Moduli, anotacije, izrazi u programskom jeziku *Java*

Matej Mihelčić

Prirodoslovno-matematički fakultet, Sveučilište u Zagrebu

matmih@math.hr

02. studenoga, 2022.

Moduli

Modul je **najviša organizacijska cjelina** u programskom jeziku *Java*. Sadrži **jedan ili više paketa** (koji mogu imati proizvoljan broj podpaketa i jedinica za prevođenje).

Modul ima **ime, podatke o ovisnostima modula i paketima koje pruža drugim modulima**. Ti podaci se definiraju u dokumentu imena *module-info.java*. Ime modula slijedi konvencije označavanja paketa u *Javi*.

```
1 module ime {
2 }
```

Primjer deklaracije modula.

Postoje dvije vrste modula, **normalni** i **otvoreni** modul. Glavna razlika između njih je **razina dopuštenja pristupa** koje modul daje drugim modulima. Normalni modul **dopušta pristup, pri prevođenju i izvođenju, samo onim paketima koji su eksplicitno označeni kao slobodni za korištenje**. Otvoreni modul **dopušta pristup pri prevođenju samo onim paketima koji su eksplicitno označeni kao slobodni za korištenje, međutim pri izvođenju dopušta korištenje svih paketa svim modulima**.

Moduli

```
1 open module ime {
2 }
```

Primjer deklaracije otvorenog modula.

Neovisno o tipu modula, drugim modulima **su dostupni samo** *public* i *protected* tipovi i elementi članovi **definirani u onim paketima za koje modul eksplicitno daje dopuštenje pristupa**. Unutar modula je **dozvoljen pristup** *public* i *protected* tipovima i elementima **svih paketa**.

Normalni modul **dozvoljava reflektivni pristup** *public* i *protected* tipovima i elementima članovima **samo u paketima za koje modul eksplicitno daje dopuštenje pristupa** (međutim nisu otvoreni). Kod **otvorenih paketa normalnog modula** je **dopušten reflektivni pristup svim tipovima i elementima članovima** iz drugih modula i unutar modula. Otvoreni modul **dopušta reflektivni pristup svim paketima, svim tipovima i elementima članovima** od strane drugih modula i unutar modula.

Moduli

- **Ovisnosti modula o drugim modulima** se specificiraju oznakom `requires`. Ovisnosti modula o drugim modulima mogu imati dodatne oznake. `transitive` **svim modulima koji ovise o danom modulu implicitno deklarira ovisnost o modulu o kojem ovisi dani modul.** `static` **označava da je modul nužan pri prevođenju, međutim opcionalan pri izvršavanju. Nije dopušteno više puta specificirati ovisnosti o istom modulu.** Ukoliko **nije moguće identificirati neki od modula**, dolazi do **greške pri prevođenju**.
- **Paketi modula kojima se smije pristupati od strane drugih modula** se definiraju oznakom `exports`.
- Oznaka `opens` **otvara paket trenutnog modula**. Time se **dopušta pristup** *public* i *protected* tipovima i elementima članovima paketa **samo tijekom izvršavanja programa** (ne kod prevođenja). **Dopušta reflektivni pristup svim tipovima i elementima članovima paketa.** `exports` i `opens` mogu imati i `to` oznaku nakon koje slijedi **niz oznaka drugih modula** kojima se **daje pravo korištenja danog paketa**.

Moduli

- `uses` oznakom se označavaju sve ***usluge*** (skupovi klasa i sučelja koji nude neku funkcionalnost) koje su potrebne trenutnom modulu.
- `provides` oznakom se pruža niz usluga drugim modulima. Oznakom `with` se specificira klasa koja implementira navedenu uslugu.

```
1  module ime {
2  requires modul1;
3  requires static modul2;
4  requires transitive modul3;
5  exports prvi;
6  exports drugi to modul3, modul5;
7  opens treci to modul1;
8  uses java.sql.Driver;
9  provides SuceljeFunkcije with
10         KlasaImplementacija;
11 }
```

Razne mogućnosti pri deklaraciji modula.

- **Nije dozvoljeno** više puta pozvati `exports` na **istom paketu**. Navedeni paket **mora biti dostupan u jedinici za prevođenje zadanog modula**.

Moduli

- **Nije dozvoljeno** više puta pozvati `opens` na **istom paketu**. `opens` se **ne smije koristiti kod deklaracije otvorenog modula**.
- *Usluga* **mora biti klasa, sučelje ili tip anotacije** (**ne smije** biti *enum*). Usluga mora biti ili **deklarirana u trenutnom modulu** ili mora biti **dostupna danom modulu**. **Nije dozvoljeno više puta** koristiti `uses` na **istoj usluzi**.
- Svaki pružatelj usluga **mora biti** *public* **klasa ili sučelje**. Također mora biti ili `klasa najvišeg nivoa` ili **static ugniježdena klasa**. Pružatelji usluga **moraju biti deklarirani u danom modulu**. Metoda koja pruža uslugu **mora biti deklarirana u danom modulu ili mora biti dostupna danom modulu**. Također, **mora biti podtip usluge** definirane `provides` oznakom. Ukoliko usluga nema metodu koja pruža uslugu, tada **mora imati konstruktor koji je podtip usluge** definirane `provides` oznakom. `provides` i `with` oznake **ne smiju dva puta navoditi istu uslugu**.

