosti ne mijenja (tj. promjena vrijedi samo unutar metode).

Zanimljivo je pitanje promjene signature, tj. pitanje da li parametri u nadjačanoj metodi mogu imati drugačiji tip od parametara u metodi iz nadklase i (ako mogu) kakav mora biti taj tip. Postoje tri (glavne) mogućnosti:

1. tip parametra se ne može mijenjati (no variance)
2. može se mijenjati tako da bude podtip u odnosu na bazni (covariance)
3. može se mijenjati tako da bude nadtip (contravariance).

Eiffel podržava covariance i za parametre metoda i za povratnu (return) vrijednost funkcije i za atribute. Java do verzije 1.4 ima u potpunosti no variance pristup, ali od verzije 1.5 (ili 5.0) podržava covariance za povratne vrijednosti funkcije. PL/SQL se ponaša kao Java 1.5.

Nekad je logičan izbor covariance (npr. u primjeru iz sljedeće točke), a nekad contravariance (pogotovo kod nasljeđivanja implementacije). Međutim, covariance i contravariance nad parametrima mogu dovesti i do određenih problema [Meyer 1997]. Giuseppe Castagna naglašava ("Object-Oriented Programming: A Unified Foundation") da ne bi trebalo birati između covariance i contravariance, već da bi OOPL trebao podržavati oboje, na adekvatan način.

Parametri u PL/SQL metodama mogu imati default vrijednosti, pa se PL/SQL može neočekivano ponašati kada u podklasi nadjačamo metodu koja ima default parametar, te nakon polimorfičnog pridruživanja pozivamo metodu bez navođenja vrijednosti default parametra. Slijedi PL/SQL primjer (bez tijela klasa):

```
CREATE OR REPLACE TYPE roditelj AS OBJECT (
   ime VARCHAR2 (10),
   MEMBER PROCEDURE prikazi (p_default VARCHAR2 := 'X')
)  NOT FINAL;
/
CREATE OR REPLACE TYPE dijete UNDER roditelj (
   OVERRIDING MEMBER PROCEDURE prikazi (p_default VARCHAR2 := 'Y')
);
/
DECLARE
   l_roditelj roditelj := roditelj ('RODITELJ');
   l_dijete   dijete   := dijete   ('DIJETE');
BEGIN
   l_roditelj.prikazi_default; -- OK, prikaže "X" iz "roditelj"
   l_dijete.prikazi_default;   -- OK, prikaže "Y" iz "dijete"
   l_roditelj := l_dijete;     -- polimorfično pridruživanje
   l_roditelj.prikazi_default; -- IZNENAĐENJE!
   -- prikaže "X", iako (što je OK) izvršava metodu iz "dijete"
END;
```

7. PREOPTEREĆENJE (OVERLOADING) METODA

PL/SQL i Java podržavaju preopterećenje metoda, tj. mogućnost da u klasi postoji više metoda istog imena, ali različite signature. PL/SQL je i prije nego što je dobio objektne mogućnosti podržavao preopterećenje procedura i funkcija u paketima. Meyer drži da je to dobra stvar za ne-OOPL jezike, ali da u objektnom modelu preopterećenje metoda donosi nepotrebne komplikacije (Eiffel ne dozvoljava čak niti da imena parametara i lokalnih varijabli budu jednaka imenima atributa/metoda). Naravno, budući da PL/SQL i Java konstruktori moraju imati isto ime kao i klasa, preopterećenje metoda je jedini način na koji oni mogu imati više različitih konstruktora u klasi. Usput, naglasimo da je preopterećenje metoda nešto sasvim drugo od preopterećenja operatora, što je mogućnost da programer definira vlastiti operator (prefiksni, infiksni ili sufiksni) kao drugo ime za svoju metodu. Eiffel podržava preopterećenje operatora, Java ne podržava. Oracle ima "user-defined" operatore (za SQL).

