

**priručnik za polaznike 2021 Srce**

**Objektno orijentirano programiranje u *Javi***

D470

Ovu su inačicu priručnika izradili:

Autor: Hrvoje Backović

Recenzent: Ivan Rančić

Urednica: Petra Gmajner

Lektorica: Mia Kožul



Sveučilište u Zagrebu

Sveučilišni računski centar

Josipa Marohnića 5, 10000 Zagreb

edu@srce.hr

ISBN 978-953-8172-63-2 (meki uvez)

ISBN 978-953-8172-64-9 (PDF)

Verzija priručnika D470-20210416



|  |  |
| --- | --- |
|  | Ovo djelo dano je na korištenje pod licencom *Creative Commons Imenovanje-Nekomercijalno-Dijeli pod istim uvjetima 4.0  međunarodna*. Licenca je dostupna na stranici:  http://creativecommons.org/licenses/by-nc-sa/4.0/. |

**Sadržaj**

[Uvod 1](#_Toc61953875)

[1. *Java* okruženje 3](#_Toc61953876)

[1.1. *Java* kao jezik, platforma i ekosustav 3](#_Toc61953877)

[1.2. Postupak prevođenja i izvedbe programa 4](#_Toc61953878)

[1.3. Pitanja za ponavljanje: *Java* okruženje 5](#_Toc61953879)

[2. Prvi program u *Javi* 7](#_Toc61953880)

[2.1. Pisanje i organizacija programskoga kôda 7](#_Toc61953882)

[2.2. Prevođenje, izvođenje i arhiviranje programa 14](#_Toc61953883)

[2.3. Vježba: Prvi program u *Javi* 18](#_Toc61953884)

[2.4. Pitanja za ponavljanje: Prvi program u *Javi* 18](#_Toc61953885)

[3. Osnove programskoga jezika *Java* 19](#_Toc61953886)

[3.1. Osnovni tipovi podataka 19](#_Toc61953888)

[3.2. Polja i stringovi 22](#_Toc61953889)

[3.3. Uvjetno izvođenje 27](#_Toc61953890)

[3.4. Petlje 32](#_Toc61953891)

[3.5. Ulazni i izlazni tokovi 38](#_Toc61953892)

[3.6. Pseudoslučajni brojevi 42](#_Toc61953893)

[3.7. Vježba: Osnove programskoga jezika *Java* 44](#_Toc61953894)

[3.8. Pitanja za ponavljanje: Osnove programskoga jezika *Java* 44](#_Toc61953895)

[4. Razredi i objekti 45](#_Toc61953896)

[4.1. Definiranje razreda i stvaranje objekata 45](#_Toc61953898)

[4.2. Dijagrami razreda 48](#_Toc61953899)

[4.3. Enkapsulacija 49](#_Toc61953900)

[4.4. Modifikatori pristupa 52](#_Toc61953901)

[4.5. Konstruktori 52](#_Toc61953902)

[4.6. Preopterećenje (engl. *Overloading*) 53](#_Toc61953903)

[4.7. Statički članovi 55](#_Toc61953904)

[4.8. Vježba: Razredi i objekti 58](#_Toc61953905)

[4.9. Pitanja za ponavljanje: Razredi i objekti 59](#_Toc61953906)

[5. Nasljeđivanje 61](#_Toc61953907)

[5.1. Hijerarhijska struktura razreda 61](#_Toc61953909)

[5.2. Nadjačavanje (engl. *Overriding*) 64](#_Toc61953910)

[5.3. Dinamički polimorfizam 66](#_Toc61953911)

[5.4. Bazni razred *Object* 68](#_Toc61953912)

[5.5. Apstraktni razredi i sučelja 70](#_Toc61953913)

[5.6. Ugniježđeni, lokalni i anonimni razredi 72](#_Toc61953914)

[5.7. Vježba: Nasljeđivanje 76](#_Toc61953915)

[5.8. Pitanja za ponavljanje: Nasljeđivanje 77](#_Toc61953916)

[6. Iznimke 79](#_Toc61953917)

[6.1. Primjeri iznimki 80](#_Toc61953919)

[6.2. Korištenje iznimki 81](#_Toc61953920)

[6.3. Stvaranje vlastitih iznimaka 88](#_Toc61953921)

[6.4. Vježba: Iznimke 90](#_Toc61953922)

[6.5. Pitanja za ponavljanje: Iznimke 90](#_Toc61953923)

[7. Parametrizacija kôda 91](#_Toc61953924)

[7.1. Parametrizacija metoda 92](#_Toc61953926)

[7.2. Parametrizacija razreda i sučelja 93](#_Toc61953927)

[7.3. Ograničavanje parametara i bezimeni parametri 97](#_Toc61953928)

[7.4. Vježba: Parametrizacija kôda 101](#_Toc61953929)

[7.5. Pitanja za ponavljanje: Parametrizacija kôda 102](#_Toc61953930)

[8. Kolekcije 103](#_Toc61953931)

[8.1. Lista 105](#_Toc61953933)

[8.2. Stog 109](#_Toc61953934)

[8.3. Red 111](#_Toc61953935)

[8.4. Vježba: Kolekcije 113](#_Toc61953936)

[8.5. Pitanja za ponavljanje: Kolekcije 113](#_Toc61953937)

[9. Oblikovni obrasci 115](#_Toc61953938)

[9.1. Kompozit (engl. *Composite Pattern*) 116](#_Toc61953940)

[9.2. Prototip (engl. *Prototype Pattern*) 120](#_Toc61953941)

[9.3. Iterator (engl. *Iterator Pattern*) 122](#_Toc61953942)

[9.4. Strategija (engl. *Strategy Pattern*) 126](#_Toc61953943)

[9.5. Vježba: Oblikovni obrasci 129](#_Toc61953944)

[9.6. Pitanja za ponavljanje: Oblikovni obrasci 129](#_Toc61953945)

[Završna vježba 130](#_Toc61953946)

[Literatura 135](#_Toc61953947)

# Uvod

Svrha ovoga priručnika jest upoznavanje s konceptima objektno orijentiranoga programiranja i temeljnim načelima oblikovanja kroz rad u programskom jeziku *Java*.

Preduvjet za razumijevanje gradiva ovoga priručnika jest poznavanje rada na računalu i poznavanje osnova programiranja u nekom programskom jeziku (npr. C/C++ ili Python). Priručnik se sastoji od devet poglavlja koja se obrađuju kroz pet dana, po četiri školska sata dnevno. Na kraju svakoga poglavlja nalaze se pitanja za ponavljanje, a većina poglavlja sadrži i praktičnu vježbu koja služi polaznicima da dodatno utvrde gradivo. Peti dan tečaja predviđen je za rješavanje završne vježbe koja obuhvaća sveukupno gradivo priručnika.

Na ovom tečaju polaznici će naučiti pisati programe u programskom jeziku *Java* koristeći objektno orijentiranu paradigmu. Nakon tečaja polaznici će razumjeti koncepte objektnoga programiranja te će biti u stanju primijeniti temeljna načela oblikovanja na nove i postojeće programske sustave koristeći oblikovne obrasce. Naučene koncepte moći će primijeniti u radu s drugim programskim jezicima.

Ovaj priručnik implementira koncepte objektno orijentiranoga razvoja u programskom jeziku *Java* jer omogućuje objektno orijentiranu paradigmu na vrlo intuitivan način zbog čega je idealan jezik za učenje koncepta.

U priručniku su važni pojmovi pisani **podebljano**. Ključne riječi, imena varijabli, funkcija, metoda i ostale programske konstrukcije pisane su drugačijim fontom od uobičajenog, na primjer: System.out.println("Hello World!"). Nazivi na engleskom jeziku pisani su *kurzivom* i u zagradi, na primjer, "razred(engl. *class*)".

**Programski kôd** pisan je na sljedeći način:

U prvom dijelu bit će napisan programski kôd.

|  |
| --- |
| for (int i = 0; i < 2; ++i)  System.out.println("Hello World!") |
| Izlaz:  Hello World!  U drugom dijelu bit će prikazan dobiveni izlaz programa.  Hello World! |

1. *Java* okruženje

Po završetku ovoga poglavlja moći ćete:

definirati osnovne pojmove vezane za Java okruženje

opisati postupak prevođenja i izvedbe Java programa.

* 1. *Java* kao jezik, platforma i ekosustav

*Java* je objektno orijentirani jezik koji su 1995. godine razvili James Gosling i drugi inženjeri u tvrtki ***Sun Microsystems***. Tvrtku je u međuvremenu preuzela ***Oracle*** korporacija koja se od tad brine za daljnji razvoj i održavanje jezika i platforme. *Java* jezik opisan je **specifikacijom** [2] koja sadrži detaljne informacije o svim aspektima samoga jezika poput tipova podataka, ključnih riječi, naredbama za upravljanje uvjetno izvođenje i sl.

Programski jezik *Java* oblikovan je idejom **„piši jednom, pokreni bilo gdje“** (engl. ***„write once, run anywhere“***) koja ilustrira prednost jezika da se jednom napisani i prevedeni program **bez preinaka** izvodi na mnoštvu različitih platformi i uređaja (računalo, mobitel, bazna stanica itd.).

