Pretvorbe u *Java* programu

Matej Mihelčić

Prirodoslovno-matematički fakultet, Sveučilište u Zagrebu

matmih@math.hr

18. listopada, 2022.

Tip i kontekst

Izvršavanjem izraza u programskom jeziku *Java* ili **ne dobijemo nikakav rezultat** ili **rezultat ima tip** koji se **može utvrditi tijekom prevođenja**. Svaki izraz se promatra u određenom **kontekstu** (uzima se u obzir **širi skup informacija** o tipovima i povezanim izrazima), stoga tip izraza **mora biti kompatibilan** s **očekivanim (tzv. ciljnim) tipom** u kontekstu.

Postoje dva način da se osigura kompatibilnost izraza s odgovarajućim kontekstom: a) tip izraza se može **definirati ovisno o ciljnom tipu** (tj. isti izraz može ovisno o kontekstu imati različiti tip), b) nakon određivanja tipa izraza, **može se izvršiti implicitna pretvorba tipa** u ciljni tip.

Ukoliko niti jedna strategija ne može postići da rezultat bude odgovarajućeg ciljnog tipa, **dolazi to greške pri prevođenju**.
Tipovi pretvorbe u programskom jeziku *Java*:

- Pretvorba identiteta - konverzija tipa u taj **isti tip** (uključujući operator pretvorbe tipa u isti tip).

Pretvorbe

- Osnovna proširujuća pretvorba - **ne gubi informaciju o veličini numeričke vrijednosti** osim kada radimo pretvorbu `float -> double` bez korištenja `strictfp`. Pri pretvorbi `int -> float`, `long -> float`, `long -> double` može doći do **gubitka preciznosti**.
- Osnovna sužujuća pretvorba - **može doći do gubitka informacije o veličini numeričke vrijednosti, preciznosti i raspona**. Konverzija `float -> T`, gdje je `T` cjelobrojni tip se provodi u dva koraka: a) `float -> long` (za `T = long`) ili `float -> int` (inače), b) `S->T`, gdje `S` reprezentira tip iz koraka a) (`long` ili `int`).

  Posebna vrsta konverzije je `byte -> char` zato što se odvija na način: `byte -> int` (osnovna proširujuća pretvorba), b) `int -> char` (osnovna sužujuća pretvorba).

  Pretvorbe osnovnih tipova **ne proizvode iznimke pri izvršavanju programa**.

Pretvorbe

- Proširujuća pretvorba referenciranog tipa - postoji između svaka dva tipa `S` i `T` takva da je `S` **podtip** od `T`.
- Sužujuća pretvorba referenciranog tipa (`S` u `T`) - tretira izraz referenciranog tipa `S` kao **izraz drugačijeg referenciranog tipa** `T`, gdje `S` **nije podtip** od `T`. **Posebna pravila** određuju kada se može primjeniti navedena pretvorba:
  - `S` nije podtip od `T`
  - Ukoliko postoje parametrizirani tipovi `X` i `Y`, gdje `X` je nadtip od `T`, `Y` je nadtip od `S` takvi da su fiksirani tipovi od `X` i `Y` jednaki, tada `X` i `Y` **ne smiju biti dokazivo različiti**.
  - `S` i `T` su tipovi klasa i vrijedi: $|S| <: |T|$ ili $|T| <: |S|$.
  - `S` i `T` su tipovi sučelja ili `S` je tip klase koja nije `final`, `T` je tip sučelja.
  - `S` je tip klase, `T` je tip sučelja, `S` je klasa tipa `final` koja nasljeđuje sučelje `T` (vrijedi i ako je T klasa a S sučelje).
  - `S` je klasa tipa `Object` ili sučelje tipa `java.io.Serializable` ili `Cloneable` a `T` je tip polja.
  - `S` je polje kojem su komponente tipa `SC`, `T` je polje kojem su komponente tipa `TC` i postoji sužujuća pretvorba iz `SC` u `TC`.

Pretvorbe

- Sužujuća pretvorba referenciranog tipa (`S` u `T`)
  - `S` je varijabla tipa i postoji sužujuća konverzija reference gornje granice tipa `S` u `T`.
  - `T` je varijabla tipa i postoji ili proširujuća ili sužujuća konverzija iz `S` u gornju granicu tipa `T`.
  - `S` je tip presjeka $S_1 \& S_2 \ldots \& S_n$ i $(\forall i \in [1, n])$ postoji ili proširujuća ili sužujuća konverzija iz S_i u `T`.
  - `T` je tip presjeka $T_1 \& T_2 \ldots \& T_n$ i $(\forall i \in [1, n])$ postoji ili proširujuća ili sužujuća konverzija iz `S` u T_i.

Primjeri:

- Ne postoji sužujuća konverzija reference iz tipa `ArrayList<String>` i tip `ArrayList<Object>` ili obrnuto zato što su tipovi `String` i `Object` **dokazivo različiti**.
- Sužujuća konverzija između tipova `ArrayList<T>` i `ArrayList<Object>` je moguća zato što je `T` varijabla tipa s gornjom ogradom `Object` a varijabla tipa od `ArrayList<Object>` je `Object`. Pošto `Object<:Object`, tipovi nisu dokazivo različiti i zadovoljavaju uvjete.