Problema sa preopterećenjem metoda može biti i bez OO mogućnosti. Npr. ako propustimo na bazu sljedeći PL/SQL paket (navodimo samo specifikaciju), prevođenje će biti uspješno, ali će se kod izvođenja javiti greške PLS-00307: too many declarations of 'X' match this call:

```
CREATE OR REPLACE PACKAGE proba_overloading AS
   PROCEDURE proc (a NUMBER, b NUMBER);                -- procedura 1
   PROCEDURE proc (a NUMBER, b VARCHAR2);              -- procedura 2
   PROCEDURE proc (a NUMBER, d NUMBER);                -- procedura 3
   PROCEDURE proc (a NUMBER, b NUMBER, c NUMBER := 0); -- procedura 4
   FUNCTION  func RETURN NUMBER;                       -- funkcija 1
   FUNCTION  func RETURN VARCHAR2;                     -- funkcija 2
END;
/
DECLARE
   broj NUMBER;
BEGIN
   proba_overloading.proc (10, 20);           -- procedura 1, 3, 4?
   proba_overloading.proc (a => 10, b => 20); -- procedura 1, 4?
   broj := proba_overloading.func;            -- funkcija  1, 2?
END;
```

Isto bi se desilo da umjesto paketa napravimo ekvivalentnu PL/SQL klasu. Za razliku od PL/SQL prevodioca, Java prevodilac neće dozvoliti niti jednu grešku u toku izvođenja programa - neće istovremeno propustiti obje procedure 1 i 3 (jer se razlikuju samo po imenima parametara), neće propustiti obje funkcije (jer se razlikuju samo po vrijednosti koju vraćaju) i ne dozvoljava default vrijednosti parametara, pa ne propušta proceduru 4.

Problem se može javiti i kada kombiniramo preopterećenje metoda i polimorfizam. Prikaz (naš) tog problema može se naći na www.quest-pipelines.com/newsletter-v4/0503_C.htm (Dynamic Method Dispatch and Method Overloading in a Subtype).

8. VIŠESTRUKO NASLJEĐIVANJE

PL/SQL, nažalost, ne podržava višestruko nasljeđivanje klasa, tj. mogućnost da klasa nasljeđuje dvije ili više klasa. Neki drže da višestruko nasljeđivanje uopće nije potrebno, da je to samo dodatna komplikacija. Drugi pak drže višestruko nasljeđivanje vrlo važnim svojstvom OOPL jezika. Npr. Meyer drži da je u mnogim konkretnim situacijama potrebno da klasa može naslijediti dvije ili više klasa i duhovito kaže da je odgovor na pitanje "Da li moja klasa C treba naslijediti klasu A ili klasu B (budući da moj jezik podržava samo jednostruko nasljeđivanje)?" često isto tako težak kao i odgovor na pitanje "Da li da izaberem mamu ili tatu?".

Nije stoga čudno da Eiffel od početka podržava višestruko nasljeđivanje (npr. C++ također podržava višestruko nasljeđivanje, ali je ono naknadno uvedeno i zbog toga nije tako dobro riješeno kao u Eiffel-u). Smatra se da Eiffel bolje nego bilo koji drugi OOPL podržava (barem) dvije stvari – jedna je višestruko nasljeđivanje, a druga je Design by Contract. Višestruko nasljeđivanje je naročito korisno u slučaju kada postoje dva (ili više) jednako važna kriterija za kreiranje hijerarhije klasa (jednostruko nasljeđivanje dopušta samo jednu hijerarhiju) i u slučaju kada podklasa od jedne nadklase nasljeđuje tip, a od druge nadklase nasljeđuje implementaciju, tzv. mix-in nasljeđivanje (primjer: klasa ARRAYED_STACK nasljeđuje od apstraktne klase STACK i klase ARRAY).

Slijedi Eiffel (nepotpuni) primjer klase C koja nasljeđuje od klasa A i B, pri čemu klasa C nadjačava jedan atribut i jednu metodu iz klase A i jednu metodu iz klase B:

```
class C
inherit
   A redefine atribut_iz_a, metoda_iz_a end
   B redefine metoda_iz_b end
feature
   ...
end
```