*Java* programi pišu se za ***Javin* virtualni stroj** (engl. ***Java Virtual Machine***,**JVM**). To je apstraktni stroj definiran **specifikacijom** [2] koja strogo definira ponašanje stroja prilikom izvođenja prevedenih *Java* programa. *Javin* virtualni stroj služi kao **platforma** za izvođenje *Java* programa. Svaki uređaj sadrži posebnu implementaciju JVM-a koja slijedi specifikaciju čime se postiže prethodno spomenuta ideja pisanja višeplatformskih aplikacija.

*Java* jezik sam po sebi nije pretjerano koristan. Kako bi omogućio korisnicima brz i efikasan razvoj aplikacija, *Java* nudi **skup biblioteka** uz pomoć kojeg programeri mogu jednostavno implementirati razne funkcije kao što su čitanje i pisanje datoteka, spremanje podataka u efikasne strukture podataka, stvaranje grafičkih sučelja itd. Detaljne informacije o bibliotekama nalaze se u dokumentu ***Java* API** (***Java Application Programming Interface***) [3].

Kako bi određeni uređaj mogao pokretati *Java* programe, na njemu je potrebno postaviti **okruženje za pokretanje** (engl. ***Java Runtime Environment***,**JRE**). Ono sadrži implementaciju *Javinoga* virtualnog stroja i obvezni skup biblioteka što predstavlja minimum za pokretanje *Java* programa.

Za pisanje i prevođenje *Java* programa potreban je ***Java* razvojni komplet** (engl. ***Java Development Kit***,**JDK**). To je nadskup JRE-a koji osim toga sadrži implementaciju prevoditelja i drugih pomoćnih alata.

Posljednje verzije paketa JRE i JDK mogu se preuzeti sa službene stranice Oracle korporacije [4].

Ovaj priručnik koristi osnovnu verziju *Java* platforme pod nazivom ***Java* SE** (***Java Standard Edition***). Osim toga postoji i puno veća specifikacija ***Java* EE** (***Java Enterprise Edition***) koja uključuje i razne tehnologije za rad s bazama podataka, izradu web-stranica i dr. U sklopu ovoga tečaja koristit će se najnovija verzija standardne edicije *Jave*, što je u ovom trenutku ***Java* SE 14**.

Prednosti *Jave* nisu samo u jeziku i platformi koja sadrži mnoštvo različitih biblioteka. Oko jezika i platforme razvio se veliki **ekosustav** drugih biblioteka, tehnologija pa čak i programskih jezika stvorenih i održavanih od strane drugih ljudi i kompanija. Neki od najpoznatijih tehnologija unutar *Java* ekosustava su *JUnit*, *Spring*, *Grails*, *Gradle* itd.

U *Java* ekosustavu razvijeni su posebni prevodioci za postojeće programske jezike (npr. *Python*, *Haskell*, *Ruby, Javascript*) koji omogućuju takvim programima izvršavanje na JVM-u, čime zapravo postaju višeplatformski.

Osim *Jave* postoje i drugi jezici koji su stvoreni isključivo za JVM platformu kao što su *Clojure* i *Kotlin*.

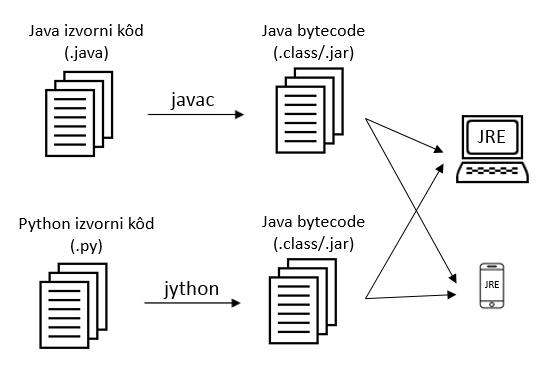
* 1. Postupak prevođenja i izvedbe programa

Programi pisani u *Javi* podijeljeni su u datoteke izvornoga kôda s ekstenzijom ***.java***. Koristeći *Java* prevoditelj (engl. ***Java compiler***, ***javac***), program će biti preveden u **međukôd** koji se naziva ***bytecode***. *Bytecode* datoteke imaju ekstenziju ***.class***, a mogu se i grupirati u arhive s ekstenzijom ***.jar*** s ciljem lakšega prenošenja. Na sličan način mogu se prevesti i drugi jezici koji to podržavaju. Npr. Python programi prevode se koristeći ***jython*** prevoditelj.

Prevedeni programi mogu se učitati u *Javin* virtualni stroj na bilo kojem uređaju koji ga podržava. Virtualni stroj će učitani *bytecode* **interpretirati**, što znači da će svaku naredbu zasebno prevoditi u strojni kôd i zatim izvršiti.

Prilikom interpretiranja, JVM može **optimizirati** dijelove programa tako da ih unaprijed prevede u strojni kôd. To se najviše isplati u dijelovima programa koji se često izvode. Takvo prevođenje obavlja **JIT prevoditelj** (***Just In Time Compiler***).

Cijeli postupak vizualiziran je na sljedećoj slici:



* 1. Pitanja za ponavljanje: *Java* okruženje

1. Kad je nastao programski jezik *Java* i tko ga trenutačno održava?
2. Što je *JVM* i čemu služi?
3. Što su *JRE* i *JDK*?
4. Što je *Java* ekosustav?
5. Nabrojite nekoliko programskih jezika za koje postoji podrška za *JVM*.
6. Kako se naziva rezultat (međukôd) prevođenja *Java* programa?
7. Što je *JIT* prevoditelj i koju funkciju obavlja?

1. Prvi program u *Javi*

Po završetku ovoga poglavlja polaznik će steći osnovno znanje potrebno za razvoj programa u Javi. Polaznik će moći:

koristiti Eclipse razvojno okruženje za pisanje programa

prevesti i izvesti prethodno napisane programe

arhivirati programe s ciljem lakšega prenošenja.

Ovaj priručnik koristi programski jezik ***Java*** kako bi prikazao i implementirao koncepte objektno orijentiranoga programiranja. Zbog jednostavnosti i olakšanoga razvojnog procesa koristi se razvojno okruženje ***Eclipse***. Postoje i druga popularna razvojna okruženja poput *IntelliJ* i *NetBeans*, ali oni neće biti obrađeni u ovom priručniku.

***Eclipse verzija***

Za potrebe ovoga tečaja koristi se verzija 2019-12 (4.14.0) razvojnog okruženja *Eclipse*.

Ovo poglavlje obrađuje osnove korištenja razvojnog okruženja te kroz vrlo jednostavan primjer daje detaljne upute za pisanje, prevođenje i izvođenje programa u *Javi*. Na kraju se obrađuje proces arhiviranja *Java* programa čime se olakšava prenošenje programa na druga računala.

1. 1. Pisanje i organizacija programskoga kôda
      1. Razvojno okruženje *Eclipse*

Kako bi se dodatno olakšalo pisanje programa, u sklopu tečaja koristi se **integrirano razvojno okruženje** (engl. ***Integrated Development Environment***,**IDE**) ***Eclipse*** koje nudi brojne mogućnosti čime se uvelike ubrzava razvoj, kao što su:

* Bojanje sintakse (engl. *syntax highlighting*)
* Dovršavanje kôda (engl. *code completion*)
* Prijavljivanje pogrešaka prilikom pisanja
* Otklanjanje pogrešaka (engl. *debugging*)
* Povezanost sa sustavima za verzioniranje kôda (engl. *version control*), npr. *Git*, *Subversion*.

*Eclipse* je vrlo popularna razvojna okolina koja nudi podršku za mnoštvo programskih jezika kao što su *C/C++*, *Haskell*, *Ruby*, *Python* itd. *Eclipse* je većinski implementiran u *Javi* te ga je moguće nadograditi dodacima (engl. *plugins*) čime se jednostavno ugrađuju dodatne funkcionalnosti (npr. *Checkstyle*, *FindBugs*).

*Eclipse* ima svoj **radni prostor** (engl. *workspace*) koji je potrebno definirati prilikom pokretanja. U tom prostoru nalaze se **projekti** unutar kojih se pišu programi.

