Lambda izrazi - nastavak, kreiranje učitavača klase, korištenje modula, semantika višedretvenih programa

Matej Mihelčić

Prirodoslovno-matematički fakultet, Sveučilište u Zagrebu

matmih@math.hr

07. prosinca, 2022.

Lambda izrazi

Svaka lokalna varijabla, formalni parametar ili parametar iznimke koji se koriste ali nisu deklarirani unutar lambda izraza **moraju biti** `final` **ili efektivno** `final`. Svaka lokalna varijabla koja se koristi ali nije deklarirana unutar tijela lambda izraza **mora imati konačno pridijeljenu vrijednost prije tijela lambda izraza**.

```
1  interface Sucelje4 {  int f(); }
2
3  void m1(int x) {
4      int y = 1;
5      Sucelje4 s = () -> (x+y); //OK, x i y su efektivno final
6  }
7
8  void m2(int x) {
9      int y;
10     y = 1;
11      Sucelje4 s = () -> (x+y); //OK, x i y su efektivno
       final
12 }
```

Korištenje lokalnih varijabli u Lambda izrazima.

Lambda izrazi

```
1  void m3(int x) {
2      int y;
3      if (x>2) y = 1;
4      Sucelje4 s = () -> (x+y); //Nije OK, y je efektivno
       final ali nije konacno pridruzena vrijednost
5  }
6
7  void m4(int x) {
8      int y;
9      if (x>2) y = 1; else y = 2;
10     Sucelje4 s = () -> (x+y); //OK, x i y su efektivno final
11 }
12
13 void m5(int x) {
14     int y;
15     if (x>2) y = 1;
16     y = 2;
17     Sucelje4 s = () -> (x+y);//y nije efektivno final
18 }//dolazi do greske pri prevodenju
```

Korištenje lokalnih varijabli u Lambda izrazima.

Lambda izrazi

```
1  void m6(int x) {
2      int y = 1;
3      Sucelje4 s = () -> (x+y);
4      x++;//Nije OK, x nije efektivno final
5  }
6
7  interface Sucelje5 {  void f(); }
8
9  void m7(int x) {
10     Sucelje5 s = () -> (x=1);// Nije OK, x nije efektivno
       final
11 }
12
13 void m8() {
14     int y;
15     Sucelje5 s = () -> (y=1);// Nije OK, y-u nije konacno
       pridruzena vrijednost prije lambda izraza
16 }
```

Korištenje lokalnih varijabli u Lambda izrazima.

Lambda izrazi

```
1  interface Sucelje6 {  String f(); }
2
3  void m9(String[] polje) {
4      for (String s : polje) {
5          Sucelje6 suc = () -> s; //OK, s je efektivno final (
       nova varijabla u svakoj iteraciji)
6      }
7  }
8
9  void m10(String[] polje) {
10     for (int i = 0; i < polje.length; i++) {
11       Sucelje6 suc = () -> polje[i]; //Nije OK, varijabla i
       nije efektivno final (operand je operatora inkrement)
12     }
13 }
```

Korištenje lokalnih varijabli u Lambda izrazima.

Tip lambda izraza

Lambda izraz je **kompatibilan s ciljnim tipom** T u kontekstu pridruživanja, poziva ili pretvorbe ako je T **funkcijsko sučelje i izraz je usklađen s pravim funkcijskim tipom izvedenim** iz T.

Pravi funkcijski tip se **izvodi** iz T na sljedeći način:
- Ukoliko je T funkcijsko sučelje parametrizirano neodređenim tipom i **lambda izraz je eksplicitno zadan** (navođenjem tipova svih parametara), tada se pravi tip zaključuje.
- Ukoliko je T funkcijsko sučelje parametrizirano neodređenim tipom i **lambda izraz je implicitno zadan** (bez navođenja tipova varijabli ili korištenjem `var`), tada je pravi tip parametrizacija od T koja ne sadrži neodređeni tip.
- Inače je **ciljni tip** T.