Kod višestrukog nasljeđivanja najčešće se navode dva glavna problema. Jedan je problem kada roditeljske klase imaju atribute/metode istog imena, ali različitog značenja. Eiffel za to ima vrlo jednostavno rješenje – preimenovanje (barem jednog) atributa/metode pomoću **rename**. Drugi je problem kada imamo tzv. ponavljajuće (repeated) nasljeđivanje, npr. u primjeru gdje su klase B i C djeca klase A, a klasa D je dijete i od B i od C, pa izlazi da je klasa D dva puta (indirektno) dijete od klase A (to se naziva i dijamantnim nasljeđivanjem, zbog sličnosti crteža takvih klasa sa skicom dijamanta). Ako atributi/metode klase A nisu nadjačani u klasama B i C, Eiffel za to ima najjednostavnije moguće rješenje – ne treba ništa napraviti, jer duplih atributa/metoda niti nema. Ako su pak atributi/metode iz klase A redefinirani u klasi B i/ili C, tada postoje dva slučaja – da je kod nadjačavanja u klasama B/C zadržano isto ime atributa/metode kao u klasi A, ili da je ime promijenjeno. Prvi slučaj (ista imena) se najčešće svodi na drugi, preimenovanjem jednog atributa/metode (a rjeđe se koristi "eliminacija" jednog atributa/metode, pomoću **undefine**). U drugom slučaju (različita imena) Eiffel traži da se eksplicitno izabere (pomoću **select**) željeni atribut/metoda, što je potrebno da bi se razriješila dilema izbora prave metode kod dinamičkog pozivanja metoda (zbog polimorfičnog pridruživanja).

Java ne podržava višestruko nasljeđivanje klasa, ali podržava višestruko nasljeđivanje sučelja ili istovremeno nasljeđivanje jednog (ili više) sučelja i samo jedne klase. To znači da npr.: sučelje S3 može naslijediti sučelja S1 i S2, klasa K1 može naslijediti sučelja S1 i S2, klasa K2 može naslijediti sučelje S1 i klasu K1. To nije loše, ali je problem u tome da su sučelja potpuno apstraktne klase, tj. sve metode sučelja su apstraktne. Zbog toga se u Javi, u nedostatku višestrukog nasljeđivanja klasa, često koriste unutarnje (inner) klase. Poseban podskup Java unutarnjih klasa su tzv. statičke unutarnje klase, koje se nazivaju i ugniježđenim klasama. Mnogi drže da su unutarnje klase nepotrebno kompliciranje (relativno jednostavnog) objektnog modela i da bi bilo bolje da Java umjesto njih ima (pravo) višestruko nasljeđivanje klasa. PL/SQL nema unutarnje klase.

9. GENERIČKE KLASE (PREDLOŠCI)

Generičke klase su mehanizam za koji mnogi drže da je čak važniji od višestrukog nasljeđivanja. Eiffel je imao generičke klase od početka, dok ih je C++, pod imenom predlošci (templates), dobio 1991. Java ima generičke klase od verzije 1.5. PL/SQL nema generičke klase, a nema niti generičke pakete (šteta, jer ADA 83 ima generičke pakete).

Možemo reći da generičke klase zapravo nisu prave klase, nego predlošci za klase, jer imaju tzv. generičke parametre. Npr. u Eiffel-u ovako izgleda generička klasa STACK [G] (generički parametar nazvan je G, a može se zvati i drugačije):

```
class STACK [G]
feature
   element_na_vrhu: G is
      do
         ...
      end

   stavi_na_vrh (p_element: G) is
      do
         ...
      end
   ...
end
```

Generička klasa se koristi kao klijent neke (druge) klase, u kojoj se (drugoj klasi) atribut, varijabla ili parametar deklarira pomoću generičke klase, tako da se generički parametar zamijeni sa nekom (trećom) klasom. Npr. klasa STACK_KLIJENT može sadržavati dva atributa definirana na sljedeći način (generički parametar G zamijenjen je klasom ZIVOTINJA, odnosno klasom REAL):

```
atribut1: STACK [ZIVOTINJA]
atribut2: STACK [REAL]
```