Refleksija

Refleksija je **postupak pristupanja i pregledavanja informacija** o klasama, metodama, konstruktorima i elementima klase **za vrijeme izvođenja** *Java* programa. Postupak dopušta korištenje **reflektiranih elemenata, metoda i konstruktora** uz određena ograničenja.

```
1  import java.lang.reflect.Method;
2  import java.lang.reflect.Field;
3  import java.lang.reflect.Constructor;
4
5  class Test//klasa kojoj zelimo pristupiti
6  {
7   private int b;
8   public metoda(){  b = 5; }
9   public void metoda1(){ System.out.println("Broj: " + b);}
10  public void metoda2(int n){System.out.println("Broj: "+n);}
11  private void metoda3(){ System.out.println("Dostupna samo
       iz tijela klase");}
12 }
```

Primjeri refleksije.

Refleksija

Kod postupka refleksije **vrijede drugačija (slabija) ograničenja pristupa** od definiranih identifikatorima pristupa.

```
1  class Glavni
2  {
3   public static void main(String args[]) throws Exception
4    {
5    Test objekt = new Test();
6    Class klasa = objekt.getClass();
7    System.out.println("Ime klase: "+klasa.getName());
8    Constructor konstruktor = klasa.getConstructor();
9    System.out.println("Ime konst." +konstruktor.getName());
10   System.out.println("Javne metode : ");
11   Method[] metode = klasa.getMethods();
12   for (Method metoda:metode)
13     System.out.println(metoda.getName());
14   Method pozivMetode1 =klasa.getDeclaredMethod("metoda2",int
       .class);
15   pozivMetode1.invoke(objekt, 19);
16   Field element = klasa.getDeclaredField("b");
```

Primjeri refleksije.

Refleksija

```
1   element.setAccessible(true);//unatoc private
2   element.set(objekt,100);
3   Method pozivMetode2 = klasa.getDeclaredMethod("metoda1");
4   pozivMetode2.invoke(objekt);
5   Method pozivMetode3 = klasa.getDeclaredMethod("metoda3");
6   pozivMetode3.setAccessible(true);//unatoc private
7   pozivMetode3.invoke(objekt);
8     }
9 }
```

Primjeri refleksije.

- Korištenjem refleksije **prodiremo u strukturu paketa** te time **činimo svoj kod podložnim promjenama u implementaciji** (obično se metode dostupne za korištenje mijenjaju rijeđe od unutarnjih metoda koje implementiraju određene funkcionalnosti ili pomoćnih funkcija).
- Reflektivni poziv je **sporiji od direktnog pristupa** elementima klase ili poziva preko instance klase.
- Primjena kod korištenja klasa, metoda, elemenata klasa iz paketa za koje **nemamo izvorni kod** nego samo biblioteku.

Tip anotacije

Deklaracijom tipa anotacije definiramo **novi specijalni tip sučelja** koji se zove **tip anotacije**. Tip anotacije se označava ključnom riječju `interface` ispred kojeg dolazi znak `@`. Tip anotacije **ne smije imati isto ime** kao klase i sučelja koje ga sadrže.

Tijelo tipa anotacije **smije sadržavati samo deklaraciju metoda** koje **čine elemente tipa anotacije**. Povratni tip metoda tipa anotacije smije biti: a) **osnovni tip**, b) **string**, c) **klasa ili generička klasa**, d) **tip enum**, e) **tip anotacije**, f) **tip polja** s elementima koji su jednog od prethodnog tipa (**nisu dozvoljena višedimenzionalna polja**).

Metode **ne smiju imati prototip** jednak (ili prevodljiv) u prototip *public* i *protected* metoda definiranih u klasama *Object* i *java.lang.annotation.Annotation*. **Nisu dozvoljene cikličke deklaracije** (element član tipa anotacije `T` ne smije biti tipa `T`).

Tip anotacije bez elemenata se zove ***marker*** a s jednim elementom tip ***jednočlane*** anotacije. Po konvenciji, ime elementa tipova jednočlanih anotacija je ***value*** (vrijednost).

Primjeri tipova anotacija

```
1  @interface ZahtjevZaProsirenjem {
2       int    id();          // ID zahtjeva za prosirenje
3      String opis();  // Opis zahtjeva za prosirenje
4      String djelatnik();  // Ime djelatnika koji je razvio
       prosirenje
5      String datum(); // Datum kada je prosirenje realizirano
6  }
7
8  @interface Preliminarno{} //marker
9
10 @interface RazvojniTim{ //tip jednoclane anotacije
11     String[] vrijednost();
12 }
13
14 interface Format {}
15 @interface LijepIspis { //koristeni tip je tip klase koja
       kao gornju granicu ima neko sucelje Format
16     Class<? extends Format> vrijednost();
17 }
```

Primjeri deklaracije tipa anotacija.

Primjeri tipova anotacija

```
1  @interface Autor {//tip jednoclane anotacije autor
2      ImeIprezime value(); //koristi drugi, normalni tip
       anotacije "ImeIprezime"
3  }
4
5  @interface ImeIprezime { //normalni tip anotacije
6      String ime();
7      String prezime();
8  }
9
10 @interface Vrijeme { //koristenje tipa enum
11     enum Stanje { SUNCE, OBLACI, MAGLA, KISA, SNIJEG, TUCA,
       GRMLJAVINA }
12     Stanje value();
13 }
```

Primjeri deklaracije tipa anotacija.

Unutar deklaracije tipa anotacija, **moguće je definirati standardne** (eng. *default*) **vrijednosti** za neke elemente.