Pretvorbe

Sužujuća pretvorba referenciranog tipa može biti **provjerena ili neprovjerena** (ovisno o tome može li se utvrditi korektnost tipova pretvorbe). Neprovjerene sužujuće pretvorbe referenciranog tipa su:

- Sužujuća pretvorba reference iz tipa `S` u parametriziranu klasu ili sučelje tipa `T` osim kada:
  - Svi argumenti tipa od `T` su neodređeni tipovi.
  - `T` $<:$ `S` i `S` nema podtip `X` osim `T` ako argumenti tipa `X` nisu sadržani u tipovima argumenata `T`.
- Sužujuća pretvorba tipa `S` u varijablu tipa `T` je neprovjerena.
- Neprovjerene sužujuće pretvorbe referenciranog tipa su:
  - Sužujuća pretvorba referenciranog tipa iz tipa `S` u tip presjeka $T_1\&, \ldots \& T_n$ je neprovjeren ako postoji T_i, $1 \leq i \leq n$ takav da S nije podtip od T_i i sužujuća pretvorba referenci iz S u T_i je neprovjerena.

    Neprovjerena konverzija uzrokuje `unchecked warning` koji se može utišati postavljanjem anotacije `@SuppressWarnings`.

Pretvorbe

- Pretvorba pakiranja - pretvara izraz osnovnog tipa u odgovarajući referencirani tip na način da poziv `r.tipValue() == p`, gdje je `r` objekt referenciranog tipa, `p` varijabla osnovnog tipa a `tip` je osnovni tip varijable `p`. Iz varijable osnovnog tipa vrijednosti `NaN`, se stvara referencirani tip tako da `r.isNaN()` vraća vrijednost `true`.
- Pretvorba otpakiravanja - pretvara izraz referenciranog tipa u odgovarajući osnovni tip (vraća vrijednost `r.tipValue()`).
- Neprovjerene pretvorbe - neka je `G` deklaracija generičkog tipa s n parametara tipa. Postoji neprovjerena konverzija iz nerafiniranog tipa klase ili sučelja `G` u proizvoljni parametrizirani tip oblika $G < T_1, \ldots, T_n >$.

Pretvorbe

- Neprovjerene pretvorbe - postoji neprovjerena pretvorba nerafiniranog tipa $G[]^k$ u bilo koji tip polja oblika $G < T_1, \ldots, T_n[] >^k$ ($[]^k$ označava k-dimenzionalno polje).
- Pretvorbe kod definiranja parametara tipa. Neka je `G` ime generičkog tipa s n varijabli tipa $A_1, \ldots A_n$ s odgovarajućim granicama $U_1, \ldots, U_n$. Parametrizirani tip $G < T_1, \ldots, T_n >$ se može pretvoriti u $G < S_1, \ldots, S_n >$, gdje $1 \leq i \leq n$ ako:
  - Ako je T_i argument nedefiniranog tipa, tada je S_i varijabla čija je gornja granica U_i (uz substituciju $[A_1 := S_1, \ldots, A_n := S_n]$ a donja granica `null`.
  - Ako je T_i argument nedefiniranog tipa oblika `? extends` B_i, tada je S_i varijabla tipa čije je gornja granica $B_i \& U_i$ (uz supstituciju $[A_1 := S_1, \ldots, A_n := S_n]$) a donja granica je tip `null`.
  - Ako je T_i neodređeni tip oblika `? super` B_i, tada je S_i varijabla tipa čija je gornja granica $[A_1 := S_1, \ldots, A_n := S_n]$ a donja granica je B_i.
  - Inače $S_i = T_i$.

Pretvorbe i konteksti pretvorbe

- Pretvorbe u string - pretvara bilo koji tip u tip `String`. Vrijednost x osnovnog tipa `T` se prvo pretvori u referenciranu vrijednost na način da se s tom vrijednošću pozovu izrazi kreiranja instanca klase, nakon čega se referencirana vrijednost pretvara u `String`. Za referenciranu vrijednost `null` se stvara string "null" a za ostale referencirane vrijednosti se poziva metoda `toString`.
- Pretvorbe između skupa vrijednosti - proces mapiranja vrijednosti tipa `float` iz jednog skupa vrijednosti u drugi bez promjene tipa. Nudi mogućnosti pri implementaciji jezika *Java* (**ne postoji od Java verzije 17**).

Postoji **šest vrsta konteksta pretvorbe**:

- Kontekst pridruživanja - vrijednost izraza je **vezana uz imenovanu varijablu**. Može doći do **širenja osnovnih i referenciranih tipova**, vrijednosti se mogu **pakirati ili otpakirati** a neki osnovni konstantni izrazi se mogu **suziti**. Može doći i do **neprovjerenih pretvorbi**.

Konteksti pretvorbe

- Kontekst strogog poziva - argument je **vezan uz formalni parametar konstruktora** ili **metode**. Može doći do **proširenja** osnovnog ili referenciranog tipa i do **neprovjerene pretvorbe**.
- Kontekst slobodnog poziva - argument je vezan uz formalni parametar. Javlja se ukoliko se ne može pronaći odgovarajuća stroga definicija metode ili konstruktora. Uz **proširenje** i **neprovjerene pretvorbe**, ovdje može doći i do **pakiranja** te **otpakiravanja**.
- Kontekst stringa - vrijednost bilo kojeg tipa se pretvara u **objekt tipa String**.
- Kontekst pretvorbe tipa - vrijednost izraza se pretvara u tip zadan operatorom pretvorbe tipa. Dopušta **sve tipove pretvorbe** osim pretvorbu u **string**. Neke pretvorbe referenciranih tipova se provjeravaju za vrijeme izvršavanja programa.
- Numerički kontekst - operandi numeričkih operatora ili neki drugi izrazi koji se primjenjuju na brojeve mogu biti **prošireni** na zajednički tip.