Lambda izraz je **usklađen** s funkcijskim tipom ako vrijedi sve navedeno:
- Funkcijski tip **nema parametre tipa**
- Broj lambda argumenata je jednak broju argumenata (tipova) funkcijskog tipa

Tip lambda izraza

- Ukoliko je lambda izraz **eksplicitno zadan**, **tipovi njegovih formalnih argumenata su identični argumentima funkcijskog tipa**.
- Ukoliko se smatra da argumenti lambda izraza **imaju isti tip** kao i argumenti funkcijskog tipa:
  - Ukoliko je **povratni tip funkcijskog tipa** `void`, tada je tijelo lambda izraza **naredba** ili **blok koji je** `void`**-kompatibilan**.
  - Ukoliko je **povratni tip funkcijskog tipa** tip R, različit od `void` tada je ili tijelo lambda izraza neki **izraz kompatibilan** s R u **kontekstu pridruživanja** ili je tijelo lambda izraza **vrijednosno-kompatibilan blok i svaki izraz koji vraća rezultat je kompatibilan** s R u **kontekstu pridruživanja**.

Ukoliko je lambda izraz kompatibilan s ciljnim tipom T, tada je tip izraza (U) pravi tip koji je izveden iz T.

Dolazi do **greške pri prevođenju** ukoliko bilo koja klasa ili sučelje koje se koristi od U ili odgovarajućeg funkcijskog tipa **nije dostupna** u klasi ili sučelju u kojem se javalja lambda izraz.

Tip lambda izraza

Lambda izraze možemo koristiti i s funkcijskim sučeljima dostupnim u paketu `java.util.function`.

```
1  // Predikat (Predicate<T>) je parametrizirano funkcijsko
       sucelje s metodom koja prima jedan argument tip T i
       vraca rezultat tipa boolean
2  java.util.function.Predicate<String> p = s -> list.add(s);
3  // Potrosac (Consumer<T>) je parametrizirano funkcijsko
       sucelje s metodom koja prima jedan argument tipa T i ne
       vraca povratnu vrijednost
4  java.util.function.Consumer<String> c = s -> list.add(s);
5  //Funkcija (Function<T,R>) je parametrizirano funkcijsko
       sucelje s metodom koja prima dva argumenta tipa T i
       vraca rezultat tipa R
6   Function<String, String> f = (s,s1) -> s + " lambda "+s1;
7
8  //Navedena sucelja se koriste ovako:
9  boolean a = p.test("Nesto");
10 c.accept("Novi");
11 String vrijednost = f.apply("Rezultat ","izraza");
```

Korištenje lambda izraza.

Evaluacija lambda izraza pri izvođenju

Evaluacija lambda izraza pri izvođenju je **slična evaluaciji klasa** (normalno izvođenje rezultira stvaranjem reference na objekt). Evaluacija tijela lambda izraza se odvija **posebno** od izvođenja tijela lambda izraza.

Dolazi ili do **stvaranja i inicijalizacije nove instance klase** (može doći do `OutOfMemoryError`) ili do **referenciranja postojeće instance klase zadanih svojstava**.

Klasa objekta koji je referenciran nakon izvrednjavanja lambda izraza ima **sljedeća svojstva:**
- Klasa **implementira tip ciljnog funkcijskog sučelja** a ukoliko je ciljni tip tip presjeka, **svako sučelje koje čini presjek**.
- Gdje lambda izraz ima tip U, $\forall$ **ne-statičku metodu** m od U vrijedi:
  - Ukoliko funkcijski tip od U ima iste argumente ili argumente fiksiranog tipa u odnosu na m, tada **klasa deklarira metodu koja nadjačava** m. Tijelo te metode **izvrednjava tijelo lambda izraza** ukoliko je izraz odnosno **izvršava tijelo** ukoliko je blok te **vraća vrijednost** ukoliko se očekuje (ovisno o lambda izrazu).