Primjeri tipova anotacija

```
1 @interface ZahtjevZaProsirenjem {
2       int     id();          // nema standardnu vrijednost,
      treba svaki puta specificirati.
3       String opis();  // nema standardnu vrijednost, treba
      svaki puta specificirati.
4       String djelatnik() default "nije dodijeljen"; // ukoliko
       nije specificirana vrijednost, vrijednost se postavlja
      na standardnu "nije dodijeljen".
5       String datum() default "nije dodijeljen"; // ukoliko
      nije specificirana vrijednost, vrijednost se postavlja
      na standardnu "nije dodijeljen".
6 }
```

Primjeri deklaracije tipa anotacije sa standardnim vrijednostima elemenata.

Anotacije se mogu i **ponavljati** ukoliko su njihove anotacije tipa **deklarirane** na **odgovarajući**, **specifičan** način.

Za tip anotacije koji je ponavljajući, **treba specificirati tip anotacije** čija vrijednost **sadrži niz anotacija tipa ponavljajuće anotacije**. **Pri deklaraciji treba koristiti anotaciju anotacije** `@Repeatable`.

Primjeri tipova anotacija

```
1  import java.lang.annotation.Repeatable;
2
3  // Boja: tip ponavljajuce anotacije koji koristi tip Boje
       kao spremnik
4  @Repeatable(Boje.class)
5  @interface Boja{ String vrijednost(); }
6
7  // Boje: Sadrzi niz elemenata tipa Boja
8  //Takoder ponavljajuci tip
9  @Repeatable(NizBoja.class)
10 @interface Boje { Boja[] vrijednost(); }
11
12 // NizBoja: Sadrzi niz elemenata tipa Boje
13 @interface NizBoja { Boje[] vrijednost(); }
```

Primjeri deklaracije ponavljajućeg tipa anotacije.

Ukoliko je tip anotacije koji sadrži ponavljajući tip anotacije i sam ponavljajući, tada vrijede pravila **mješovite anotacije** (moguće je ponavljati više puta ili sadržani tip ili tip koji sadrži ponavljajući tip, ali ne oba).

Primjeri tipova anotacija

```
1  import java.lang.annotation.Target;
2  import java.lang.annotation.ElementType;
3  import java.lang.annotation.Repeatable;
4
5  @Target(ElementType.TYPE) //Boja se moze javljati kod
       deklaracije bilo kojeg tipa
6  @Repeatable(Boje.class)
7  @interface Boja {String vrijednost();}
8
9  @Target(ElementType.ANNOTATION_TYPE)//Boje se moze javljati
       kod deklaracije bilo kojeg anotacijskog tipa
10 @interface Boje {
11     Boja[] vrijednost();
12 }
```

Ograničavanje lokacija ponavljanja anotacija.

Pošto se tip annotacije *Boje* može aplicirati na **podskup deklaracija tipa** u odnosu na tip *Boja*, tip *Boja* se **može ponavljati samo tamo gdje je se može koristiti tip** *Boje*.

Predefinirani tipovi anotacija

- `@Target` - definira kada je tip anotacije primjenjiv.
- `@Retention` - određuje javlja li se anotacija samo u izvornoj datoteci, u binarnoj reprezentaciji i može li se koristiti za vrijeme izvršavanja programa (`java.lang.annotation.RetentionPolicy.{SOURCE, CLASS, RUNTIME}`.
- `@Inherited` - označava da podklasa nasljeđuje anotaciju nadklase.
- `@Override` - označava da se metoda nadjačava.
- `@SuppressWarnings` - javlja prevodiocu da ne izbacuje poruke upozorenja za anotirani element. Moguća upozorenja: `Unchecked warnings`, `Deprecation warnings`, `Removal warnings`, `Preview warnings`.
- `@Deprecated` - označava da je klasa ili metoda zastario dio koda i da se nebi trebao koristiti.
- `@SafeVarargs` - javlja prevodiocu da su tipovi parametara metode sigurni za korištenje, nakon čega prevodioc ne javlja `Unchecked warnings`.
- `@Repeatable` - označava da se tip anotacije može ponavljati.
- `@FunctionalInterface` - označava da je sučelje **funkcijsko sučelje**.

Anotacije

Anotacija je oznaka koja **pruža informacije o nekom dijelu programa**, ali **nema utjecaja pri izvršavanju**. Anotacija je **poziv tipa anotacije** i **pruža vrijednosti elemenata tipa**.

Postoje tri tipa anotacija: a) **normalne anotacije**, b) **markeri**, c) **jednočlane anotacije**.

Normalne anotacije: **specificiraju ime tipa anotacije** i **listu parova element-vrijednost** razmaknutih zarezima. **Obavezno** definira vrijednosti **svih elemenata** odgovarajućeg tipa anotacije osim onih koje imaju **predefiniranu** (eng. *default*) **vrijednost** (vrijednosti tih elemenata ne moraju biti nužno navedene).

```
1 @ZahtjevZaProsirenjem(
2     id          = 1035724,
3     opis = "Dodaj novi gumb na sucelje",
4     djelatnik = "Marko M.",
5     datum       = "30/3/2021"
6 )
7 public static void dodajGumb(JFrame f) { ... }
```

Primjer anotacije.

Anotacije

Marker anotacija se koristi kao **poziv tipa markera anotacije**. **Skraćeni oblik normalne anotacije.**

```
1 @Preliminarno public class Adapter { ... }
```

Primjer marker anotacije.