Eiffel ima (i Java 1.5, a C++ nema) i tzv. ograničenu generičnost (constrained genericity), gdje se generički parametar ograničava nekom određenom klasom, pomoću operatora ->. U slučaju ograničene generičnosti, klasa koja zamjenjuje generički parametar mora biti ili ista kao klasa koja ograničava generički parametar, ili podklasa te klase. Npr. ako bismo ovako definirali (ograničenu) generičku klasu STACK:

```
class STACK [G -> ZIVOTINJA] ... end
```

tada bismo imali sljedeće:

```
atribut1: STACK [BAKTERIJA]      -- greška, nije podklasa od ZIVOTINJA
atribut2: STACK [ZIVOTINJA]      -- OK, može biti ista klasa
atribut3: STACK [KUCNI_LJUBIMAC] -- OK, to je podklasa od ZIVOTINJA
```

10. DESIGN BY CONTRACT

Design by contract (u nastavku DBC) je ideja čiji je autor također Meyer, ali korijeni ideje su u 70-im, kada su razvijene formalne metode u razvoju softvera (jedan od pionira bio je Edsger W. Dijkstra). Meyer je prije dizajniranja Eiffel-a sudjelovao u razvoju formalnih specifikacijskih jezika Z i M. Stoga nije čudno da Eiffel u potpunosti podržava DBC. Za sada niti jedan drugi OOPL ne podržava DBC u potpunosti, barem ne na razini jezika. DBC je opširno opisan u [Meyer 1997]. Može se reći da DBC ima dosta dodirnih točaka sa metodama analize, dizajna i izrade poslovnih pravila (business rules), koje sve više dobivaju na popularnosti i za koje postoji dosta bogata literatura.

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 jedan dodatni dio - invarijantu (invariant). Ugovor (contract) 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). Ideja je na neki način upravo suprotna od tzv. defanzivnog programiranja, koje zagovora da se u svim mogućim trenucima pokušava što više toga provjeriti.

Eiffel za specifikaciju DBC elemenata koristi ključne riječi **require** (odnosno **require else** u nadjačanoj metodi) za označavanje pretkondicije, **ensure** (odnosno **ensure then** u nadjačanoj metodi) za postkondicije i **invariant** za invarijante klasa. Navedene naredbe su najvažnije za DBC podršku, ali Eiffel ih ima još. Naredba **check** služi za provjeru nekog uvjeta u bilo kom trenutku. Za provjeru programskih petlji postoje dvije naredbe: **variant** provjerava da li se cjelobrojni izraz smanjuje kod svakog prolaza u petlji, a **invariant** (opet ista riječ, ali je u kontekstu petlje značenje drugačije) provjerava da li je određeni uvjet u petlji zadovoljen u svakom prolazu.

Može se postaviti pitanje utjecaja izvršenja pretkondicija, postkondicija i invarijanti na brzinu izvođenja programa. Nažalost, utjecaj postoji, a naročito je skupa provjera invarijanti. Stoga se u Eiffel-u može odrediti nekoliko stupnjeva provjere. Najslabija provjera (ali sa najmanjim negativnim utjecajem na brzinu izvođenja) je provjera samo pretkondicija (ta se provjera obično ostavlja i nakon završetka faze testiranja, tj. ostaje u produkcijskom kodu). Sljedeći stupanj uključuje provjeru postkondicija, slijedi provjera invarijanti, zatim provjera petlji i na kraju provjera pomoću **check** naredbi. Najveći stupanj provjere se obično koristi samo kod testiranja, jer se takvi programi izvršavaju i do 2-3 puta sporije u odnosu na programe koji nemaju nikakvih provjera.

Treba naglasiti da su pretkondicije, postkondicije i invarijante korisne čak i kada nisu realizirane kao programski kod, nego samo kao komentar (najčešće zato što je nešto teško ili nemoguće izraziti – Eiffel nema operatore univerzalnog i egzistencijalnog kvantifikatora iz predikatnog računa), jer poboljšavaju dokumentiranost izvornog koda. Zbog DBC podrške, može se reći da je Eiffel i specifikacijski (a ne samo programski) jezik.