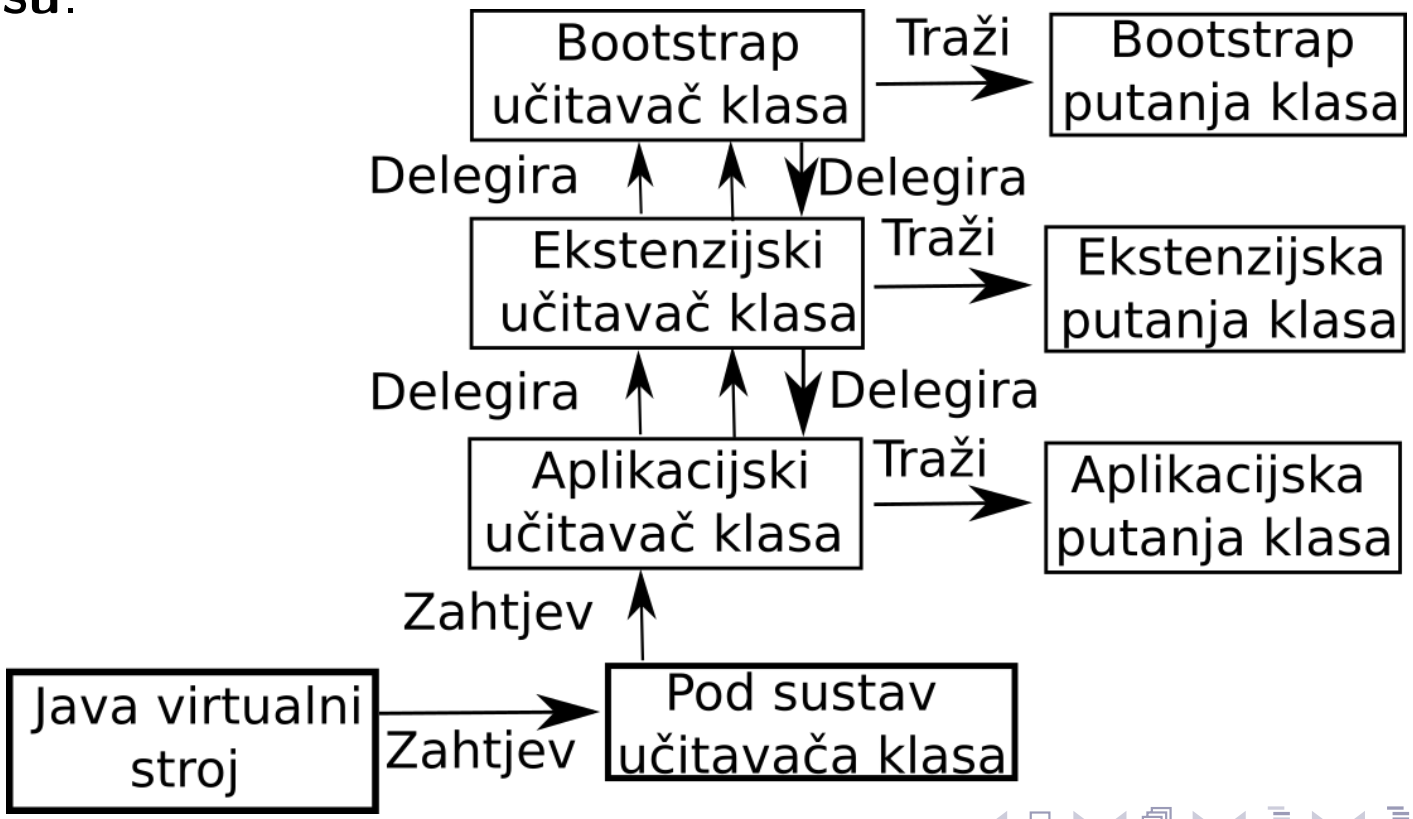
Evaluacija lambda izraza pri izvođenju

- Ukoliko se fiksirani tip metode koja je nadjačana **razlikuje po signaturi** od fiksiranog funkcijskog tipa U, tada prije izvršavanja ili izvrednjivanja tijela lambda izraza, **tijelo metode provjerava da je vrijednost svakog argumenta instanca podklase ili podsučelja** finksiranog tipa odgovarajućeg argumenta u funkcijskom tipu U. Ukoliko to nije slučaj dolazi do `ClassCastException`.