Jednočlane anotacije: Označavaju **poziv tipa jednočlane anotacije**. Također su **pokrata za normalnu anotaciju**.

```
1 @RazvojniTim({"Marko", "Pero", "Ivan"})//jednoclana
    anotacija ima element koji je polje stringova.
2 public class KompliciranaKlasa{...}
3
4 @RazvojniTim("Matej Mihelcic")//ne trebaju {} ako element
    ima samo jedan clan
5 public class KolegijJava { ... }
```

Primjer jednočlane anotacije.

Anotacije

```
1  @Author(@ImeIPrezime(ime = "Marko", prezime = "Maric"))//
       jednoclana anotacija kao parametar prima normalnu
       anotaciju
2  public class  Algoritam{ ... }
3
4  @Vrijeme(Vrijeme.Stanje.Sunce)//jednoclana anotacija koja
       sadrzi element tipa enum
5  public class StanjeNaCesti { ... }
6
7  class NeobicniFormat implements Format { ... }
8
9  @LijepIspis(NeobicniFormat.class)//OK tip anotacije prima
       klasu koja implementira Format
10 public class Kaos { ... }
11
12 @LijepIspis(String.class)//NIJE OK, string ne implementira
       Format
13 public class Tekst { ... }
```

Primjer jednočlane anotacije.

Gdje smijemo koristiti anotacije?

Anotacija deklaracije se koristi **kod deklaracija** ukoliko je tip anotacije primjenjiv u deklaracijskom kontekstu (definiramo pomoću anotacije anotacija `@Target`). Koristi se i kod **klasa**, **sučelja**, **enuma**, **tipa anotacija** ili **deklaracija parametara tipa**.

Anotacija tipa je anotacija koja se **primjenjuje na tip** (ili njegov bilo koji dio) i primjenjiva je u tom kontekstu.

```
1 @Nesto int f; //moze biti anotacija deklaracije ili
      anotacija tipa ovisno o meta-anotaciji @Target(
      ElementType.FIELD) ili @Target(ElementType.TYPE_USE).
2
3 @C int @A [] @B [] f;//anotacije tipa ukoliko su A, B, i C
      meta-anotirane s @Target(ElementType.TYPE_USE)
4 //@A se odnosi na polje tipa int[][], @B na komponentu tipa
      int[], @C na element tipa int.
```

Anotacija deklaracije i anotacija tipa.

Gdje smijemo koristiti anotacije?

Kod deklaracija koje se razlikuju samo u broju dimenzija polja, **anotacija s lijeve strane tipa se uvijek odnosi na isti tip**.

```
1 @C int f; //@C se odnosi na tip int u sve tri deklaracije.
2 @C int[] f;
3 @C int[][] f;
```

Anotacija tipa.

Uobičajeno je da se deklaracije anotacija pišu **prije drugih modifikatora**, a anotacije tipa **neposredno prije tipa na koji se odnose**. Anotacija se može javiti kod deklaracija konstruktora, elemenata članova klase, deklaracija formalnih argumenata i parametara iznimki, lokalnih varijabli (uključujući varijable `for` petlje i posebne konstrukcije `try` bloka koji koristi resurse). U tim slučajevima se anotacije uvijek smatraju **anotacijama deklaracije**. Unatoč tome, **one mogu djelovati** i na **tip u posebnim slučajevima**.

Gdje smijemo koristiti anotacije?

- Ukoliko je tip anotacije primjenjiv u kontekstu deklaracije a ne u kontekstu tipa, tada se anotacija primjenjuje **samo na deklaraciju**.
- Ukoliko je tip anotacije primjenjiv u kontekstu tipa a ne u kontekstu deklaracije, tada se anotacija primjenjuje **samo na tip najbliži anotaciji**.
- Ukoliko se tip anotacije može primjeniti i u kontekstu deklaracije i u kontekstu tipa, tada se anotacija **primjenjuje i na deklaraciju i na tip** koji je najbliži anotaciji.

Kod deklaracija metoda povratnog tipa `void` i lokalnih varijabli deklariranih s `var` **nema najbližeg tipa** (dolazi do greške pri prevođenju ukoliko se anotacija može primjeniti samo na najbliži tip). Kod anotacija koje se primjenjuju na konstruktor, kao tip se uzima **tip novokreiranog objekta**. U svim ostalim slučajevima, kao tip se uzima **tip upisan u izvornom kodu za deklarirani element**.

```
1 @Nesto public static String f;//najblizi tip je String
2 @Nesto <T> int[] m() {...} //najblizi tip je int
```

Primjena anotacija.

Višestruke anotacije istog tipa

Višestruke anotacije istog tipa se smiju javljati **samo ako** je anotacija `@Repeatable`.

```
1  @Boja("Crvena") @Boja("Plava")// OK
2  public class Hlace{...}
3
4  @Boja("Crvena") @Boje(@Boja{"Plava"})//isto OK
5  public class Hlace{...}
6
7  @Boja("Crvena") @Boja("Zelena") @Boje(@Boja{"Plava"})//nije
       dozvoljeno, ne smijemo imati anotaciju koja se ponavlja
       i anotaciju koja ju sadrzi
8  public class Hlace{...}
9
10 @Boja("Crvena") @Boje({@Boja("Zelena")}) @Boje({@Boja("Plava
       ")})//nije dozvoljeno, ne smijemo imati ponovljenu
       anotaciju koja sadrzi anotaciju ukoliko se i ta
       anotacija javlja (cak i ako je anotacija koja se
       ponavlja @Repeatable).
11 public class Hlace{...}
```

Višestruke anotacije istog tipa.