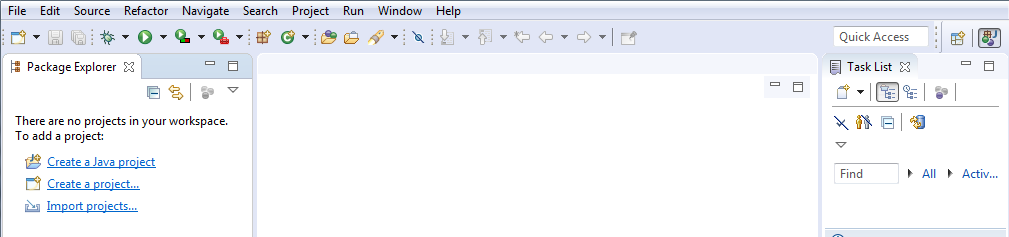
➎

➍

➌

➋

➊



1. Glavna traka izbornika (engl. *Menu bar*).
2. Hijerarhijski pregled projekata, paketa i datoteka.
3. Alatna traka s komandama za prevođenje i pokretanje programa, kreiranje paketa, razreda itd.
4. Glavni prozor za pregled i pisanje programa.
5. Gumbi za odabir perspektive. **Perspektiva** je razmještaj prozora i pogleda unutar programa *Eclipse* čime se može efikasno prebacivati u drugi način rada (npr. iz pisanja programa u pokretanje i otklanjanje pogrešaka). Pretpostavljena perspektiva je ***Java* perspektiva** koja se koristi za razvoj *Java* programa.
   * 1. Stvaranje novoga projekta

Kako bi započeo razvoj *Java* programa, potrebno je najprije stvoriti projekt koristeći kraticu iz alatne trake ili glavnu traku izbornika odabirom *File* 🡪 *New* 🡪 ***Java Project***. Pritom se otvara novi dijaloški okvir s postavkama projekta koje je preporučljivo konfigurirati kao na slici:

➊



➏

➎

➍

➌

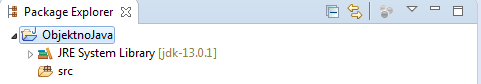
➋

1. Ime projekta, pri čemu se najčešće koristi *Pascal Case*, gdje su riječi pisane spojeno te svaka riječ počinje velikim slovom.
2. Lokacija projekta na računalu. Pretpostavljena i preporučena vrijednost je unutar radnoga prostora *Eclipsea*.
3. Odabir specifične verzije JRE-a koja će se koristiti unutar projekta. Preporučuje se korištenje najnovije verzije.
4. Odabir razmještaja datoteka unutar projekta. Zbog jednostavnosti se preporučuje odabir odvojenih direktorija za izvorne (.java) i prevedene (.class) datoteke. Pritom se ti direktoriji najčešće nazivaju **src** (engl. ***source***) i **bin** (engl. ***binary***).
5. Pritiskom na *Next* dobije se niz drugih postavki projekta kojima se konfiguriraju vanjske biblioteke, međuovisnosti projekata i dr. Za potrebe tečaja nije nužno mijenjati te postavke.
6. Pritiskom na *Finish* kreira se projekt sa svim odabranim postavkama.

*Eclipse* će s lijeve strane nakon stvaranja projekta prikazati hijerarhijsku strukturu projekta prikazanu na slici:

➊

➋



1. Direktorij s paketima i izvornim (.java) datotekama projekta. Cjelokupni kôd projekta nalazit će se unutar ovog direktorija.
2. Arhivirane standardne biblioteke iz JDK-a koje su dostupne za korištenje unutar projekta.
   * 1. Stvaranje paketa

Programi u *Javi* tipično su raspodijeljeni u **pakete** (engl. ***packages***) koji služe za logičku i hijerarhijsku raspodjelu izvornih programa s ciljem lakšega snalaženja. Uz to, oni služe kao **imenski prostori** (engl. ***namespaces***) koji omogućavaju imenovanje više izvornih datoteka istim nazivom.

Zbog osiguravanja jednoznačnosti, u *Javi* nije dozvoljeno definiranje više datoteka istog imena. To bi brzo postao veliki problem jer već postoji mnoštvo datoteka unutar standardnih biblioteka JDK-a pa bi se vrlo lako dogodilo da definiramo datoteku s postojećim imenom. Iz tog razloga postoje paketi čije se ime nadodaje punom imenu datoteke. Tako će, na primjer, *MojaDatoteka.java* unutar paketa *moj.paket* imati puno ime ***moj.paket.MojaDatoteka.java***.

U praksi se za imenovanje paketa koristi kombinacija **obrnute internetske domene** neke kompanije ili institucije te **naziva projekta ili softvera**, na primjer, *hr.unizg.srce.d470.iznimke*. Ovakvo imenovanje služi za sprečavanje kolizija u imenima razreda prilikom korištenja programa koji su razvijeni u različitim kompanijama i sl.

Datoteke koje ne pripadaju nijednom paketu smještene su automatski u **neimenovani paket** (engl. ***unnamed/default package***). Ovo je loša praksa koju službena dokumentacija jezika ne preporučuje.

Novi paket može se stvoriti korištenjem kratice iz alatne trake ili odabirom *File* 🡪 *New* 🡪 ***Package*** u glavnoj izbornoj traci. Pritom se otvara novi dijaloški okvir u kojem je dovoljno specificirati ime paketa. Prilikom imenovanja paketa uobičajeno je koristiti mala slova.

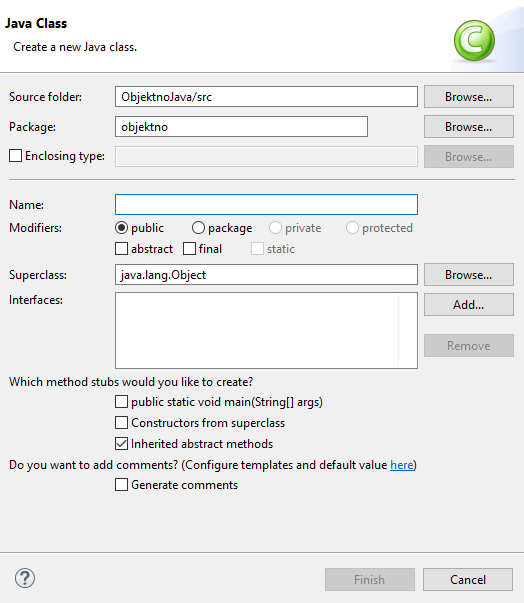
Nakon stvaranja paketa stvoren je i istoimeni direktorij u radnom prostoru *Eclipsea*. Struktura paketa ocrtava strukturu direktorija unutar projekta čime se elegantno razdjeljuju izvorne datoteke.

*Java* nema podršku za definiranje **potpaketa** unutar drugih paketa, ali je moguće definirati dublju strukturu direktorija koristeći točku u imenu paketa. Tako će, na primjer, paket *moj.veliki.paket* stvoriti lanac direktorija ***moj\veliki\paket***. Moguće je uz to definirati i paket *moj*, ali je važno napomenuti da on po definiciji nije nadpaket od *moj.veliki.paket*, iako dijele strukturu direktorija.

* + 1. Stvaranje razreda

Programski kôd u *Javi* podijeljen je u **razrede** (engl. ***classes***). U praksi Java projekti sadrže mnoštvo razreda pa je zato uobičajeno i preporučeno da se implementacija svakoga razreda piše u zasebnu istoimenu datoteku. Na primjer, razred *Trokut* bit će implementiran u datoteci *Trokut.java*. Detaljnije o razredima i objektima u kasnijim poglavljima.

Kako bi napisali prvi program u *Javi*, potrebno je stvoriti novi razred korištenjem kratice iz alatne trake ili odabirom *File* 🡪 *New* 🡪 ***Class*** u glavnoj izbornoj traci. Pritom se otvara novi dijaloški okvir prikazan na slici:



➊

➒

➏

➑

➐

➎

➍

➌

➋

1. Bazni direktorij izvornoga kôda. Preporučuje se odabir *src* direktorija unutar projekta.
2. Ime paketa u koji će biti smješten razred.
3. Opcija za stvaranje ugniježđenoga razreda.
4. Ime razreda.
5. Modifikatori razreda.
6. Odabir baznoga razreda.
7. Lista sučelja koju razred implementira.
8. Opcije za stvaranje krnjih metoda kao što su ***main*** ili naslijeđene apstraktne metode.
9. Opcija za automatsko stvaranje dokumentacije, odnosno *Javadoc* komentara unutar razreda.

Za pisanje prvoga programa dovoljno je upisati ime razreda te odabrati stvaranje metode ***main*** (točke 4. i 8.). Ostale opcije trebale bi imati pretpostavljene vrijednosti koje su u ovom trenutku dovoljne. Svi nepoznati pojmovi kao što su sučelja, ugniježđeni razredi i modifikatori pristupa bit će detaljno objašnjeni u kasnijim poglavljima.

Važno je napomenuti da prethodno navedene opcije služe kako bi prilikom stvaranja razreda dobili pripremljenu datoteku izvornoga kôda s djelomično napisanim programom. Time se ubrzava proces razvoja te se smanjuje mogućnost pogreške. U slučaju da se neka opcija zaboravi postaviti, može se naknadno ručno implementirati u samom programu.

* + 1. Pisanje programa

Nakon stvaranja razreda, u prozoru razvojnog okruženja otvorit će se nova datoteka s izvornim tekstom programa:

|  |
| --- |
| **package** hr.unizg.srce.d470;  **public** **class** PrviProgram {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  }  } |

Ovaj program definira razred PrviProgram unutar paketa hr.unizg.srce.d470. Taj razred ima jednu metodu main koja služi kao početna točka izvršavanja programa. Svaki *Java* program mora imati barem jednu takvu metodu main od koje će krenuti. Ovako napisan program je valjan, ali zasad ne radi ništa dok ga se ne nadogradi.

Ključna riječ **package** nakon koje slijedi ime paketa predstavlja naredbu kojom se definira paket u kojem će se nalaziti taj razred. Važno je primijetiti da svaka naredba završava znakom „;“ kao što je slučaj u mnogim drugim programskim jezicima.

Ključna riječ **class** nakon koje slijedi ime razreda služi za definiranje razreda. Nakon imena slijede vitičaste zagrade unutar kojih se nalazi tijelo razreda. Nakon vitičastih zagrada ne stavlja se znak „;“.