- Klasa **ne nadjačava niti jednu drugu metodu** ciljnog tipa funkcijskog sučelja ili drugih tipova sučelja no **može implementirati metode klase** `Object`.

Kreiranje učitavača klasa u *Javi*

Učitavač klase je u *Javi* reprezentiran klasom `ClassLoader` iz paketa `java.lang`. Služi učitavanju klasa **pri izvođenju** po potrebi. Učitava i **sve ovisnosti** klase. Učitavači klasa su **raspoređeni u hijerarhiju**, svaki učitavač (ukoliko nije učitao klasu), zahtjev za učitavanje prvo **proslijedi učitavaču roditelju**. Na taj način se osigurava da **samo jedan učitavač učita klasu**.



Kreiranje učitavača klasa u *Javi*

Učitavač klasa u *Javi* je baziran na tri principa:

- **Principu delegacije:** **prosljeđuje** zahtjev za čitanjem klase **učitavaču roditelju**. Učitava klasu samo ako učitavač roditelj **ne učita klasu ili ne može pronaći klasu**.
- **Principu vidljivosti:** učitavač klasa - dijete **može vidjeti** sve klase učitane od strane učitavača roditelja. Učitavač roditelj **ne može** vidjeti klase učitane od strane učitavača dijeteta.
- **Principu jedinstvenosti:** klasa se učitava samo **jednom**. Postiže se korištenjem principa delegacije koji osigurava da učitavač - dijete ne učitava klasu ukoliko je tu klasu već učitao učitavač roditelj.

Postoje tri vrste učitavača klasa:

- **Bootstrap učitavač klasa:** učitava standardni skup dokumenata klasa programskog jezika *Java* iz dokumenta `rt.jar` i `i18n.jar` (obično se nalaze u `lib` direktoriju ***Java* okruženja za izvođenje**). **Korijen** hijerarhije učitavača klasa.

Kreiranje učitavača klasa u *Javi*

- **Ekstenzijski učitavač klasa:** delegira učitavanje klasa roditelju. Ukoliko je učitavanje neuspješno, učitava klase iz `lib/ext` direktorija ***Java* okruženja za izvođenje**. Implementiran je u klasi `ExtClassLoader` ***Java* virtualnog stroja**. Može učitati klase iz proizvoljnog eksternog direktorija čija putanje je navedena u `java.ext.dirs`.
- **Sistemski učitavač klasa:** učitava klase specifične za aplikaciju iz varijable okruženja `CLASSPATH`. Taj učitavač je dijete ekstenzijskog učitavača i reprezentira se apstraktnom klasom `java.lang.ClassLoader` (sve klase sistemskog učitavača implementiraju tu klasu).

Java virtualni stroj prosljeđuje **puno ime** klase i poziva metodu `loadClass()` klase `java.lang.ClassLoader`. Metoda `loadClass()` poziva `findLoadedClass()` koja **provjerava** je li već **prije klasa učitana**. Ukoliko je klasa učitana, **delegira** poziv **roditelju** tog učitavača klase. Ukoliko učitavač klase ne može pronaći klasu, poziva metodu `findClass()` da bi klasu tražio u **datotečnom sustavu**.

Kreiranje učitavača klasa u *Javi*

```
1  public class NoviUcitavacKlasa extends ClassLoader {
2    @Override public Class<?> findClass(String ime) {
3      byte[] nb = ucitajPodatkeKlase(ime);
4      System.out.println("Klasa ucitana novim ucitavacem!");
5      return defineClass(ime, nb, 0, nb.length); }
6
7     private byte[] ucitajPodatkeKlase(String imeKlase) {
8        System.out.println(imeKlase);
9        InputStream ulaz = getClass().getClassLoader().
     getResourceAsStream(imeKlase.replace(".","/")+".class");
10       ByteArrayOutputStream nizBajtova = new
     ByteArrayOutputStream();
11       int len =0;
12       try {
13             while((len=ulaz.read())!=-1){
14             nizBajtova.write(len);}}
15             catch (IOException e) {
16                       e.printStackTrace(); }
17         return nizBajtova.toByteArray();  }   }
```

Definicija novog učitavača klasa.