Specifičnosti naredbi u programskom jeziku *Java*

Naredbe **nemaju vrijednost** i **kontroliraju način izvršavanja programa**. Neke naredbe **sadrže druge naredbe** unutar svoje strukture dok neke naredbe **sadrže izraze**. Programski jezik *Java* dijeli većinu naredbi s *C*-om. Međutim, za neke *C*-ovske naredbe kao `goto` **nema ekvivalent**, stoga **određene funkcionalnosti** nadoknađuje drugačijim konstrukcijama. *Java* sadrži neke naredbe i konstrukcije koje su **specifične** baš za *Javu*.

Naredbe se mogu izvršiti **normalno** ili **neočekivano**. Naredba se može izvršiti neočekivano zbog izvršavanja naredbe: a) `break` bez labele, b) `break` s labelom, c) `continue` bez labele, d) `continue` s labelom, e) `return` bez vrijednosti, f) `return` sa zadanom vrijednosti, g) `throw` sa zadanom vrijednosti, uključujući iznimke koje izbaci *Java virtualni stroj*, h) `yield` sa zadanom vrijednosti.

Ukoliko dođe do **neočekivanog izvršavanja izraza**, tada se i **odgovarajuća naredba neočekivano izvrši**. Svi koraci izvršavanja koji bi se proveli u normalnom načinu izvršavanja se **ne provode**.

Specifičnosti naredbi u programskom jeziku *Java*

Analogno, **neočekivano izvršavanje podnaredbe** uzrokuje **neočekivani završetak naredbe koja ju sadrži** i **svih koraka** koji bi se izvršili pri normalnom izvođenju tih naredbi. Ukoliko **nije došlo do neočekivanog izvršavanja niti jednog izraza ili podnaredbe**, tada kažemo da je **naredba izvršena normalno**.

Deklaracija lokalne klase:
Lokalna klasa je **ugniježđena klasa** koja **ima ime** i **nije član niti jedne druge klase** (smještena je unutar bloka). Lokalne klase **ne smiju sadržavati** identifikatore pristupa *public*, *protected* ili *private* ili identifikator *static*.

```
1 class A {
2     class B {} //ugnijezdena, unutarnja klasa
3     void f1() { //funkcija clanica klase A
4         new B(); // stvaramo instancu klase A.B
5         .
6         .
```

Deklaracija lokalne klase.

```
1   class B extends B {} // kruzna definicija, nije dozvoljeno! B vidi cijelu deklaraciju svoje klase, stoga nasljeduje samu sebe.
2   {
3       class L{} //lokalna klasa, nije clanica niti jedne druge klase
4       {
5           class L {} //nije dozvoljeno unutar metode
6       }
7   class L {} // nije dozvoljeno unutar metode
8       class  G{
9           void bar() {
10              class L {} // ok, jer je lokalna unutar klase vece dubine
11          }
12      }
13  }
14      class L {} // ok, nije unutar dosega prijasnje deklaracije L
15  }
16 }
```

Deklaracija lokalne klase.

Deklaracija lokalnih varijabli korištenjem ključne riječi *var*

```
1  var a = 1;                    // OK, a je tipa int
2  var b = 2, c = 3.0;           // Visestruke deklaracije tipa var
       nisu dozvoljene
3  var d[] = new int[4]; // Dodatne [], nije OK
4  var d = new int[4]; //OK
5  var e;                        // Fali inicijalizacija!
6  var f = { 6 };                // Inicijalizacija za polje, nije OK!
7  var g = (g = 7);              // Nije OK, referenca same varijable
       unutar izraza.
8  var b = java.util.List.of(1, 2);  // OK, b ima tip 'List<
       Integer>'
9  var c = "x".getClass();       // c ima tip 'Class<? extends
       String>'
10 var d = new Object() {};  // d ima tip java.lang.Object
11 var e = (CharSequence & Comparable<String>) "x";
12 // e ima tip CharSequence & Comparable<String>
13 var f = () -> "BOK!";         // Nije dozvoljeno pridruziti
       lambda izraz
14 var g = null;              // Nije dozvoljeno pridruziti null tip
```

Primjeri dozvoljenih i nedozvoljenih deklaracija koristeći var.

Specifičnosti naredbi u programskom jeziku *Java*

Java sadrži praznu naredbu koja se **uvijek izvrši normalno**.

```
1 ;
```

Prazna naredba.

Naredbe mogu imati ime (uglavnom se koristi u kombinaciji s `break` i `continue`).

```
1  public class Iteriranje {
2       double[][] elementi;
3       double sum = 0.0;
4
5  public void sumaParnih(){
6   i:
7       for (int i = 0; i < elementi.length; i++) {
8         for(int j=0;j<elementi.length;j++){
9            if(i%2==0) sum+=elementi[i][j];
10           else continue i;//nova iteracija vanjske petlje
11         }//racuna sumu elemenata
12      }//u svim redcima matrice s parnim indeksom
13   }
```

Imenovane naredbe.

Naredba koja sadrži izraz

Dozvoljeno je korištenje **samo nekih** izraza u tom kontekstu: a) **pridruživanja**, b) **prefiks/postfiks inkrement/dekrement**, c) **poziv metode**, d) **izraz za kreiranje instance klase**.

```
1  System.out.println("Hello world");  // OK, naredba koja
       sadrzi poziv metode
2  (System.out.println("Hello world"));  // nije dozvoljeno
3  (void)... ;  // nije dozvoljeno
4  x++;          //OK!
5  (++x);        // nije dozvoljeno
6  int y = x;    //OK!
7  int y = (x); //OK!
8  int y = (++x); //OK!
9  int y = (x++); //OK!
10 (int y) = x; //nije dozvoljeno
11 int (y) = x; //nije dozvoljeno
12 int (y = x); //nije dozvoljeno
13 int y;
14 (y =x);       //nije dozvoljeno
```

Naredbe koje sadrže izraze.