Ključna riječ **public** predstavlja modifikator pristupa koji se može koristiti ispred definicija razreda, metoda ili atributa. Ovaj modifikator označava da su razred PrviProgram i metoda main javni (engl. *public*) što znači da su vidljivi drugim razredima čak i izvan paketa u kojem se nalaze. Postoji više modifikatora koji su detaljnije obrađeni u kasnijim poglavljima.

U tijelu razreda nalazi se metoda main u kojoj počinje izvođenje programa. Metode u *Javi* funkcioniraju jednako kao funkcije u brojnim drugim programskim jezicima.

Metoda main kao **ulazni parametar** prima String[] args. To je polje, odnosno niz elemenata tipa String koje program dobije prilikom pokretanja. String je jedan od brojnih tipova podataka u *Javi* te predstavlja niz znakova (npr. *„jabuka“*). Osnovni tipovi podataka bit će obrađeni u nastavku.

Ispred imena metode nalazi se **povratni tip podataka** koji metoda treba vratiti nakon izvođenja. U ovom slučaju to je ključna riječ **void** koja označava da metoda zapravo ne vraća ništa.

Ispred imena metode main nalazi se i ključna riječ **static**. To je također modifikator koji označava da se radi o statičkoj metodi odnosno **metodi razreda** (engl. *class method*). Statički članovi bit će objašnjeni naknadno.

U programima je moguće pisati **komentare** koji služe da bi dali objašnjenja za određene dijelove kôda te tako povećavaju čitljivost. Komentare je moguće pisati u trima oblicima:

* **Jednolinijski komentari** počinju znakovima // i protežu se do kraja istoga retka. Takav komentar vidljiv je i u prethodnom primjeru:

// **Obični jednolinijski komentar.**

* **Višelinijski komentari** počinju znakovima /\* i završavaju prvim pojavljivanjem znakova \*/.

/\* Ovo je primjer

\* višelinijskog

\* komentara.  
 \*/

* **Dokumentacijski komentari**, odnosno ***Javadoc*** komentari kojima se piše službena dokumentacija. Ovi komentari počinju znakovima /\*\* i završavaju prvim pojavljivanjem znakova \*/. Obično se pišu iznad definicija razreda, metoda i atributa kako bi ih pobliže opisali:

***Javadoc* komentari**

U dokumentacijskim komentarima koriste se predefinirane oznake (engl. ***tags***), npr. **@param** i **@return**, kojima se oblikuje dokumentacija.

Moguće je automatski generirati kompletnu dokumentaciju projekta u HTML formatu.

Više informacija o *Javadoc* komentarima može se pronaći u dodatnoj literaturi [5].

/\*\*

\* Glavna metoda prvog programa.

\*

\* **@param** args Ulazni parametri programa.

\*/

**public** **static** **void** main(String[] args) {

Da bi prvi program bio zanimljiviji, može se nadograditi da ispisuje tekst na ekran na sljedeći način:

|  |
| --- |
| **package** hr.unizg.srce.d470;  **public** **class** PrviProgram {  **public** **static** **void** main(String[] args) {  System.***out***.println("Prvi program.");  }  } |
| Izlaz:  Prvi program. |

***Eclipse* predlošci kôda**

Naredba za ispisivanje može se brže napisati uz pomoć predložaka kôda u razvojnom okruženju *Eclipse*. Dovoljno je napisati *„sysout“* te pritisnuti kombinaciju tipki Ctrl + Space.

Popis postojećih predložaka može se pronaći odabirom *Window* 🡪 ***Preferences*** u glavnoj izbornoj traci, nakon čega se otvara dijaloški okvir u kojem je potrebno odabrati *Java* 🡪 *Editor* 🡪 ***Templates***.

Program poziva metodu println koja služi za ispisivanje sadržaja na ekran, a kao argument joj se predaje niz znakova "Prvi program." koji će se ispisati. Metodi println mogu se predati i drugi tipovi podataka kao što su brojevi, znakovi ili drugi objekti.

Važno je napomenuti da se metoda println nalazi u drugom razredu (konkretno PrintStream) pa ju je potrebno pozvati koristeći objekt toga razreda (***out***) koji se pak nalazi unutar razreda System. Razredi i objekti bit će detaljno obrađeni u sljedećem poglavlju.

Prilikom pisanja programa vrlo je važan **stil pisanja** **kôda** (engl. ***code style***) koji mora biti elegantan i čitljiv. To je osobito važno kad na istom projektu sudjeluje više ljudi koji će biti vrlo zahvalni ako prilikom čitanja taj kôd bude lijepo oblikovan. Iz tog razloga preporučuje se pridržavanje **jednom** stilu pisanja. U praksi je čest slučaj da postoje strogi mehanizmi koji će automatski spriječiti mijenjanje programa ako ta promjena nije u skladu s predefiniranim stilom pisanja.

Razvojno okruženje *Eclipse* nudi kraticu **Ctrl + Shift + F** kojom će se trenutno otvoreni program (bez sintaksnih pogrešaka) **formatirati** kako bi se prilagodio ugrađenom stilu pisanja. Ugrađeni stil pisanja moguće je dorađivati u postavkama okruženja odabirom *Window* 🡪 *Preferences* u glavnoj izbornoj traci te *Java* 🡪 *Code Style* 🡪 ***Formatter*** u otvorenom dijaloškom okviru. Postoje brojni gotovi stilovi pisanja koje je moguće dohvatiti putem Interneta i uvesti u *Eclipse*.

* 1. Prevođenje, izvođenje i arhiviranje programa

U ovom potpoglavlju bit će objašnjeno kako prevesti, izvesti te naposljetku arhivirati prethodno napisani prvi program u *Javi*. Najprije će biti opisan postupak koji koristi komandnu liniju, a zatim će biti objašnjeno kako jednostavnije to napraviti u razvojnom okruženju *Eclipse*.

* + 1. Korištenje komandne linije

Nakon uspješne instalacije JDK-a, na *Windows* računalima mogu se pronaći izvršni programi potrebni za razvoj programa u *Javi*. Oni se najčešće nalaze u ***C:\Program Files\Java\jdk-xx.x.x\bin***za određenu verziju JDK-a. Kako bi se tim programima moglo pristupati iz bilo kojega dijela sustava, potrebno je tu putanju dodati u ***PATH*** varijablu okruženja. U nastavku je prikazano kako koristiti te programe za prevođenje, izvođenje i arhiviranje *Java* programa.

**Varijable okruženja**

Na *Windows* računalima postoje varijable okruženja (engl. ***environment variables***) među kojima je i varijabla ***PATH*** koja sadrži putanje koje će preko komandne linije biti dostupne iz bilo kojeg dijela sustava.

Na *Windows* *10* računalima dovoljno je upisati *„environment variables“* u okvir pretraživanja na programskoj traci kako bi došli do cijelog popisa varijabli koje je moguće mijenjati.

Za prevođenje *Java* programa u *bytecode* koristi se *Java* prevoditelj (engl. *compiler*) u obliku izvršnoga programa ***javac***. U najjednostavnijem slučaju, *javac* kao argument prima izvornu (*.java*) datoteku koju treba prevesti. Prethodno napisani prvi program prevodi se tako da se u komandnoj liniji pozicionira na izvorni direktorij projekta (*src*) te se pokrene naredba:

**javac hr\unizg\srce\d470\PrviProgram.java**

čime se u istom direktoriju pored izvorne datoteke stvara i ***.class*** datoteka koja sadrži ***bytecode*** programa.

Za izvršavanje prevedenih *Java* programa koristi se izvršni program ***java*** koji pokreće lokalnu implementaciju *Javinoga* virtualnog stroja (*JVM*) te izvršava dani program na njemu. Izvršni program *java* kao argument prima puno ime razreda koji se pokreće te koji treba sadržavati metodu main gdje započinje izvršavanje programa. Prethodno prevedeni prvi program može se izvršiti korištenjem naredbe:

**Komandna linija**

Komandna linija je tekstualno okruženje pomoću kojeg korisnik zadaje različite naredbe koje operacijski sustav treba izvršiti.

Na *Windows 10* računalima komandna linija može se pokrenuti upisivanjem „cmd“ u okvir pretraživanja na programskoj traci.

**java hr.unizg.srce.d470.PrviProgram**

pri čemu se na ekranu ispisuje poruka. U praksi *Java* programi sadrže mnoštvo razreda što rezultira s puno *.class* datoteka čime postaje nepraktično dijeliti prevedene *Java* programe, na primjer s drugim ljudima i računalima. Iz tog razloga prevedeni *Java* programi pakiraju se u **arhive** (engl. ***Java archive***) čime se cijeli *Java* projekti mogu svesti na jednu datoteku. *Java* arhivi po prirodi su slični *.zip* arhivima, s ponekim razlikama i drugačijom ekstenzijom: ***.jar***. Prethodno prevedeni prvi program može se arhivirati korištenjem naredbe:

***Java* arhivi**

*Java* arhivi mogu se otvoriti s bilo kojim programom za rukovanje arhivima poput *.zip* ili *.rar* datoteka.