Java je u verziji 1.4 dobila naredbu **assert** (kao Eiffel **check**), koja omogućava određenu podršku za DBC, ali (relativno) potpuna podrška za DBC u Javi za sada postoji samo na razini nezavisnih alata (kao što je iContract alat, kojeg je 1999. napravio Reto Kramer). PL/SQL za sada ne podržava DBC. I poznati grafički jezik UML (danas de-facto standard za modeliranje OO sustava) podržava DBC od 1999., zahvaljujući OCL (Object Constraint Language) formalnom jeziku (čiji su autori Jos Warmer i Anneke Kleppe). Nešto više o tome može se naći u knjizi [Jezequel 2000], koja prikazuje izvorna 23 GoF (Gang of Four) uzorka dizajna (design patterns). Za razliku od GoF originala ("Design Patterns - Elements of Reusable Object-Oriented Software"), [Jezequel 2000] za prikaz primjera koristi Eiffel (a ne C++), te upotpunjuje primjere sa DBC elementima.

11. OBRADA IZNIMAKA (EXCEPTION HANDLING)

Uza sve provjere, u toku izvođenja programa može doći do događaja koji zovemo iznimka (exception), koji može uzrokovati da pozvana metoda (procedura/funkcija) završi sa neuspjehom, ako iznimka nije u njoj obrađena. Ako metoda ne uspije, dolazi do iznimke u metodi koja ju je pozvala. Jedan od prvih jezika koji je imao obradu iznimaka bio je IBM-ov PL/1.

PL/SQL ima gotovo isti mehanizam obrade iznimaka kao i ADA, dok je Eiffel-ov modificiran (u odnosu na ADA). Uvođenje objektnih mogućnosti u PL/SQL nije donijelo ništa novo u obradu iznimaka, tj. PL/SQL i dalje ima sistemske (Oracle) iznimke i programerski-definirane iznimke, koje programer eksplicitno deklarira i poziva sa **RAISE**:

```
DECLARE
   prekoracenje_limita EXCEPTION; -- deklaracija programerske iznimke
BEGIN
   ...
   IF minus_na_racunu > max THEN
      RAISE prekoracenje_limita
   END IF;
   ...
EXCEPTION
   WHEN ZERO_DIVIDE THEN
      ...; -- obrada sistemske iznimke
   WHEN prekoracenje_limita THEN
      posudi_od_mame;
END;
```

Eiffel kod bio bi sličan (**EXCEPTION** se tamo zove **rescue**), ali postoji jedna važna razlika. Ako u Eiffel-u želimo postići da se iznimka uspješno obradi, moramo u rescue dijelu koristiti naredbu **retry**, kojom vraćamo izvršavanje na početak metode. Eiffel-ov pristup je: obrada iznimke mora ili svesti stanje na ono koje je bilo na početku metode i pokušati opet otpočetka, ili završiti neuspješno. Primijenjeno na PL/SQL, to bi značilo da u **EXCEPTION** dijelu moramo ili izvršiti **RAISE** (i time ponovno aktivirati iznimku) ili vratiti se na početak metode (to je malo teže izvesti, jer PL/SQL nema **retry**, a dobri stari **GOTO** ne može skakati između **EXCEPTION** i izvršnog dijela bloka). Napomenimo da u Eiffel-u 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 razliku od PL/SQL iznimaka, Java iznimke su objekti. C++ iznimke mogu biti objekti, ali ne moraju, a slično vrijedi i za Eiffel, koji do nedavno nije imao iznimke kao objekte. Java mehanizam obrade iznimaka zasniva se na naredbama **try** (označavanje početka koda u kojem se može dogoditi iznimka), **throw** (kao RAISE), i **catch** (kao **WHEN** u PL/SQL **EXCEPTION**). Java može imati i **finally** dio, koji se uvijek izvršava, neovisno da li ima iznimaka (nije kao **WHEN OTHERS** u PL/SQL). Java metoda mora imati i tzv. specifikaciju iznimaka - u deklaraciji metode (iza parametara) navede se riječ **throws**, sa popisom iznimaka koje se u metodi mogu desiti. Java prevodilac zahtijeva da se u specifikaciji navede iznimka koja se može desiti u metodi, a nije u njoj obrađena (ali, ne buni se ako je u specifikaciji iznimka koja se ne može desiti). Java pristup, tj. provjera kod prevođenja (poznato kao checked exceptions) naizgled izgleda dobro, ali mnogi drže da se u praksi taj pristup pokazao kao loš (jedan od njih je i Bruce Eckel [Eckel 2003]).