Naredba assert

Naredba `assert` **sadrži logički izraz**. Ona može biti **omogućena** ili **onemogućena**. Ukoliko je omogućena, **evaluira se logički izraz** te se **javlja greška** ukoliko je **izraz lažan**. Ukoliko je `assert` onemogućen, tada se **naredba ignorira**.

```
1 int x = 5;
2 int y = 4;
3 assert y<4;//normalni izgled assert naredbe
4 assert x%2==0: "x nije paran!"; //assert s dodatnim izrazom
```

Naredba assert.

Naredba `assert` uz logički izraz može opcionalno imati **dodatni izraz** koji **služi za davanje dodatnih informacija ukoliko dođe do pogreške**. U primjeru će se string *x nije paran!* **ispisati zajedno s drugim informacijama o pogrešci** ukoliko je naredba `assert` **omogućena**.

Ukoliko je logički izraz `assert` naredbe **lažan**, **pokreće se izbacivanje pogreške** od strane *Java virtualnog stroja*. Greška se u principu **može obraditi** iako je **zamišljeno da se to nikada ne radi**.

Naredba `throw`

Naredba `throw` **pokreće izbacivanje iznimke**. Nakon izbacivanja iznimke dolazi do **automatskog prijenosa kontrole izvršavanja**, što može prouzročiti **izlaskom** iz više naredbi i izvršavanja konstruktora, inicijalizacija instanci, statičkih inicijalizacija i inicijalizacija elemenata klase sve **dok se ne naiđe na naredbu** `try` koja obrađuje iznimku odgovarajuće (izbačene) klase. Ukoliko ne postoji takav `try` **dolazi do prekida izvršavanja dretve** koja je izvršila naredbu `throw` nakon izvođenja metode `uncaughtException`.

Izraz unutar `throw` naredbe **mora biti varijabla**, **vrijednost referenciranog tipa koja se može pridružiti tipu** `Throwable` ili biti **null referenca**. U suprotnom dolazi do **pogreške pri prevođenju**. Referencirani tip **mora biti ne generički tip klase**.

```
1 if(x==0){
2  throw new java.lang.ArithmeticException("x je nula!"); /*
     izbaci java.lang.ArithmeticException uz poruku "x je
     nula!" ako x == 0 */}
```

Naredba throw.

Naredba throw

Naredba `throw` **prvo evaluira izraz** i zatim **završi izvođenje neočekivano** (bez obzira na vrijednost izraza).

- Ukoliko je naredba `throw` **sadržana** unutar `try` naredbi dolazi do izvršavanja `finally` blokova (ukoliko postoje). Ukoliko dođe od **iznimnog završetka izvršavanja** `finally` bloka, **može doći do poremećaja u prijenosu kontrole izvršavanja pokrenute od strane** `throw`.
- Ukoliko se `throw` **nalazi** unutar metoda, lambda izraza i konstruktora i ne dolazi do obrade iznimke, tada izvršavanje tih metoda, lambda izraza i konstruktora **završava neočekivano**.
- Ukoliko se `throw` javlja u kodu statičke inicijalizacije tada ili dolazi do `unchecked exception` ili dolazi do obrade iznimke. Kod izvršavanja, **ukoliko nije došlo do obrade iznimke**, **ponovo se izbacuje (prijavljuje) iznimka** ukoliko je instanca klase `Error` ili njezine podklase, **inače se omota u objekt klase** `ExceptionInInitializerError` i **prijavljuje se ta iznimka**.

Naredbe `throw` i `synchronized`

- Ukoliko se naredba `throw` javlja u kodu inicijalizacije instance, tada je povratna vrijednost `throw` naredbe ili `unchecked exception` ili **dolazi do obrade iznimke** ili se **tip prijavljene iznimke javlja u svakom konstruktoru klase**.

Tipovi koji prijavljuju iznimku definiranu od strane korisnika bi **u pravilu trebali biti podklase** klase `Exception` koja je **podklasa klase** `Throwable`.

Naredba `synchronized` se koristi kod **višedretvenih programa** da bi **dretva dohvatila lokot koji osigurava isključiv pristup dijeljenim resursima**. Nakon korištenja dijeljenog resursa, **naredba otpušta lokot**. Naredba `synchronized` **uvijek djeluje na referencirani tip** (u suprotnom dolazi do greške pri prevođenju). Jedna dretva može dohvatiti jedan te isti lokot **više puta**.

Naredba `synchronized` **prvo pokuša izvrijedniti izraz** na koji je primijenjena. Ukoliko izvrijednjavanje izraza **završi neočekivano**, **cijela naredba završi izvođenje neočekivano**.

Naredba synchronized

Ukoliko je **rezultat izvrednjavanja** `null`, pokreće se `NullPointerException`. U suprotnom **kreće izvršavanje bloka koji sadrži zaštićene resurse**. Ukoliko izvođenje bloka **završi normalno**, **otključava se lokot i naredba završava normalno**. Ukoliko izvršavanje bloka **završi neočekivano**, tada se **lokot otključava** a naredba `synchronized` **završava neočekivano**.

```
1  class Test {
2      public static void main(String[] args) {
3          Test t = new Test();
4          synchronized(t) {
5              synchronized(t) {//moguce je da ista dretva
       dohvati lokot za isti resurs vise puta, inace bi ovdje
       doslo do potpunog zastoja (eng. deadlock)
6                  System.out.println("dohvacen lokot!");
7              }
8          }
9      }
10 }
```

Naredba synchronized.