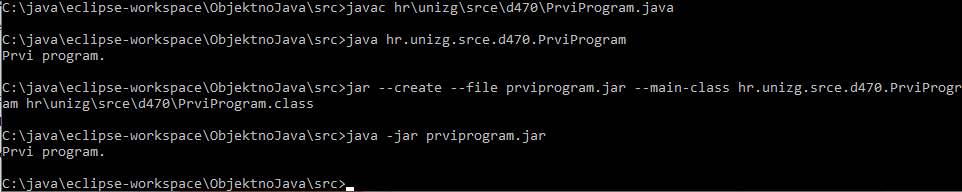
U *Java* arhivu može se pronaći datoteka ***MANIFEST.MF*** koja sadrži informacije o arhiviranim *Java* programima. Između ostaloga, sadrži parametar ***Main-Class*** s punim imenom razreda u kojem će krenuti izvođenje programa.

**jar –-create –-file prviprogram.jar   
 –-main-class hr.unizg.srce.d470.PrviProgram   
 hr\unizg\srce\d470\PrviProgram.class**

pri čemu se u trenutnom direktoriju stvara arhiv ***prviprogram.jar***. Tako arhivirani program može se direktno pokrenuti korištenjem naredbe:

**java -jar prviprogram.jar**

Cjelokupni postupak prevođenja, arhiviranja i izvršavanja programa u komandnoj liniji prikazan je na slici:



* + 1. Korištenje razvojnog okruženja

Prethodno opisani proces prevođenja, izvršavanja i arhiviranja programa postaje nepraktičan uvođenjem više razreda i paketa. Time prethodne naredbe postaju komplicirane za korištenje te je vrlo lako napraviti grešku.

Iz tog razloga preporučuje se korištenje razvojnog okruženja *Eclipse* koje ima ugrađenu podršku za prevođenje, izvršavanje i arhiviranje programa. Prednost razvojnog okruženja je u tome što korisnik ne treba brinuti o postavljanju raznih parametara tijekom cijelog procesa, već se sve obavlja automatski pritiskom na gumb.

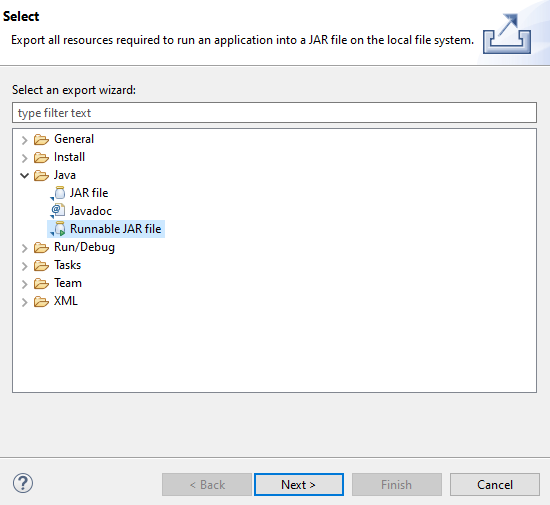
Napisani program može se izvršiti korištenjem kratice iz alatne trake ili odabirom *Run* 🡪 *Run As* 🡪 ***Java Application*** u glavnoj izbornoj traci. U donjem dijelu prozora razvojnog okruženja nalazi se okvir u kojem će se pojaviti izlaz iz programa (u ovom slučaju tekst *„Prvi program.“*).

U *Eclipseu* je moguće definirati konfiguracije izvršavanja (engl. *run configurations)*odabirom *Run* 🡪 ***Run Configurations***u glavnoj izbornoj traci. U konfiguracijama je između ostalog moguće namjestiti **ulazne parametre programa** te definirati **glavni razred** koji će biti početna točka za izvršavanje programa.

Osim uobičajenog izvršavanja, *Java* programe moguće je pokretati u načinu uklanjanja pogrešaka (engl. ***debug mode***) te načinu za računanje pokrivenosti kôda (engl. ***coverage mode***). Više informacija o tome može se pronaći u službenom priručniku razvojnog okruženja *Eclipse* [6].

Važno je napomenuti da će izvršeni program biti automatski preveden prije samog izvođenja. Prevedene (*.class*) datoteke bit će spremljene u ***bin*** direktoriju unutar projekta. Eventualne pogreške koje prevoditelj može prijaviti bit će prikazane odmah prilikom pisanja programa čime se olakšava i ubrzava proces ispravljanja pogrešaka.

Postojeće projekte moguće je arhivirati odabirom *File* 🡪 ***Export*** u glavnoj izbornoj traci razvojnog okruženja. Pritom se otvara dijaloški okvir kao na slici:



Za kreiranje arhiva koji se može direktno izvršiti, potrebno je odabrati „***Runnable JAR file***“ te izabrati odgovarajući konfiguraciju izvršavanja čime se određuje glavni razred u kojem će program započeti s izvršavanjem.

* 1. Vježba: Prvi program u *Javi*

Sljedeće zadatke potrebno je riješiti isključivo korištenjem *Eclipse* razvojnog okruženja. Pridržavajte se konvencija za imenovanje paketa, razreda i sl.

1. Otvorite razvojno okruženje *Eclipse* te odaberite radni prostor.
2. Stvorite novi *Java* projekt.
3. Unutar projekta stvorite novi paket.
4. Stvorite novi razred unutar paketa te u glavnoj funkciji ispišite proizvoljnu poruku.
5. Pokrenite program te se uvjerite da se poruka ispisala.
6. Arhivirajte program u .jar datoteku.
   1. Pitanja za ponavljanje: Prvi program u *Javi*
7. Što su *Java* paketi i čemu služe?
8. Koja se naredba koristi za definiranje paketa?
9. Koja se ključna riječ koristi za definiranje razreda?
10. Gdje počinje izvođenje svakoga *Java* programa?
11. Koja se metoda koristi za ispisivanje teksta?
12. Koji sve oblici komentara postoje u *Javi*?
13. Koja se kratica u razvojnom okruženju *Eclipse* koristi za formatiranje teksta programa?
14. Kako se zovu izvršni programi za prevođenje, izvođenje i arhiviranje *Java* programa?
15. Osnove programskoga jezika *Java*

Po završetku ovoga poglavlja polaznik će upoznati osnovne značajke programskoga jezika Java. Polaznik će moći:

navesti i koristiti osnovne tipove podataka

koristiti uvjetne naredbe i petlje

učitavati i ispisivati podatke koristeći ulazno-izlazne tokove

Ovo poglavlje obrađuje osnovne značajke jezika poput tipova podataka, uvjetnih konstrukcija i petlji. Te značajke nužan su preduvjet za praktičnu primjenu objektno orijentirane paradigme u kasnijim poglavljima.

1. 1. Osnovni tipovi podataka

Programski jezik *Java* je **statički tipiziran**, što znači da je sve varijable potrebno deklarirati prije korištenja. Time se unaprijed i jednoznačno određuju tip, ime i inicijalna vrijednost varijable. Tip podataka definira vrijednosti koju određena varijabla može poprimiti te operacije koje se mogu izvršiti nad njom. U *Javi* postoje dvije skupine tipova podataka: **reference i primitivni tipovi podataka**.

**Reference** su tip podataka koji čuva memorijsku lokaciju objekta nekoga razreda. Kao što će biti prikazano u idućem poglavlju, stvaranjem objekta rezervira se potreban memorijski prostor, a u programu se tom objektu pristupa preko reference. Reference se deklariraju koristeći ime razreda na čije će objekte pokazivati. U nastavku je prikazano stvaranje objekta tipa String koji sadrži tekst "Moja poruka". Varijabla poruka je referenca koja čuva memorijsku lokaciju tog objekta.

String poruka = **new** String("Moja poruka");

Važno je napomenuti da su objekti u *Javi* **dinamički alocirani** što znači da se njihov memorijski prostor zauzima na **gomili** (engl. ***heap***) tijekom izvođenja programa. Za razliku od nekih drugih jezika, alociranu memoriju nije potrebno eksplicitno oslobađati, već se za to brine *Javin* automatski **sakupljač smeća** (engl. ***Garbage Collector***).

**Primitivni tipovi podataka** definirani su specifikacijom jezika te njihova imena pripadaju skupu ključnih riječi jezika. Za razliku od referenci, varijable primitivnih tipova podataka ne čuvaju memorijsku lokaciju, već konkretnu vrijednost koja joj je pridjenuta. Postoji osam primitivnih tipova podataka koji se razlikuju po veličini memorije koju zauzimaju te vrijednostima koje mogu poprimiti.

U tablici je prikazan popis tipova podataka s informacijama o veličini memorijske lokacije koju zauzimaju, rasponu vrijednosti te inicijalnim vrijednostima.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tip podataka** | **Veličina memorijske lokacije [bit]** | **Raspon vrijednosti** | **Inicijalna vrijednost (za atribute)** |
| byte | 8 |  |  |
| short | 16 |  |  |
| int | 32 |  |  |
| long | 64 |  |  |
| float | 32 |  |  |
| double | 64 |  |  |
| char | 16 |  |  |
| boolean | - |  |  |
| (referenca) | - | - |  |

Veličina memorijske lokacije za reference i *boolean* nije strogo definirana specifikacijom jezika jer ovisi o konkretnoj implementaciji *Javinoga* virtualnog stroja. Iako je za logički tip *boolean* dovoljan jedan bit za kodiranje informacije — vrijednost 1 za istinu i 0 za laž, na računalima to nije praktično te se najčešće koristi veličina 8 bita. Za reference je u praksi najčešći slučaj da imaju veličinu od 32 bita na 32-bitnim arhitekturama računala i 64 bita na 64-bitnim arhitekturama.