Primjeri pretvorba i konteksta pretvorbi

```
1   int a=5;
2   int b;
3
4   b=a; //pretvorba identiteta
5        //pretvorba pridruzivanja
6
7   b = (int) a; //pretvorba identiteta
8                //kontekst pretvorbe tipa
9                //pretvorba identiteta
10               //kontekst pridruzivanja
11
12  float c = a; //osnovna prosirujuca pretvorba
13               //kontekst pridruzivanja
14
15  int d = (int) c; //osnovna suzujuca pretvorba
16                   //kontekst pretvorbe tipa -> pi -> kp
17
18  Integer e = d; //pretvorba pakiranja
19                 //kontekst pridruzivanja
```

Razni tipovi konteksta pretvorbe

Primjeri pretvorba i konteksta pretvorbi

```
1   Object f = e; //prosirujuca pretvorba referenciranog tipa
2                 //kontekst pridruzivanja
3   float g = c+a; //osnovna prosirujuca pretvorba,
4                  //numericki kontekst
5                  //pretvorba identiteta,
6                  //kontekst pridruzivanja
7
8   String h = g+""; //pretvorba u string
9                    //kontekst stringa
10
11  ArrayList<Integer> i = new ArrayList(); //neprovjerena
12                                          //pretvorba
13  ArrayList j = new ArrayList(); //nerafinirani tip
14  ArrayList<Integer> k = j;      //neprovjerena pretvorba
15
16  ArrayList<Double> l = new ArrayList<>();
17        double m = Math.PI;
18        l.add(m);//pretvorba pakiranja (double -> Double)
19
```

Razni tipovi konteksta pretvorbe

Strojni jezik *Java virtualnog stroja*

Strojni kod *Java virtualnog stroja* možemo generirati koristeći `javap` alat dostupan u *Java Development Kit*-u.

```
1 <index> <opcode> [ <operand1> [ <operand2>... ]] [<comment>]
2
```

Oblik instrukcije Java virtualnog stroja

- $<$ index $>$ - indeks operacijskog koda instrukcije (eng. *opcode*) u polju bajtova *Java virtualnog stroja* dane metode (pomak u byte-ovima od početka metode).
- $<$ opcode $>$ - mnemonic za operacijski kod instrukcije.
- $<$ operandI $>$ - operandi instrukcije.
- $<$ comment $>$ - komentar na kraju instrukcije.

Primjer prevođenja *Java* koda u jezik *Java virtualnog stroja*

```
  void petlja() {
    int i;
    for (i = 0; i < 100; i++) {
        ;    // prazno tijelo petlje
    }
}//java kod

Method void petlja()//jedan moguci prijevod
0   iconst_0     // stavi int konstantu 0 na stog
1   istore_1     // spremi u lokalnu varijablu 1 (i=0)
2   goto 8       // skoci na instrukciju na indeksu 8 (prvi
    puta ne radi inkrement)
5   iinc 1 1     // inkrement lokalne varijable 1 za 1 (i++)
8   iload_1      // stavi lokalnu varijablu 1 (i) na stog
9   bipush 100   // stavi lokalnu konstantu 100 na stog i
    dohvati kao operand
11  if_icmplt 5 // usporedi i skoci na instrukciju 5 (ponovi
     petlju) ako (i < 100)
14  return       // prazni return
```

Primjer Java koda i jednog mogućeg prijevoda za Java virtualni stroj

Primjer prevođenja *Java* koda u jezik *Java virtualnog stroja*

```
iconst_<m1>, iconst_<{0,..,5}> //instrukcija za stavljanje
   konstanti -1 (prva) ili {0,1,2,3,4,5} (druga instrukcija
    s odgovarajucim indeksom) na stog.
istore_<n>  //n mora biti indeks na lokalno polje varijabli
    u trenutnom okviru, vrijednost na vrhu parametarskog
   stoga mora biti tipa int. Parametar se skida sa stoga i
   sprema u lokalnu varijablu s indeksom n.
 goto branchbyte1 branchbyte2 //koristi 2 byte-a za
   racunanje offseta sljedece instrukcije (ukupno 3 byte-a)
 iinc ind _const //1 byte za index (koji mora biti unutar
   polja lokalnih varijabli trenutnog okvira), 1 byte za
   konstantu, 1 byte za kod instrukcije. Inkrementira int
   varijablu na indeksu index za konstantu _const.
 iload_<k> //indeks (unsigned byte) mora biti index u polju
    lokalnih varijabli trenutnog okvira. Varijabla na
   zadanom indeksu mora sadrzavati int. Vrijednost
   varijable se stavlja na stog operanada.
```

Primjer Java koda i jednog mogućeg prijevoda za Java virtualni stroj

Primjer prevođenja *Java* koda u jezik *Java virtualnog stroja*

```
bipush _byte // 1 byte za operand, 1 za kod instrukcije.
  Byte je prosiren do vrijednosti int i stavlja se na stog
   operanada.
if_icmp<cond> branchbyte1 branchbyte2 //testira uvjet <
  cond> te skace na sljedeci index racunajuci pomak (
  offset) od trenutnog indeksa. Pomak se racuna koristeci
  branchbyte1 i branchbyte2. Moguci uvjeti <cond>: eq (==)
  , ne (!=), lt (<), le (<=), gt (>), ge (>=).
return //metoda mora imati povratni tip void. Ukoliko nema
   iznimaka odbacuje sve operande s parametarskog stoga.
```