Kreiranje učitavača klasa u *Javi*

```
1 public class Test {
2
3  public void ispisi(){
4         System.out.println("Pozdrav!");
5  }
6 }
```

Klasa koju želimo učitati.

```
1 package classloaderproba;
2 import java.lang.reflect.Constructor;
3 import java.lang.reflect.InvocationTargetException;
4 import java.lang.reflect.Method;
5
6  public static void main(String[] args){
7    NoviUcitavacKlasa loader = new NoviUcitavacKlasa();
8
9    try{
10     Class<?> c = loader.findClass("classloaderproba.Test");
11     Constructor<?> con = c.getDeclaredConstructor();
```

Učitavanje klase.

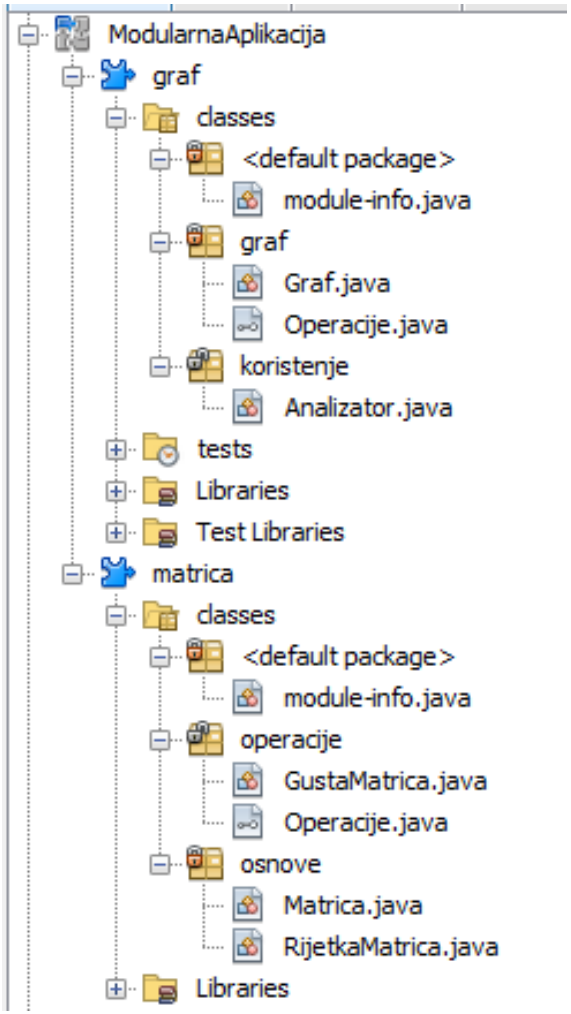
Kreiranje učitavača klasa u *Javi*

```
11      Object ob =  con.newInstance();//ne smijemo napraviti
    eksplicitnu pretvorbu u klasu Test! Zasto?
12      Method md = c.getMethod("ispisi");
13      md.invoke(ob);
14
15      ClassLoader parentLoader = Thread.currentThread().
    getContextClassLoader(); //standardni sistemski ucitavac
16      Class<?> c1 = parentLoader.loadClass("classloaderproba.
    Test");
17      Constructor<?> con1 = c1.getDeclaredConstructor();
18      Test ob1 =  (Test)con1.newInstance(); //smijemo
    napraviti eksplicitnu pretvorbu u klasu Test! Zasto?
19      ob1.ispisi();}
20      catch(ClassNotFoundException e){ e.printStackTrace();}
21      catch(IllegalAccessException e){ e.printStackTrace();}
22      catch(InstantiationException e){ e.printStackTrace();}
23      catch(NoSuchMethodException e){ e.printStackTrace();}
24    catch(InvocationTargetException e){e.printStackTrace();}
25     }   }
```

Učitavanje klase.