**Decimalni brojevi**

Za spremanje decimalnih brojeva preporuča se korištenje razreda **double** umjesto **float**, osim u programima gdje je količina memorije vrlo ograničena.

Važno je napomenuti da se **lokalne varijable** (deklarirane unutar metoda) **ne inicijaliziraju automatski** na vrijednosti iz prethodne tablice. Glavni razlog tome jest brzina izvođenja programa. Prevoditelj će prijaviti upozorenje u slučaju korištenja neinicijalizirane lokalne varijable.

Također, u *Javi* je, kao i u mnogim drugim jezicima, moguće definirati **nepromjenjive varijable** korištenjem modifikatora **final** na sljedeći način:

**final int** broj = 42;

Ovako definirana varijabla ne može promijeniti vrijednost te će svaki pokušaj promjene rezultirati greškom pri prevođenju. Osim varijabli, moguće je definirati i **nepromjenjive argumente funkcija**. Takvi argumenti ne mogu promijeniti vrijednost u tijelu te funkcije.

* + 1. Operatori

U programskom jeziku *Javi* postoji skup operacija koje je moguće obaviti nad osnovnim tipovima podataka. U sljedećoj tablici navedeni su svi postojeći **operatori**:

|  |  |
| --- | --- |
| **Operator** | **Opis** |
| = | Operator pridruživanja |
| +, - | Unarni operatori predznaka |
| +, -, \*, /  +=, -=, \*=, /= | Binarni operatori zbrajanja, oduzimanja, množenja i dijeljenja |
| %, %= | Ostatak pri dijeljenju |
| ++, -- | Inkrement/dekrement vrijednosti  (mogu biti prefiks ili postfiks) |
| <<, >>  <<=, >>= | Operatori binarnoga posmaka |
| >>>  >>>= | Operatori logičkoga posmaka |
| ~, &, |, ^  &=, |=, ^= | Bitovni operatori komplementa, i/ili, isključivo ili |
| !, &&, || | Logički operatori komplementa, i/ili |
| <, <=, >, >=, ==, != | Logički operatori (ne)jednakosti |
| ? : | Ternarni uvjetni operator |
| -> | Deklaracija lambda izraza |

* + 1. Omotači primitivnih tipova podataka

U *Javi* postoji niz **razreda omotača** (engl. ***wrapper classes***) koji omogućuju da se primitivni tipovi podataka tretiraju kao objekti. Kasnije u tečaju bit će prikazani primjeri gdje će biti nužno korištenje razreda omotača, na primjer, kod korištenja parametriziranih metoda i kolekcija. U sljedećoj tablici prikazani su primitivni tipovi podataka i pripadajući razredi omotači.

|  |  |
| --- | --- |
| **Primitivni tip podataka** | **Razred omotač** |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

Primitivne tipove podataka i njihove razrede omotače moguće je miješati pa je tako sljedeća naredba valjana:

|  |
| --- |
| **int** broj = **new** Integer(3); |

Prilikom prevođenja događa se automatsko **pakiranje** (engl. ***autoboxing***) i **raspakiravanje** (engl. ***unboxing***) pri čemu se radi automatska pretvorba primitivnoga tipa u pripadajući objekt razreda omotača i obrnuto.

Razredi omotači također sadrže korisne atribute i metode koji olakšavaju rad s primitivnim tipovima podataka. Način korištenja nekih od njih prikazani su u nastavku, a za više informacija potrebno je pogledati dokument *Java* API [3].

|  |
| --- |
| // Najmanja moguca vrijednost tipa int, -2147483648  **int** najmanji\_integer = Integer.***MIN***\_***VALUE***;  // Najveca moguca vrijednost tipa long,  // 9223372036854775807  **long** najveci\_long = Long.***MAX***\_***VALUE***;  // Pretvorba stringa u broj  **int** tri = Integer.*parseInt*("3"); // 3  // Pretvorba broja u string  String s = Integer.*toString*(921); // "921"  // Racunanje maksimuma dva broja  **double** veci = Double.*max*(25.135, 24.827); // 25.135  // Racunanje minimuma dva broja  **double** manji = Double.*min*(25.135, 24.827); // 24.827 |

* 1. Polja i stringovi

U praksi je često potrebno obrađivati velike količine podataka. U tom slučaju nije praktično spremati vrijednosti u individualne varijable primitivnih tipova podataka, već se podaci grupiraju u strukture podataka ili kolekcije.

Najjednostavnija struktura podataka jest **polje** (engl. ***array***) gdje su podaci spremljeni u niz (najčešće susjednih) memorijskih lokacija. Pri tome podaci imaju **zajedničko ime**, a individualnim lokacijama pristupa se putem pozitivnoga **numeričkog indeksa**, počevši od prvog elementa s indeksom **0** do zadnjeg elementa s indeksom **veličina - 1**. Važno je napomenuti da svi elementi polja **moraju biti istoga tipa**.

U *Javi* je moguće definirati polje bilo kojeg tipa podataka (primitivni tip ili referenca) na sljedeći način:

**tip[] imePolja = new tip[veličina]**

pri čemu se definira polje određenoga tipa i veličine. Vrijednosti elemenata polja **bit će inicijalizirane na nulu** prema tablici s početka poglavlja bez obzira na to gdje se u programu polje stvori. Elementi polja mogu se inicijalizirati na proizvoljne vrijednosti prilikom stvaranja polja na sljedeći način:

**tip[] imePolja = new tip[] {  
 vrijednost1, vrijednost2, vrijednost3  
 }**

pri čemu će biti stvoreno polje od triju elemenata s pripadajućim vrijednostima. Prethodnu naredbu moguće je napisati još kraće na sljedeći način:

**tip[] imePolja = {  
 vrijednost1, vrijednost2, vrijednost3  
 }**

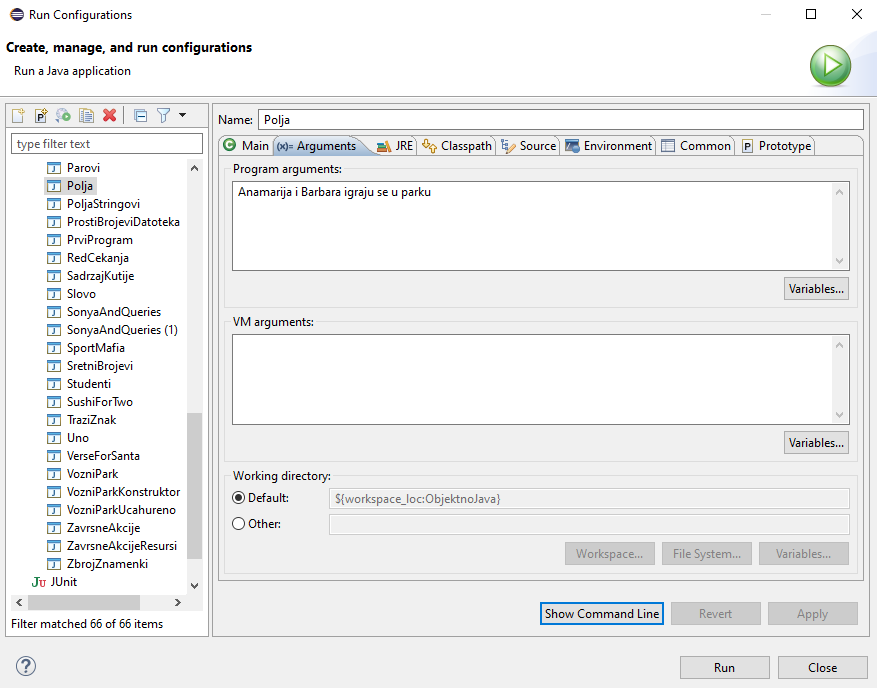
Pristupanje određenom elementu polja obavlja se indeksiranjem polja na sljedeći način:

**imePolja[indeks]**

Varijabla polja ***imePolja*** jest referenca preko koje se pristupa elementima polja. Time se olakšava rad s poljima jer se, na primjer, **ne kopira** cijelo polje prilikom slanja varijable kao argument u metode, već se šalje samo adresa memorijske lokacije polja. Kao što vrijedi i za ostale reference, polja su dinamički alocirana na gomili.

Svako polje u sebi ima pohranjenu informaciju o broju elemenata u atributu ***length***. Taj se atribut često koristi prilikom obilaska (iteriranja) polja po elementima, kao što je vidljivo u sljedećem primjeru.