Naredba `try` s resursima

`try` s resursima je **parametriziran lokalnim varijablama** (koje se zovu resursi). Oni se **inicijaliziraju prije izvršavanja** `try` bloka i **automatski zatvaraju nakon izvršavanja bloka u obrnutom poretku u odnosu na poredak inicijalizacije**. `catch` i `finally` su **često nepotrebni** kada se **resursi automatski zatvaraju**. Resursi često **sadrže veze s datotekama** ili **mrežne resurse** (npr. utičnice) koje se **automatski zatvaraju** kod `try` naredbe s resursima. Automatsko zatvaranje resursa se može postići tako da klasa resursa **implementira sučelje** `java.lang.AutoCloseable`.

```
1 try (BufferedReader br = new BufferedReader(new FileReader(
    putanjaDoDatoteke))) {
2       return br.readLine();
3    }
```

Try s resursima.

Resursi se inicijaliziraju u poretku **s lijeva na desno**. Ukoliko dođe do iznimke pri inicijalizaciji **bilo kojeg resursa**, **svi resursi se zatvaraju**.

Naredba try s resursima

```
1 try(
2  java.util.zip.ZipFile zf = new java.util.zip.ZipFile(
     imeZipDokumenta);
3 java.io.BufferedWriter writer = java.nio.file.Files.
     newBufferedWriter(putanjaIzlazneDatoteke, charset))
4 { ... }//charset obicno java.nio.charset.StandardCharsets.
     UTF_8
5
```

Try s više resursa.

try s resursima kod kojeg je definirano više resursa se **prikazuje kao više ugniježdenih** try-catch-finally naredbi.

Naredba `yield`

Naredba `yield` **postavlja određenu vrijednost kao povratnu vrijednost** `switch` naredbe (prekida izvođenje daljnjih switch blokova). Po namjeni **proširuje mogućnosti korištenja** `break` unutar `switch` uvjeta.

Ukoliko `yield` **nema specificiranu ciljnu varijablu** (varijablu kojoj je pridružena vrijednost vraćena od `yield`), dolazi do **greške pri prevođenju**. Ciljna varijabla od `yield` **ne smije sadržavati** metode, konstruktore, inicijalizatore instanci, statičke inicijalizatore ili lambda izraze. Izraz na koji se primijenjuje `yield` **ne smije biti** `void`.

Naredba `yield` **prvo izvrijedni izraz**. Ukoliko izvrednjavanje **završi neočekivano**, naredba `yield` **završi neočekivano**. Ukoliko izvrednjavanje izraza **završi normalno**, tada `yield` **završi neočekivano uz odgovarajuću povratnu vrijednost**.

Naredba yield

```
1  int i = 4;
2  int j = switch(i){
3            case 1:
4              System.out.println("Mala vrijednost"); yield 0;
5            case 2:
6              System.out.println("Srednja vrijednost"); yield 2;
7            case 3:
8              System.out.println("Veca vrijednost"); yield 3;
9            case 4:
10             System.out.println("Velika vrijednost"); yield 5;
11           default:
12             System.out.println("Jako velika vrijednost");
13                   yield 20;
14           };
15 System.out.println("Povratna vrijednost switch: "+j);
16
```

Naredba yield.

Program ispiše: *Povratna vrijednost switch: 5*

Nedohvatljive naredbe

Postojanje naredbe koja **nije dohvatljiva uzrokuje grešku pri prevođenju**. Dohvatljivost naredbi je strogo definirana nizom pravila koja **donekle** slijede intuiciju (vidi odjeljak 14.22)[1].

```
1  while (false) { x=3; } //greska, naredba x=3 je
       nedohvatljiva (uvjet while petlje je false konstanta)
2  boolean u = false;
3  while(u){x = 3;} //formalno dohvatljivo, u nije konstanta. x
       =3 se nece izvrsiti bez izmjene u.
4  if (false) { x=3; }//OK, iako sadrzi konstantu
5  //ideja je da se moze definirati
6  static final boolean DEBUG = false;
7  if (DEBUG) { x=3; }//moze se koristiti za debugiranje
8  int x=0;
9  for(;false;)
10    x++; //nedohvatljiva naredba (false konstanta u uvjetu)
11 boolean flag = false;
12 for(;flag;) x++; //OK!
```

Dohvatljive i nedohvatljive naredbe.

[1] https://docs.oracle.com/javase/specs/jls/se17/html/jls-14.html#jls-14.22

Nedohvatljive naredbe

```
1  int zbroji(int x, int y){
2    return x+y;
3    System.out.println(x+y);//nedohvatljiva naredba
4  }
5
6   int xu=2;
7         while(true){
8             if(xu==2){
9                 break;
10            System.out.println(x);//nedohvatljiva naredba
11            }
12        }
13
14        while(true){
15            if(xu==2)
16                break;
17            System.out.println(x);//OK, moze se dohvatiti
      ukoliko promijenimo vrijednost od xu.
18        }
```

Dohvatljive i nedohvatljive naredbe.