Korištenje modula

Module koristimo za grupiranje različitih (**povezanih**) paketa i kompilacijskih jedinica.



Korištenje modula pojačava enkapsulaciju unutar programa, pošto se eksplicitno specificira koji paketi se smiju koristiti od strane drugih modula (definiraju se ovisnosti). Moduli ili grupa modula se jednostavno nadograđuju i koriste za stvaranje kompleksnijih *Java* aplikacija.

Unutar programskog okruženja *Apache NetBeans*, modularnu aplikaciju stvaramo koristeći `File-> New Project -> Java Modular Project`.

Korištenje modula

Primjer

Napravimo modularnu aplikaciju koja će se sastojati od dva modula `matrica` i `graf`. Modul `matrica` će imati dva paketa: `osnove` i `operacije`, a modul `graf` će imati pakete `graf` i `koristenje`. Navedene module ćemo koristiti da bi za dani graf $\mathcal{G}$ izračunali sve parove čvorova između kojih postoji put duljine k u danom grafu.

- Paket `osnove`, modula `matrica`, sadrži definicije raznih tipova matrica i jednostavnije funkcije za dohvat ili postavljanje elemenata matrice.
- Paket `operacije`, modula `matrica`, sadrži sučelje `Operacije` koje sadrži osnovne matrične operacije (zbrajanje, oduzimanje, množenje, potenciranje).
- Klasa `GustaMatrica`, paketa `operacije`, nasljeđuje klasu `Matrica` i implementira sučelje `Operacije`.
- Modul `matrica` dopušta korištenje paketa `operacije` drugim modulima.