Za pokretanje sljedećega primjera potrebno je postaviti ulazne parametre programa. Kao što je već bilo spomenuto u prošlom poglavlju, u razvojnom okruženju *Eclipse* to je moguće napraviti odabirom *Run* 🡪 ***Run Configurations*** u glavnoj izbornoj traci. Pritom se otvara novi dijaloški okvir u kojem je potrebno odabrati karticu ***Arguments***. Ulazne parametre treba upisati u tekstualni okvir ***Program arguments*** na način da argumenti budu međusobno odvojeni razmacima.



|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** Polja {  **public** **static** **void** main(String[] args) {  System.***out***.println("Uneseno je "  + args.length  + " rijeci.");  System.***out***.println("Prva rijec: " + args[0]);  System.***out***.println("Zadnja rijec: "  + args[args.length - 1]);  **int**[] polje = **new** **int**[] { 2, 4, 6 };  **double**[] novo\_polje = **new** **double**[polje.length];  novo\_polje[0] = polje[0];   novo\_polje[1] = polje[1];  novo\_polje[2] = polje[2];  System.***out***.println(novo\_polje[1]);  }  } |
| Ulaz:  Anamarija i Barbara igraju se u parku  Izlaz:  Uneseno je 7 rijeci.  Prva rijec: Anamarija  Zadnja rijec: parku  4.0 |

Jedan od najvažnijih tipova podataka jesu **nizovi znakova** koje koristimo za reprezentaciju riječi i rečenica unutar programa. Nizovi znakova najčešće su definirani poljem znakova (**char[]**), ali programski jezik *Java* nudi već spomenuti specijalni razred ***String*** koji u sebi interno sadrži polje znakova, ali uz to nudi mnoštvo korisnih funkcija za manipulaciju znakovima. Znakovi u razredu *String* također su pobrojani počevši od indeksa 0, ali im nije moguće direktno pristupati koristeći uglate zagrade, već se za to koristi pripadajuća metoda ***charAt***. Neke od funkcija razreda String pobrojane su u sljedećoj tablici, ali postoji i puno drugih koje su detaljno opisane u dokumentu *Java* API [3].

|  |  |
| --- | --- |
| **Funkcija** | **Opis** |
| char **charAt**(int index) | Vraća znak s danog indeksa. |
| boolean **endsWith**(String suffix) | Provjerava završava li String danim sufiksom. |
| boolean **equals**(String str) | Provjerava jednakost Stringova. |
| int **length**() | Vraća duljinu Stringa |
| String **toUpperCase**() | Vraća String kojem su sva mala slova pretvorena u velika. |
| String **toLowerCase**() | Vraća String kojem su sva velika slova pretvorena u mala. |
| String **valueOf**(int i) | Vraća tekstualnu reprezentaciju danoga broja. |

Sljedeći primjer prikazuje korištenje funkcija nabrojanih u prethodnoj tablici.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** Stringovi {  **public** **static** **void** main(String[] args) {  String rijec = "Otorinolaringologija";  System.***out***.println("Drugo slovo je: "  + rijec.charAt(1));  System.***out***.println("Duljina rijeci je "  + rijec.length());  System.***out***.println("Velikim slovima: "  + rijec.toUpperCase());  System.***out***.println("Malim slovima: "  + rijec.toLowerCase());  System.***out***.println("Zavrsava s -logija: "  + rijec.endsWith("logija"));  String prikaz5 = String.*valueOf*(5);  System.***out***.println(  "Tekstualni prikaz broja 5: "  + prikaz5);  }  } |
| Izlaz:  Drugo slovo je: t  Duljina rijeci je 20  Velikim slovima: OTORINOLARINGOLOGIJA  Malim slovima: otorinolaringologija  Zavrsava s -logija: true  Tekstualni prikaz broja 5: 5 |

U praksi je vrlo često potrebno **uspoređivati objekte tipa String**, ali treba biti vrlo oprezan kako ne bismo dobili neželjene rezultate. Naizgled je intuitivno koristiti **operator ==**, ali on uspoređuje samo adrese na kojima se nalaze reference što može dovesti do pogrešnog rezultata u slučaju dvaju različitih objekata koji imaju isti sadržaj. Umjesto toga, potrebno je koristiti funkciju **equals()**, kao što je prikazano na sljedećem primjeru.

Zanimljivo je primijetiti da naredba "rijec" == "rijec" ipak vraća **true**. Razlog tomu je što će se interno String "rijec" zapisati u jednu lokalnu varijablu te će se u navedenoj naredbi usporediti sama sa sobom.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** JednakostStringova {  **public** **static** **void** main(String[] args) {  String a = **new** String("rijec");  String b = **new** String("rijec");  System.***out***.println(a == b);  System.***out***.println(a.equals(b));  System.***out***.println("rijec" == "rijec");  }  } |
| Izlaz:  false  true  true |

Važno je napomenuti da je razred String **nepromjenjiv** (engl. ***immutable***) što znači da konkretnom objektu tipa String nije moguće mijenjati vrijednost znakova koje čuva u sebi. Sve metode razreda String koje služe za manipulaciju znakovnoga sadržaja zapravo vraćaju novi objekt koji će sadržavati rezultat konkretne promjene, poput prethodno spomenutih metoda toLowerCase() ili toUpperCase().

S razredom String treba biti oprezan jer lako može doći do **problema s performansama** uslijed brojnih manipulacija. Razlog tomu je što se puno procesorskoga vremena troši na kopiranje sadržaja kako bi se osigurala nepromjenjivost sadržaja svakog objekta. Za efikasnu manipulaciju znakovima koristi se razred **StringBuilder** koji također sadrži brojne korisne metode od kojih su neke navedene u sljedećoj tablici.

|  |  |
| --- | --- |
| **Funkcija** | **Opis** |
| StringBuilder **append**(char c) | Dodaje znak na kraj sadržaja. |
| StringBuilder **reverse**() | Preokreće tekstualni sadržaj. |
| String **toString**() | Vraća String u kojem se nalazi cjelokupni sadržaj objekta StringBuilder. |

Sljedeći primjer prikazuje korištenje razreda StringBuilder gdje se postepeno gradi tekst „abcdefgh“ koji se na kraju preokreće i ispisuje.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** PrimjerStringBuilder {  **public** **static** **void** main(String[] args) {  StringBuilder sb = **new** StringBuilder();  **for** (**char** c = 'a'; c <= 'h'; ++c)  sb.append(c);  System.***out***.println(sb.reverse().toString());  }  } |
| Izlaz:  hgfedcba |

* 1. Uvjetno izvođenje

Kao i mnogi drugi programski jezici, *Java* nudi upravljačke strukture kojima se postiže grananje i ponavljanje kôda, čime se povećava ekspresivnost jezika i omogućuje korisniku brži razvoj složenih programa.

* + 1. Upravljačka struktura *if* i *if-else*

Osnovna struktura grananja je ***if*** pri čemu se evaluira uvjet naveden u zagradama. Važno je napomenuti da navedeni uvjet mora rezultirati tipom podatka ***boolean***, u suprotnom će prevoditelj prijaviti pogrešku. U slučaju da je uvjet istinit, izvršavaju se naredbe iz tijela strukture ***if***. Sljedeći primjer ispisuje prvi ulazni argument te koristi strukturu ***if*** kako bi izbjegao slučaj kada programu nije predan ulazni argument:

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** Pozdrav {  **public** **static** **void** main(String[] args) {  **if** (args.length > 0) {  System.***out***.println("Bok, " + args[0] + ".");  }  }  } |
| Ulaz:  Ivana  Izlaz:  Bok, Ivana. |
| Ulaz:  *{Ništa}*  Izlaz:  *{Ništa}* |

Strukturu ***if*** moguće je proširiti dodatkom ***else***. U slučaju da predani uvjet nije istinit, izvršit će se naredbe iz ***else*** bloka.

Važno je napomenuti da u slučaju kada blok sadrži samo jednu naredbu, **nije potrebno navoditi vitičaste zagrade**. Prethodni primjer moguće je proširiti tako da ipak nešto ispiše u slučaju da nije predano ime:

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** Pozdrav {  **public** **static** **void** main(String[] args) {  **if** (args.length > 0)  System.***out***.println("Bok, " + args[0] + ".");  **else**  System.***out***.println("Nije predano ime.");  }  } |
| Ulaz:  *{Ništa}*  Izlaz:  Nije predano ime. |

U slučaju da je potrebno evaluirati nekoliko uvjeta, to je moguće postići proširivanjem postojeće strukture ***else******if*** konstrukcijom. Prilikom izvršavanja programa uvjeti se slijedno ispituju te se izvršava prvi blok čiji je uvjet istinit. Ako nijedan uvjet nije zadovoljen, izvršavanje će se nastaviti u ***else***bloku ili, ako ***else*** dio nije naveden, na prvoj naredbi nakon cijele ***if******else******if*** konstrukcije. Prethodni primjer može se uz pomoć te konstrukcije nadopuniti da obradi slučaj kad je predano nekoliko imena.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** Pozdrav {  **public** **static** **void** main(String[] args) {  **if** (args.length > 1)  System.***out***.println("Trenutno pozdravljam"  + " samo jedno ime.");  **else** **if** (args.length == 0)  System.***out***.println("Nije predano ime.");  **else**  System.***out***.println("Bok, " + args[0] + ".");  }  } |
| Ulaz:  Ivana  Izlaz:  Bok, Ivana. |
| Ulaz:  *{Ništa}*  Izlaz:  Nije predano ime. |
| Ulaz:  Ivana Josip  Izlaz:  Trenutno pozdravljam samo jedno ime. |

* + 1. Ternarni uvjetni operator

Prethodno opisana upravljačka struktura ***if-else*** može se kraće pisati korištenjem **ternarnog uvjetnog operatora**. Osnovni oblik operatora je:

uvjet ? izraz1 : izraz2

pri čemu se izraz1 izvršava ako je uvjet istinit, inače se izvršava izraz2. Važno je napomenuti da se ternarni operator koristi pri dodjeljivanju vrijednosti varijablama ili kod prosljeđivanja argumenata metodama. Iz tog razloga dani izrazi **moraju vraćati vrijednost**, a te vrijednosti moraju biti **istoga tipa** (ili različitih tipova kod kojih je moguća automatska konverzija iz jednog tipa podatka u drugi).