Primjer Java koda i jednog mogućeg prijevoda za Java virtualni stroj

Što kada petlja koristi varijablu tipa double?

```
  void petlja() {
    double i;
    for (i = 0.0; i < 100.0; i++) {
        ;     // prazno tijelo petlje
    }
}//java kod

Method void petlja()//jedan moguci prijevod
0  dconst_0     // stavi double konstantu 0.0 na stog
1  dstore_1     // spremi u lokalne varijable 1 i 2 (double
    zauzima dvije varijable)
2  goto 9       // skoci na instrukciju na indeksu 9 (prvi
    puta ne radi inkrement)
5  dload_1     // stavi lokalnu varijablu 1 i 2 na stog
6  dconst_1      // stavi double konstantu 1.0 na stog
7  dadd  // zbroji operande na stogu
8  dstore_1 // spremi rezultat u lokalne varijable 1 i 2
```

Petlja s praznim tijelom koja iterira po varijabli tipa double

Što kada petlja koristi varijablu tipa double?

```
9    dload_1        // stavi lokalnu varijablu 1 i 2 na stog
10   ldc2_w #4       // ucitaj konstantu 100.0 sa skupa
    konstanti kod izvrsavanja i stavi na stog
13   dcmpg  // usporedi vrijednost spremljenu u varijabli i s
     konstantom 100.0
14   iflt 5 // skoci na instrukciju 5 ukoliko dcmpg vraca
    vrijednost <0 (dogada se ako i<100.0)
17   return         // prazni return

dconst_<d>  // stavi konstantu tipa double (0.0 ili 1.0) na
    stog operanada.
dstore_<n>  // <n> i <n+1> moraju biti indeksi u lokalnom
    polju varijabli trenutnog okvira. Vrijednost na vrhu
    stoga operanada mora biti tipa double. Ta vrijednost se
    skida sa stoga, vrsi se potrebna konverzija skupa
    vrijednosti i ta vrijednost se sprema u lokalne
    varijable <n> i <n+1>.
```

Petlja s praznim tijelom koja iterira po varijabli tipa double

Što kada petlja koristi varijablu tipa double?

```
dadd // oba operanda v1 i v2 moraju biti tipa double.
    Vrijednosti se skidaju sa stoga operanada i prolaze
    konverziju skupa vrijednosti. Rezultat v1+v2 se stavlja
    na stog operanada.
ldc2_w indexbyte1 indexbyte2 //stavlja long ili double sa
    skupa konstanti kod izvrsavanja na stog operanada (3
    byte-a)
dcmp<op> //<op> moze biti g - greater ili l - less. Obije
    opcije vracaju istu vrijednost  osim kada je jedan od
    operanada NaN. Obije instrukcije operiraju na dva
    operanda sa stoga operanada koji moraju biti tipa double
    . Vrijednosti se skidaju sa stoga operanada uz
    konverziju vrijednosti. Nakon toga dolazi do usporedbe
    floating-point vrijednosti:  a) u slucaju value1>value2
    vrijednost 1 se stavlja na stog, b) u slucaju value1==
    value2, vrijednost 0 se stavlja na stog, c) inace
    ukoliko niti jedna vrijednost nije NaN, se vrijednost -1
     stavlja na stog operanada,
```

Petlja s praznim tijelom koja iterira po varijabli tipa double

Što kada petlja koristi varijablu tipa double?

```
      //d) ukoliko je barem jedna vrijednost NaN, dcmpg
    stavlja vrijednost 1 na stog operanada dok dcmpl stavlja
     vrijednost -1.

if<cond> branchbyte1 branchbyte2 //<cond> je nesto od <eq> (
    value == 0), <ne> (value!=0), <lt> (value <0), <le> (
    value <=0), <gt> (value >0), <ge> (value >=0). Ukoliko
    je uvjet zadovoljen, branchbyte1 i branchbyte2 se
    koriste za racunanje pomaka od adrese operacijskog koda
    instrukcije if<cond>. Ciljna adresa mora biti
    operacijski kod instrukcije unutar metode koja sadrzi if
    <cond>. Ukoliko uvjet nije zadovoljen, izvrsavanje se
    nastavlja na adresi instrukcije koja slijedi if<cond>.
```

Petlja s praznim tijelom koja iterira po varijabli tipa double

Primjer pretvaranja *while* petlje

```
void whilePetljaInt() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}

Method void whilePetljaInt()
0   iconst_0
1   istore_1
2   goto 8
5   iinc 1 1
8   iload_1
9   bipush 100
11  if_icmplt 5
14  return
```

Petlja s praznim tijelom koja iterira po varijabli tipa double

Funkcija koja vraća zbroj dva realna broja

```
double doubleLocals(double d1, double d2) {
    return d1 + d2;
}//java kod

Method double doubleLocals(double,double) //moguci prijevod
0    dload_1  // prvi argument se ucita iz lokalnih varijable
     1 i 2 i stavlja na stog
1    dload_3  // drugi argument se ucita iz lokalnih
    varijabli 3 i 4 i stavlja na stog
2    dadd     // zbrajanje dva realna broja d1 i d2 koja se
    prvo skidaju sa stoga operanada te se rezultat (njihov
    zbroj) sprema na stog.
3    dreturn  // metoda mora imati povratni tip double.
    Povratna vrijednost mora biti tipa double. Ukoliko ne
    dode do iznimke, vrijednost se skida sa stoga operanada,
     dolazi do konverzije skupa vrijednosti, te se
    rezultantna vrijednost stavlja na stog operanada okvira
    pozivatelja metode. Sve druge vrijednosti se micu sa
    stoga operanada trenutne metode.
```

Funkcija koja vraća zbroj dva realna broja.