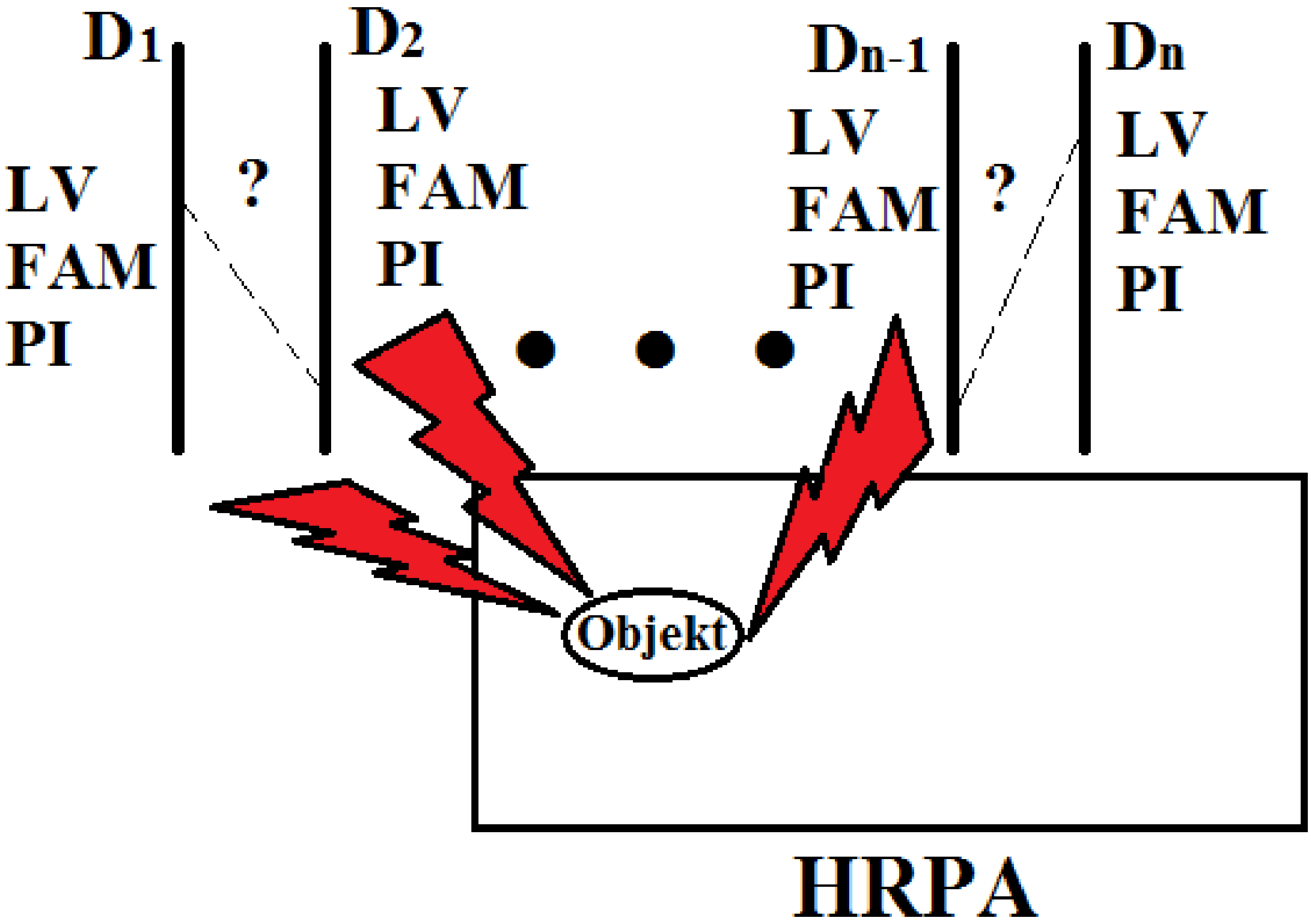
Korištenje modula

- Paket `graf`, modula `graf` sadrži definiciju grafa i osnovne funkcije za njegovu konstrukciju. Sučelje `Operacije` sadrži definiciju operacija nad grafovima koje implementira klasa `Graf`.
- Paket `korištenje` sadrži klasu `Analizator` koja kreira graf iz ulaznih podataka, računa i ispisuje sve parove čvorova između kojih postoji put duljine k.
- Modul `graf` ovisi (nužno treba za rad) modul `matrica` a dopušta korištenje paketa `korištenje` drugim modulima.

Nakon prevođenja modularnog projekta stvaraju se dva dokumenta `matrica.jar` i `graf.jar`. Ti dokumenti se navode u modularnim ovisnostima za prevođenje i izvođenje (`Project properties -> Libraries -> Modulepath` pod `Compile` i `Run`) projekata koji ih trebaju koristiti. Nemodularne `.jar` datoteke se logički tretiraju kao `automatski` moduli (stvara se modul istog imena kao uključena `.jar` datoteka).

Moduli se mogu koristiti i u nemodularnim aplikacijama, tada navedene `.jar` dokumente stavljamo u `Classpath` kao da se radi o bibliotekama.

Semantika višedretvenih programa



Semantika višedretvenih programa

Svaka dretva izvodi niz instrukcija po **pravilima izvođenja instrukcija za jednu dretvu u izolaciji** (instrukcije se izvode slijedno uz **moguće optimizacije prevodioca**). Zbog optimizacija prevodioca **ne možemo računati na apsolutni poredak instrukcija** (one se mogu izvršavati u izokrenutom poretku, neke vrijednosti se mogu spremiti u registre procesora i koristiti na više različitih mjesta u kodu bez osvježavanja itd). Takve optimizacije **ne utječu** na rezultat izvođenja naredbi kada se izvode od strane jedne dretve u izolaciji, međutim **mogu imati veliki utjecaj** kod paralelnog izvođenja naredbi od strane više dretvi.

Više paralelnih dretvi se fizički mogu izvoditi: a) **od strane jednog procesora vremenski ograničenim izvođenjem fragmenata koda uzastopno za svaku dretvu** (skup instrukcija početne dretve se izvodi ograničeno vrijeme, zatim se redom izvode fragmenti koda drugih dretvi nakon čega se nastavlja izvoditi kod početne dretve itd.), b) **od strane više procesora vremenski ograničenim izvođenjem fragmenata koda dretvi**. Niti kod jednog načina izvođenja **ne možemo točno predvidjeti konačni rezultat izvođenja**.