Nedohvatljive naredbe

```
1  try {
2     throw new Exception("Neka iznimka");
3       System.out.println("Pozdrav!");//nedohvatljiva naredba
4        }
5     catch (Exception e) {
6           e.printStackTrace();
7       }
8
9  while(x==0 && x!=0){
10              x++; //OK, iako se ne moze dohvatiti
11                  //razlog, uvjet nije konstanta false
12          }
13
14 for (int i = 0; i < 5; i++)
15     {
16       continue;
17       System.out.println("Bok!"); //nedohvatljiva naredba
18      }
```

Dohvatljive i nedohvatljive naredbe.

Izrazi u programskom jeziku *Java*

Izrazi se koriste za: **dobivanje rezultata koji se spremaju u varijable**, **dobivanje vrijednosti koje se koriste kao argumenti** ili **operandi u većim izrazima**, te **za promjenu toka izvršavanja naredbi**.

Rezultat izvrednjavanja izraza može biti: a) **varijabla**, b) **vrijednost**, c) **ništa** (`void`). Ukoliko izraz **označava varijablu**, a **potrebna je vrijednost u daljnjim izvrednjavanjima izraza**, tada se **koristi vrijednost te varijable**. Zbog toga **možemo u oba slučaja govoriti o vrijednostima izraza**.
Konverzija skupa vrijednosti se primijenjuje na rezultat izvrednjavanja **svakog izraza koji stvara vrijednost**.

Izraz nakon izvrednjavanja **ne proizvodi vrijednost** ako i samo ako je izraz **poziv metode koja ne vraća vrijednost** (povratna vrijednost je `void`). Takav izraz se može koristiti **samo kao naredba ili kao jedan izraz tijela lambda izraza**. U svim drugim kontekstima se **očekuje vrijednost**. Naredba koja odgovara izvrednjavanju izraza i tijelo lambda izraza **mogu sadržavati** i poziv funkcija koje **vraćaju rezultat** (on se **odbacuje**).

Izrazi u programskom jeziku *Java*

Izrazi se javljaju kod: deklaracije nekog tipa (klase ili sučelja), kod inicijalizacije elemenata klase ili sučelja, kod deklaracije konstruktora, poziva metoda ili anotacija, kod anotacija deklaracije paketa ili deklaracija tipa najvišeg nivoa.

Prema svom **obliku**, izrazi mogu biti:
izrazi imena (deklaracije varijabli, elemenata klase, puna imena paketa, klasa ili elemenata članova), **osnovni izrazi** (literali, kreiranje objekata, pristup elementima članovima klasa ili sučelja, poziv metoda, reference metoda, pristup elementima polja, izrazi u zagradama), **izrazi unarnih operatora**, **izrazi binarnih operatora**, **izrazi ternarnih operatora**, **lambda izrazi**, `switch` **izrazi**.

Prema kontekstu, izrazi mogu biti:
konstantni izrazi (izrazi čija vrijednost se može utvrditi prilikom prevođenja), **samostojeći izrazi** (svi izrazi koji nisu konstantni niti izrazi ovisni o kontekstu), **izrazi ovisni o kontekstu** (tip vrijednosti nastalih izvrednjavanjem ovih izraza ovisi i može se mijenjati ovisno o kontekstu).

Izrazi u programskom jeziku *Java*

Sljedeći oblici izraza mogu biti ovisni o kontekstu:
izrazi u zagradama, **izrazi kreiranja instance klase**, **izrazi poziva metode**, **izrazi reference metode**, **uvjetni izrazi**, **lambda izrazi**, `switch` **izrazi**.

```
1  ArrayList<Integer> a = (new ArrayList<>()); //izraz u
       zagradama ovisan o kontekstu (tipu referenciranog tipa).
2  ArrayList<Integer> a = new ArrayList<>();//izraz kreiranja
       instance klase.
3
4    public static <T extends Integer,S extends Double> S
       zbroji(T prvi, S drugi){
5       double r = prvi.doubleValue() + drugi.doubleValue();
6             Double rezultat = r;
7             return (S)rezultat; }
8
9  Integer broj1 = 250;
10 Double broj2 = 314.45, rezultat;
11 rezultat = Izrazi.zbroji(broj1,broj2);// izraz poziva metode
```

Primjeri izraza ovisnih o kontekstu.

Izrazi u programskom jeziku *Java*

```
1  public interface PozoviFunkciju {
2      void ispis(String s);  }
3
4  class Izrazi{
5   public static void ispisi(String s){
6              System.out.println("Vasa poruka: ");
7              System.out.println(s);
8      }
9
10     public static void ispisi1(String s){
11             System.out.println("Drugaciji ispis: ");
12             System.out.println(s);
13     }
14    }
15
16 PozoviFunkciju p = Izrazi::ispisi; //izraz reference metode
17         p.ispis("Kako si?");
18         p = Izrazi::ispisi1;
19         p.ispis("Kako si?");
```

Primjeri izraza ovisnih o kontekstu.

Izrazi u programskom jeziku *Java*

```
1  String d = "abracadabra";
2 List<String> ls = d.contains("b") ? Arrays.asList() : Arrays
      .asList("a","b");//uvjetni izraz
3
4 ArrayList<Integer> numbers = new ArrayList<Integer>();
5 numbers.add(5); numbers.add(9);
6 numbers.forEach( (n) -> { System.out.print(n+" "); } ); //
      lambda izraz
7
8 int v = 0;
9     double result = switch (v) {
10    case 0 -> 5;
11    case 1 -> 10;
12    default -> 15;
13   };//switch izraz
```

Primjeri izraza ovisnih o kontekstu.