Glavne aritmetičke i logičke operacije

Operator	double	float	long	int
$+$	dadd	fadd	ladd	iadd
$-$	dsub	fsub	lsub	isub
$*$	dmul	fmul	lmul	imul
$/$	ddiv	fdiv	ldiv	idiv
%	drem	frem	lrem	irem
$\neg$	dneg	fneg	lneg	ineg
bit &	-	-	land	iand
bit $\vert$	-	-	lor	ior
bit $\oplus$	-	-	lxor	ixor

Pristupanje skupu konstanti kod izvršavanja

Mnoge **numeričke konstante**, **objekti**, **elementi članovi** i **metode** trenutne klase se dohvaćaju preko skupa konstanti kod izvršavanja.

Podaci tipa `int`, `long`, `float`, `double` te reference na instance klase `String` se učitavaju (stavljaju na stog operanada) korištenjem instrukcija `ldc`, `ldc_w` i `ldc2_w`.

`ldc` i `ldc_w` se koriste za pristup vrijednostima tipova `int`, `float`, `long` i `String`. `ldc_w` se koristi kada postoji veliki broj elemenata u skupu konstanti kod izvrsavanja i potreban je veći indeks (w - eng. *wide*).

Instrukcija `ldc2_w` se koristi za dohvaćanje svih vrijednosti tipova long i double (postoji samo "w" verzija).

Cjelobrojne konstante tipova `byte`, `char`, `short`, male cijelobrojne vrijednosti, mogu se učitavati koristeći `bipush` (*byte*), `sipush` (*short*) ili `iconst_<i>` (*int*) instrukcije. Neke manje floating-point konstante se mogu učitati koristeći instrukcije `fconst_<f>` i `dconst_<d>`.

Primjer pristupanja skupu konstanti kod izvršavanja

```
void inicijalizacije() {
    int i = 100;
    int j = 1000000;
    long l1 = 1;
    long l2 = 0xffffffff;
    double d = 2.2;
 } //java kod

 Method void inicijalizacije() //prevedeni kod
0   bipush 100  // mala int konstanta - bipush
2   istore_1
3   ldc #1      // velika int konstanta (1000000) - ldc
5   istore_2
6   lconst_1  // mala vrijednost tipa long - lconst_1 (brzo)
7   lstore_3
8   ldc2_w #6  // long 0xffffffff (int -1) - ldc2_w
11  lstore 5
13  ldc2_w #8  // double konstanta 2.200000 - ldc2_w
16  dstore 7
```

Inicijalizacije varijabli raznog tipa konstantom.

Dohvaćanje parametara i poziv metode

n argumenata poslanih **metodi instance klase** se po konvenciji sprema u **lokalne varijable** okvira kreiranog za **poziv nove metode**. Te varijable su numerirane $1, \ldots, n$ a **argumenti se primaju u poretku u kojem su predani metodi**. Po konvenciji, metodi instance klase se šalje **referenca na njenu instancu** (tzv. `this`) i sprema u lokalnu varijablu 0.

Za razliku od metoda instance klase, **statičke metode klase nemaju instancu**, stoga se **lokalne varijable** od indeksa 0 koriste za spremanje argumenata metode.

```
int zbroji(int i, int j) {
    return i + j;
} //java kod

Method int zbroji(int,int) //prevedeni kod
0 iload_1 //stavi na stog vrijednost lokalne varijable 1 (i)
1 iload_2 //stavi na stog vrijednost lokalne varijable 2 (j)
2 iadd    //zbroji i stavi rezultat na stog operanada
3 ireturn   //vrati rezultat tipa int.
```

Dohvaćanje parametara.

Dohvaćanje parametara i poziv metode

```
static int zbrojiStatic(int i, int j) {
    return i + j;
} //java kod

Method int zbrojiStatic(int,int) //prevedeni kod
0 iload_0 //stavi na stog vrijednost lokalne varijable 0 (i)
1 iload_1 //stavi na stog vrijednost lokalne varijable 1 (j)
2 iadd    //zbroji i stavi rezultat na stog operanada
3 ireturn   //vrati rezultat tipa int.
```

Dohvaćanje parametara static funkcije.

Poziv **normalne metode instance klase** se provodi pozivom instrukcije `invokevirtual` koja kao argument prima **indeks na skup konstanti kod izvršavanja**. Od tamo dohvaća ime metode i opis.

Dohvaćanje parametara i poziv metode

```
int zbroji12I13() {
    return zbroji(12, 13);
}

Method int zbroji12I13()
0   aload_0                 // stavi varijablu 0 na stog (this)
                            //aload<n> stavlja referencu na stog
1   bipush 12               // stavi int konstantu 12
3   bipush 13               // stavi int konstantu 13
5   invokevirtual #4        // pozovi metodu Klasa.zbroji(II)I
8   ireturn         // stavi sumu na stog pozivatelja
```

Pozivanje normalne metode.