12. ZAKLJUČAK

Oracle nas je u bazi 9i ugodno iznenadio objektnim mogućnostima PL/SQL (i SQL) jezika. Držimo da je u PL/SQL puno bolje ugrađen objektni model, nego što je to napravljeno sa jezikom ADA (jezik iz kojeg je PL/SQL nastao) u verziji ADA 95. Vjerujemo da je put daljnjem razvoju i usavršavanju objektnih mogućnosti PL/SQL jezika otvoren i nadamo se da će Oracle u sljedećim verzijama napraviti barem neka od ovih poboljšanja:

- uvođenje privatnih metoda, po ugledu na privatne procedure/funkcije u tijelu PL/SQL paketa; uvođenje privatnih atributa (ako i bude moguće) vjerojatno će biti otežano, jer je pitanje kako privatne atribute uklopiti u sadašnji sustav kontrole pristupa nad objektima baze (objektima u širem smislu, ne u smislu instance klase); naravno, bilo bi još ljepše kada bi PL/SQL uveo vrlo fleksibilan sustav kontrole pristupa koji ima Eiffel, ali držimo da je to teško očekivati
- uvođenje referenci za tranzijentne objekte, u smislu da ako klasa K1 ima atribut k2 koji se odnosi na klasu K2, taj atribut može (ako to programer želi) sadržavati referencu na objekt klase K2, a ne cijeli (pod)objekt; naravno, to bi (vjerojatno) uvelo dodatnu komplikaciju dizajnerima jezika – potrebu za automatskim sakupljanjem smeća, jer ne želimo da programer mora sam (kao u C++) voditi brigu o brisanju nepotrebnih objekata; također bi bilo lijepo kada bi konstruktor metode mogle imati slobodno definirana imena (kao u Eiffel-u) i ne ovisiti (ako konstruktora ima više) o mehanizmu preopterećenja metoda
- dodati mogućnost razlikovanja između dvije različite vrste nasljeđivanja: nasljeđivanja tipa i nasljeđivanja implementacije, što C++ ima već odavno, a nedavno je nasljeđivanje implementacije dobio i Eiffel
- uvođenje višestrukog nasljeđivanja; to će vjerojatno ići vrlo teško, jer niti u svijetu Jave (za koju je Oracle vrlo vezan) nema nekog većeg govora o mogućnosti uvođenja višestrukog nasljeđivanja između klasa (no nikad se ne zna - i C++ je dobio višestruko nasljeđivanje naknadno)
- generičke klase možda imaju veću šansu za uvođenje u PL/SQL, iz (barem) dva razloga: Java ih je dobila u verziji 1.5 (Eiffel ih ima od početka, a u C++ su uvedene 1991.) a jezik ADA (od kojeg je PL/SQL puno toga naslijedio) ima generičke pakete od početka (generičnost je prvi uveo Algol 68)
- uvođenje elemenata koncepta "Design by Contract", po uzoru na Eiffel; nešto od toga se možda i može očekivati, jer je i Java u verziji 1.4 uvela neke elemente tog koncepta.

Literatura:

1 Oracle 9i / 10g priručnici:
  a) PL/SQL User's Guide and Reference
  b) Application Developer's Guide – Object-Relational Features
2 Bruce Eckel: Thinking in C++, 2. izdanje, Prentice Hall, 2000.
3 Bruce Eckel: Thinking in Java, 3. izdanje, Prentice Hall, 2003.
4 Steven Feuerstein: Oracle PL/SQL Programming, O'Reilly, 2002.
5 Jean-Marc Jezequel i drugi: Design Patterns and Contracts, Addison-Wesley, 2000.
6 Thomas Kyte: Expert One-on-One Oracle, Apress, 2001.
7 Ian Joyner: Objects unencapsulated – Java, Eiffel and C++??, Prentice Hall, 1999.
8 Craig Larman: Applying UML and Patterns, 2. izdanje, Prentice Hall, 2002.
9 Bertrand Meyer: Object-Oriented Software Construction, 2.izdanje, Prentice Hall, 1997.
10 ECMA International, Standard ECMA-367: Eiffel Analysis, Design and Programming Language, June 2005