Semantika višedretvenih programa

Dretve su u programskom jeziku *Java* reprezentirane klasom `Thread` iz paketa `java.lang`. Svaka dretva sadrži **svoje lokalne varijable**, **formalne argumente metoda** i **parametre iznimaka** (ne može im se pristupati iz drugih dretvi) dok objektima, instancama referenciranih tipova (koji se spremaju na hrpi) **mogu pristupati sve dretve**.

Kakvi problemi se mogu javiti kada bi višedretvene programe izvršavali kao da se radi o jednodretvenim, odnosno bez ikakvih dodatnih ograničenja?

Pretpostavimo da paralelno izvodimo dvije dretve D_1 i D_2 na istom objektu `br` klase `Brojac` koje izvršavaju `run` metode klasa `Tprvi` i `Tdrugi`.
- Hoće li se prije dogoditi ispis varijable `b.broj` ili poziv `b.povecaj()` dretve D_1?
- Hoće li se prije izvršiti `if(b.broj == 1)` ili `b.povecaj()` dretve D_1?
- Hoće li dretva D_2 spremiti vrijednost varijable `b.broj` u registar procesora ili *cache* memoriju? U tom slučaju poziv `b.povecaj()` dretve T_1 neće biti registrirana.

Ovakve neodređenosti čine program **neupotrebljivim!**

Semantika višedretvenih programa

```
1  public class Brojac {
2    public int broj;
3    public void povecaj(){ broj++; }
4    public void ispisi(){System.out.println(broj);}
5  }
6
7  public class Tprvi implements Runnable {
8      Brojac b;
9      public Tprvi(Brojac br){ b = br; }
10     public void run() { b.povecaj(); }
11 }
12
13 public class Tdrugi implements Runnable {
14       Brojac b;
15     public Tdrugi(Brojac br){ b = br; }
16     public void run(){b.ispisi();
17       if(b.broj == 1} b.povecaj();}
18       }
```

Neispravna definicija klasa za višedretveno izvršavanje.

Semantika višedretvenih programa

Da bi se osigurao **identičan, točan i očekivan konačan rezultat**, bez obzira na način izvođenja individualnih dretvi kod višedretvenih programa, **uvodi se niz mehanizama** koji **garantiraju korektnost izvođenja**.

Najednostavniji mehanizam komunikacije između dretvi, kod višedretvenog izvođenja, je **sinkronizacija** koji je implementiran korištenjem **monitora** (pogledati Operacijske sustave).

Svaki objekt u *Javi* ima dodijeljen monitor kojega dretva može **zaključati** ili **otključati**. **Samo jedna dretva u danom vremenskom trenutku može držati lokot na monitoru**, a **ostale dretve koje pokušaju zaključati taj monitor su blokirane** dok ne mogu preuzeti vlasništvo nad lokotom tog monitora. Dretva **može zaključati monitor više puta** a **svako otključavanje poništava jednu operaciju zaključavanja**.

Sinkronizirana naredba dohvaća referencu objekta i pokuša zaključati pridruženi monitor.

Semantika višedretvenih programa

Dretva **čeka** dok ne dobije pristup lokotu, nakon čega **zaključava lokot** i **počinje izvoditi tijelo sinkronizirane naredbe**. Ukoliko izvođenje ikada **završi** (normalno ili neočekivano), **lokot se otključava**. **Sinkronizirana metoda** automatski zaključava monitor, tijelo metode se izvodi tek kada zaključavanje uspije, do onda dretva čeka pristup lokotu. Ukoliko je sinkronizirana **metoda instance**, zaključava se monitor **pridružen toj instanci**, ako je metoda statička, zaključava se monitor **pridružen** `Class` **objektu** koji **odgovara klasi koja sadrži statičku metodu**. Kod normalnog ili neočekivanog završetka izvođenja, **lokot se otključava**.

Programski jezik *Java* **ne detektira niti ne spriječava stanje potpunog zastoja** (eng. *deadlock*) koje se može javiti kod višedretvenih programa, već se za izbjegavanje istih mora pobrinuti programer.