Operand instrukcije `invokevirtual` (indeks #4) nije pomak do instrukcije u metodi instance klase. Pošto prevodiocu **nije poznata struktura instance klase**, on **generira simboličku referencu na metode instance**, koje su spremljene u skupu konstanti kod izvršavanja. **Stvarna lokacija** navedenih objekata se **utvrđuje tijekom izvođenja programa**. Isto se odnosi na **sve ostale instrukcije** *Java virtualnog stroja* koje **pristupaju instancama klase**.

Dohvaćanje parametara i poziv metode

```
int zbroji12I13() {
    return zbrojiStatic(12, 13);
}

Method int zbroji12I13()
0    bipush 12
2    bipush 13
4    invokestatic #3      // metoda Klasa.zbrojiStatic(II)I
7    ireturn
```

Pozivanje static metode.

Prevođenje poziva **statičke metode** (metode klase) je **slično prevođenju poziva metode instance** (normalne metode) osim što se `this` **ne prosljeđuje** od strane pozivatelja. Argumenti metode se primaju krečući od lokalne varijable 0. Instrukcija `invokestatic` se **uvijek** koristi za poziv statičkih metoda.

Pozivanje metoda superklase

Instrukcija `invokespecial` se koristi kod **poziva metode iz superklase** ili kod **poziva metode za inicijalizaciju instanci**.

```
class A {
    int it;
    int getItA() {
        return it;
    }
}
class B extends A {
    int getItB() {
        return super.getItA();
    }
} //java kod

Method int getItB()//prevedeni kod
0    aload_0   //stavi na stog lokalnu varijablu 0 (this)
1    invokespecial #4     // metoda A.getItA()I
4    ireturn //vrati vrijednost tipa integer, makni sve s
    programskog stoga trenutne metode
```

Pozivanje metode superklase.

Prevođenje instanci klasa

Instance klasa *Java virtualnog stroja* se stvaraju koristeći instrukciju `new`. **Konstruktor** (metoda za inicijalizaciju instance) kod *Java virtualnog stroja* se označava s `<init>`. Svaka klasa može imati **više metoda za inicijalizaciju instanci** koje odgovaraju više različitih konstruktora. **Prvo se kreira instanca klase** i **odgovarajuće varijable klase** i **superklasa**, zatim se poziva **metoda za inicijalizaciju instance**.

```
Object stvori() {
    return new Object();
}

Method java.lang.Object stvori()
0   new #1                  // klasa java.lang.Object
3   dup  //dupliciraj vrijednost na vrhu stoga i vrati na
    stog
4   invokespecial #4        // metoda java.lang.Object.<init>()V
7   areturn  //vrati referencu, skini sve vrijednosti sa
    stoga metode
```

Kreiranje instance klase

Prevođenje instanci klasa

```
new indexbyte1 indexbyte2 //indexbyte1 i indexbyte2 se
    koriste za stvaranje indeksa u skupu konstanti kod
    izvrsavanja trenutne klase. Na indeksu se mora nalaziti
    simbolicka referenca na klasu ili sucelje odgovarajuceg
    tipa. Alocira se memorija za novu instancu te klase (iz
    slobodnog dijela memorija) i inicijaliziraju se
    varijable clanice klase te instance. Referenca objekta
    se stavlja na parametarski stog.
dup // Duplicira vrijednost s vrha stoga operanada i stavlja
     je na vrh stoga operanada. Smije se koristiti kod svih
    tipova vrijednosti osim long i double.
invokespecial indexbyte1 indexbyte2 //indexbyte1 i
    indexbyte2 se koriste za stvaranje indeksa u skupu
    konstanti kod izvrsavanja trenutne klase. Na indeksu se
    mora nalaziti simbolicka referenca na metodu ili sucelje
     metode. Referenca dopusta pristup informacijama o imenu
     i opisu metode ili sucelja i simbolicku referencu na
    klasu ili sucelje u kojoj se moze pronaci metoda ili
    sucelje.
```

Kreiranje instance klase

Prevođenje polja

Polja *Java virtualnog stroja* su **također objekti**. Polja se stvaraju i obrađuju posebnim skupom instrukcija. Instrukcija `newarry` se koristi za **stvaranje polja numeričkog tipa**.

```
void stvoriPolje() {
    int polje[];
    int duljina = 100;
    int vrijednost = 12;
    polje = new int[duljina];
    polje[10] = vrijednost;
    vrijednost = polje[11];
}//java kod

Method void stvoriPolje()//jedan moguci prijevod
0   bipush 100 // stavi int konstantu 100 na stog (duljina)
2   istore_2   //skini 100 sa stoga i spremi u lokalnu
    variablu 2 (duljina)
```

Kreiranje instance polja

Prevođenje polja

```
3   bipush 12  // stavi int konstantu 12 (vrijednost) na
    stog
5   istore_3  // skini 12 sa stoga i spremi u lokalnu
    varijablu 3 (vrijednost)
6   iload_2   // stavi vrijednost varijable duljina na stog
7   newarray int   //skini "duljinu" sa stoga i stvori polje
     duljine "duljina"
9   astore_1          // povezi polje s referencom "polje"
10  aload_1           // stavi referencu polje na stog
11  bipush 10         // stavi int konstantu 10 na stog
13  iload_3           // stavi vrijednost na stog
14  iastore           //skini vrijednost i konstantu 10 sa
    stoga i spremi vrijednost u polje[10]
15  aload_1           // spremi referencu polje na stog
16  bipush 11         // stavi int konstantu 11 na stog
18  iaload            //skini int konstantu 11 i referencu
    polje sa stoga, stavi vrijednost polje[11] na stog
19  istore_3  // spremi vrijednost u varijablu "vrijednost"
20  return   //vrati i skini sve vrijednosti sa stoga metode
```

Kreiranje instance polja

Prevođenje višedimenzionalnog polja

```
int[][][] stvori3DPolje() {
    int mreza[][][];
    mreza = new int[10][5][];
    return mreza;
}

Method int stvori3DPolje()[][][]
0   bipush 10    // stavi konstantan int 10 na stog (prva
    dimenzija)
2   iconst_5     // stavi konstantan int 5 na stog (druga
    dimenzija)
3   multianewarray #1 dim #2 // klasa [[[I, trodimenzionalno
     polje int-ova. Stvara prve dvije dimenzije
7   astore_1     // povezi polje s referencom "mreza"
8   aload_1      // stavi referencu mreza na stog
9   areturn      // vrati referencu mreza, skini sve operande
     metode sa stoga
```

Kreiranje instance višedimenzionalnog polja

Prevođenje naredbe *switch*

Prevođenje *switch* naredbe koristi instrukcije `tableswitch` i `lookupswitch`. Instrukcija `tableswitch` se koristi kada se niz `case` slučaja može **učinkovito prikazati** u obliku **tablice pomaka na odgovarajuće instrukcije**. Ukoliko izraz ne zadovoljava niti jedan od ponuđenih uvjeta, koristi se `default` uvjet.

```
int izaberi(int i) {
    switch (i) {
        case 0:  return  0;
        case 1:  return  1;
        case 2:  return  2;
        default: return -1;
    }
}//java kod
```

Prevođenje switch naredbe.

Prevođenje naredbe *switch*

```
Method int izaberi(int)
0  iload_1  // Stavi lokalnu varijablu 1 (argument i) na
    stog
1  tableswitch 0 to 2: // Dozvoljeni indeksi su 0 do 2
     0: 28          // Ako je i=0, izvrsi instrukciju na 28
     1: 30          // Ako je i=1, izvrsi instrukciju na 30
     2: 32          // Ako je i=2, izvrsi instrukciju na 32
     default:34  // Inace, izvrsi instrukciju na 34
28 iconst_0      // i je bio 0; stavi int konstantu 0 na stog
29 ireturn       // vrati vrijednost i eliminiraj sve
    vrijednosti sa stoga trenutne metode
30 iconst_1     // i je bio 1; stavi int konstantu 1 na stog
31 ireturn      // kao 29
32 iconst_2     // i je bio 2; stavi int konstantu 2 na stog
33 ireturn      // kao 29
34 iconst_m1    // inace, stavi konstantu -1 na stog
35 ireturn      // kao 29
```

Prevođenje switch naredbe.

Prevođenje naredbe *switch*

Nije učinkovito koristiti `tableswitch` kada su slučajevi **rijetki** (sparse). Ukoliko se u **velikom numeričkom rasponu** javlja **samo nekoliko vrijednosti** (tablični prikaz bi imao puno praznih elemenata). U tom slučaju se koristi `lookupswitch` instrukcija koja **sparuje vrijednosti** `case` labela s pomacima na odgovarajuće instrukcije. Kod izvođenja se vrijednosti izraza **uspoređuju s ključevima mapiranja**. Ukoliko ključ postoji, izvodi se odgovarajuća instrukcija, inače se izvodi instrukcija s posebnim ključem `default`.

```
int izaberi1(int i) {
    switch (i) {
        case -100: return -1;
        case 0:    return  0;
        case 100:  return  1;
        default:   return -1;
    }
}
```

Primjer switch naredbe gdje nije učinkovito koristiti tableswitch.

Prevođenje naredbe *switch*

```
Method int izaberi1(int)
0   iload_1
1   lookupswitch 3:
          -100: 36
             0: 38
           100: 40
       default: 42
36  iconst_m1
37  ireturn
38  iconst_0
39  ireturn
40  iconst_1
41  ireturn
42  iconst_m1
43  ireturn
```

Primjer switch naredbe gdje nije učinkovito koristiti tableswitch.

Tablica `lookupswitch` instrukcije **mora biti sortirana po ključu** da bi se omogućila učinkovita implementacija pretraživanja ključeva (iako `tableswitch` ima učinkovitiji pristup elementu $O(1)$ u odnosu $O(log_2(n))$ kod `lookupswitch`-a).

Prevođenje iznimaka

Iznimke se izbacuju (pokreću) korištenjem ključne riječi `throw`.

```
void neNula(int i) throws TestIznimka {
    if (i == 0) {
        throw new TestIznimka();
    }
}

Method void neNula(int)
0   iload_1      // Stavi argument 1 (i) na stog
1   ifne 12      // If i==0, alociraj instancu i pozovi throw
4   new #1       // Kreiraj instancu TestIznimka
7   dup          // Dupliciraj referencu (za konstruktor)
8   invokespecial #7  // Pozovi metodu TestIznimka.<init>()V
11  athrow   // Druga referenca je izbacena (izbacivanje
    iznimke)
12  return   // Ne izvrsava se ukoliko je doslo do iznimke
```

Primjer izbacivanja iznimke.

Prevođenje iznimaka

```
athrow //uzima referencu objekta sa stoga. Referenca mora
    pokazivati na objekt koji je instanca klase Throwable
    ili neke njezine podklase.  Referenca tog objekta se
    zatim izbacuje na mjestu gdje se unutar trenutne metode
    prvi puta javlja objekt koji sadrzi lokaciju koda za
    obradu iznimke klase toga objekta. U tom trenutku se
    registar programskog brojila postavlja na lokaciju koda
    za obradu iznimke, stog operanada trenutnog okvira se
    cisti i referenca objekta klase Throwable se stavlja na
    stog operanada -  cime se nastavlja izvrsavanje. Ukoliko
     nema odgovarajuceg objekta koji sadrzi kod za obradu
    iznimke, trenutni okvir se eliminira. Ukoliko postoji
    pozivatelj metode, njegov okvir se aktivira i ponovo se
    izbacuje referenca objekta iznimke. Ukoliko ne postoji
    okvir pozivatelj, glavna dretva zavrsava izvodenje.
```

Primjer izbacivanja iznimke.

Prevođenje *try-catch* izraza

Ukoliko ne dođe od iznimke tijekom izvođenja `try` bloka, izvođenje se odvija kao da nema `try` bloka (**poziva se** `probaj()` i `tryCatch` **završava s izvođenjem**.

Funkcija `obradi()` se također prevodi kao **normalni poziv metode**. Međutim, prisutnost `catch` izraza uzrokuje **stvaranje tablice iznimki**. Tablica iznimki za metodu `tryCatch` sadrži jedan redak koji odgovara jednom argumentu (instanci klase `TestIznimka`) koju `catch` obrađuje.

```
void tryCatch() {
    try {
        probaj();
    } catch (TestIznimka e) {
        obradi(e);
    }
}
```

Primjer try-catch.

Prevođenje *try-catch* izraza

```
Method void tryCatch()
0    aload_0    // Pocetak try blocka, ucitavanje this
    reference
1    invokevirtual #6      // Poziv metode Primjer.probaj()V
4    return                // Kraj try bloka; normalni return
5    astore_1  // Spremi izbacenu vrijednost u lokalnu
    varijablu 1
6    aload_0               // Stavi this na stog
7    aload_1               // Stavi izbacenu vrijednost na stog
8    invokevirtual #5      // Pozovi metodu za obradu iznimke:
                           // Primjer.obradi(LTestIznimka;)V
11   return                // Vrati nakon obrade TestIznimke

Tablica iznimki:
Od       Do      Cilj          Tip
0        4        5            Klasa TestIznimke
```

Primjer try-catch.

Prevođenje *try-catch* izraza

Ukoliko se, tijekom izvršavanja instrukcija s indeksima 0 do 4 unutar `tryCatch`, izbaci neka vrijednost koja je instanca klase TestIznimke, izvođenje se nastavlja od indeksa 5 (implementira `catch`). Ukoliko izbačena vrijednost nije klase TestIznimke, navedeni `catch` je ne može obraditi i ona se prosljeđuje pozivatelju.

Višestruki `catch` se prevodi jednostavnim dodavanjem koda *Java virtualnog stroja* za svaki `catch` (jedan iza drugog) i dodavanjem redaka u tablicu iznimaka. Prvi dostupni `catch` obrađuje iznimku odgovarajućeg tipa. Ukoliko ne postoji `catch` odgovarajućeg tipa, iznimka se prosljeđuje pozivatelju.

Prevođenje *try-finally* izraza

Prevođenje bloka `try-finally` je slično prevođenju bloka `try-catch`. Glavna razlika je da se prije završetka izvođenja `try` bloka nužno mora izvršiti `finally` blok (neovisno o tome je li došlo do iznimke). Ukoliko je došlo do iznimke, prvo se obrađuje iznimka a zatim izvrši `finally` blok.

```
void tryFinally() {
    try {
        probaj();
    } finally {
        finaliziraj();
    }
}
```

Primjer try-finally.

Ukoliko `probaj` završi izvođenje bez iznimaka, počinje se izvršavati `finally` blok korištenjem `jsr` instrukcije. `jsr` instrukcija pozove kod `finally` bloka kao funkciju bez povratne vrijednosti (tako se i prevodi). Nakon završetka izvođenja `finally` bloka, nastavlja se izvođenje instrukcije nakon `jsr`.

Prevođenje *try-finally* izraza

```
Method void tryFinally()
0    aload_0                  // Ucitaj this referencu
1    invokevirtual #6  // Pozovi metodu Primjer.probaj()V
4    jsr 14 //Pozovi finally blok (kao void funkciju), spremi
     adresu sljedece naredbe (7 return) na stog
7    return                   // Kraj try bloka
8    astore_1  // Pocetak procedure za obradu proizvoljne
    iznimke (spremi iznimku u lokalnu varijablu 1)
9    jsr 14     // Pozovi finally blok (kao 4 - sprema 12)
12   aload_1  // Stavi referencu izbacene iznimke na stog
13   athrow // izbaci iznimku metodi pozivatelju i skini je
    sa stoga
14   astore_2  // Pocetak finally bloka, spremi adresu
    instrukcije za poziv nakon finally
15   aload_0                  // Stavi this na stog
16   invokevirtual #5 // Pozovi metodu Primjer.finaliziraj()V
19   ret 2 // Vrati iz finally, izvrsi 7 ili 12 (prema V2)
Exception table:
Od     Do       Cilj          Tip
0      4        8             proizvoljan
```

Primjer try-finally.