U nastavku je prikazan primjer korištenja ternarnog operatora u kojem se računa minimum dva broja te se ispituje njegova parnost.

**Napomena**

Radi preglednosti prikazani primjeri ne pokrivaju sve slučajeve u kojima korisnik može definirati nevaljane argumente programa.

U praksi je preporučeno da se svi mogući slučajevi obrade kako bi spriječili neočekivane prekide izvođenja programa.

**Djeljivost brojeva**

Za ispitivanje djeljivosti brojeva koristi se modulo operator (npr. ***a % b***) koji vraća ostatak pri dijeljenju jednoga broja s drugim.

Kada je ostatak nula, može se zaključiti da je ***a*** dijeljiv s ***b***.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** MinimumDvaBroja {  **public** **static** **void** main(String[] args) {  **long** prviBroj = Long.*parseLong*(args[0]);  **long** drugiBroj = Long.*parseLong*(args[1]);  **long** manjiBroj = (prviBroj < drugiBroj) ?  prviBroj : drugiBroj;  System.***out***.println("Minimum je: " + manjiBroj);  System.***out***.println(manjiBroj + " je "  + (manjiBroj % 2 == 0 ? "paran" : "neparan")  + " broj.");  }  } |
| Ulaz:  57 22  Izlaz:  Minimum je: 22  22 je paran broj. |
| Ulaz:  -3 0  Izlaz:  Minimum je: -3  -3 je neparan broj. |

Ternarne operatore moguće je ugnijezditi, ali to u većini slučajeva **nije preporučeno** zbog uvelike smanjene čitljivosti kôda:

|  |
| --- |
| **int** x = 2, y = 3;  String result = x > y ? "x je veci od y" :  x < y ? "x je manji od y" :  x == y ? "x je jednak y" :  "Nema rezultata"; |

* + 1. Upravljačka struktura *switch*

Čest je slučaj da korisnik želi ispitati vrijednost određenog izraza. To je moguće napraviti korištenjem strukture *if*, ali to vrlo brzo postaje nepregledno kod većeg broja uvjeta. U tu svrhu najbolje je koristiti upravljačku strukturu ***switch*** koja ima oblik:

**switch** (izraz) {  
 **case** konstanta1:  
 naredba1;  
 naredba2;  
 ..  
 **break**;  
 **case** konstanta2:  
 naredba5;  
 naredba6;  
 ..  
 **break**;  
 **default**:  
 naredba9;  
 naredba10;  
 ..  
}

Blok ***switch*** sastoji se od jednog ili više slučajeva koji se ispituju, a po potrebi (opcionalno) sadrži i ***default*** slučaj koji je analogan *else* bloku iz upravljačke strukture *if*. Vrijednost izraza uspoređuje se slijedno s navedenim vrijednostima (konstantama), a u slučaju da su vrijednosti jednake, izvršit će se odgovarajući blok naredbi.

Ključna riječ ***break*** označava završetak određenoga bloka naredbi pri čemu će program izaći iz strukture *switch* te nastaviti izvođenje na prvoj idućoj naredbi. Ključnu riječ ***break*** moguće je izostaviti, ali u tom slučaju izvršavanje programa nastavlja s idućim naredbama koje mogu pripadati bloku drugoga slučaja.

U nastavku je prikazan primjer korištenja upravljačke strukture ***switch***. Program očekuje dva cijela broja i odgovarajući operator, te ispisuje rezultat operacije na ekran. Program podržava operacije zbrajanja i oduzimanja, a u slučaju nepoznatog operatora ispisuje odgovarajuću poruku.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** IzracunajDvaBroja {  **public** **static** **void** main(String[] args) {  **int** prviBroj = Integer.*parseInt*(args[0]);  String operator = args[1];  **int** drugiBroj= Integer.*parseInt*(args[2]);  **int** rezultat;  **switch** (operator) {  **case** "+":  rezultat = prviBroj + drugiBroj;  **break**;  **case** "-":  rezultat = prviBroj - drugiBroj;  **break**;  **default**:  System.***out***.println("Nepoznati"  + " operator.");  **return**;  }  System.***out***.println(prviBroj  + " " + operator  + " " + drugiBroj  + " = " + rezultat);  }  } |
| Ulaz:  -5 + 8  Izlaz:  -5 + 8 = 3 |
| Ulaz:  -17 - -5  Izlaz:  -17 - -5 = -12 |
| Ulaz:  2 / 3  Izlaz:  Nepoznati operator. |

**Naredba return**

U ovom primjeru naredba **return** koristi se za prijevremeni izlazak iz programa.

Izostavljanje ključne riječi ***break*** moguće je iskoristiti za grupiranje više slučajeva. U nastavku je prikazan takav primjer koji za pročitano slovo određuje radi li se o samoglasniku ili suglasniku.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** Slovo {  **public** **static** **void** main(String[] args) {  **if** (args[0].length() != 1) {  System.***out***.println("Nije predano "  + "jedno slovo.");  **return**;  }  **switch** (args[0].toLowerCase()) {  **case** "a":  **case** "e":  **case** "i":  **case** "o":  **case** "u":  System.***out***.println("Samoglasnik.");  **break**;  **default**:  System.***out***.println("Suglasnik.");  **break**;  }  }  } |
| Ulaz:  A  Izlaz:  Samoglasnik. |
| Ulaz:  c  Izlaz:  Suglasnik. |
| Ulaz:  aBcDeFgH  Izlaz:  Nije predano jedno slovo. |

* 1. Petlje

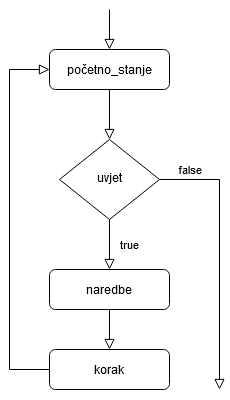
U programiranju je čest slučaj da je potrebno određene dijelove programa izvršiti više puta (npr. svakom zaposleniku povećati plaću). Iz tog razloga većina jezika nudi nekoliko struktura kojima se vrlo sažeto mogu napisati programi koji će višestruko obavljati određene akcije. Te strukture zovu se **petlje** (engl. ***loops***), a u nastavku se obrađuju tri vrste: ***for***, ***while*** i ***do-while***.

* + 1. Petlja *for*

Petlja ***for*** u praksi je najčešće korišten oblik petlje. Obično se rabi ako je unaprijed poznat broj potrebnih ponavljanja. Opći oblik *for* petlje jest:

**for (početno\_stanje; uvjet; korak) {  
 naredbe;  
 }**

Izraz **početno\_stanje** obično sadrži deklaraciju i inicijalizaciju **kontrolne varijable** koja se koristi unutar petlje. Tijelo petlje sadrži naredbe koje će se izvršavati sve dok je zadani uvjet zadovoljen. U slučaju da se radi o samo jednoj naredbi, vitičaste zagrade moguće je izostaviti. Nakon izvršavanja tijela petlje, izvršit će se izraz **korak** u kojem se najčešće mijenja kontrolna varijabla. **Dijagram toka** *for* petlje prikazan je na slici:



Izraze unutar zaglavlja for petlje nije nužno pisati pa je tako moguće oblikovati, na primjer, beskonačnu petlju na sljedeći način:

**for(;;);**

Ovakva petlja izvršavat će se vječno. Važno je uočiti da su znakovi točke sa zarezom obavezni. Također, u slučaju kad petlja nema pripadajućetijelo, potrebno je staviti točku sa zarezom na kraj. U slučaju kad tijelo petlje sadrži samo jednu naredbu, moguće je izostaviti vitičaste zagrade.

U nastavku je prikazan praktični primjer korištenja *for* petlje gdje se računa maksimum brojeva koji su dani putem polja argumenata komandne linije.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **public** **class** MaksimumBrojeva {  **public** **static** **void** main(String[] args) {  **int** maximum = Integer.***MIN\_VALUE***;  **for** (**int** i = 0; i < args.length; ++i)  maximum = Integer.*max*(maximum,  Integer.*parseInt*(args[i]));  **if** (args.length > 0)  System.***out***.println("Maksimum ucitanih"  + "brojeva je " + maximum);  }  } |
| Ulaz:  1 3 546 1 -23 4 -13486 208 8 1  Izlaz:  Maksimum ucitanih brojeva je 546 |

U *Javi* postoji drugi oblik *for* petlje oblika:

**for (deklaracija : kolekcija) {  
 naredbe;  
 }**

koji služi specifično za obilazak elemenata polja/kolekcije. Pritom se umjesto kontrolne indeksne varijable koristi varijabla koja će sadržavati vrijednost elementa kolekcije. Ovako definirana *for* petlja često se naziva i ***foreach*** petljom po uzoru na druge programske jezike koji ju definiraju tom specifičnom ključnom riječi.

Prednost ovakvog oblika *for* petlje jest u povećanoj čitljivosti čime se smanjuje mogućnost pogreške, ali nedostatak je u smanjenoj fleksibilnosti (npr. nije moguće obilaziti polje obrnutim redoslijedom). Petlju iz prethodnog primjera moguće je jednostavnije implementirati na sljedeći način:

|  |
| --- |
| **for** (String arg : args)  maximum = Integer.*max*(maximum,  Integer.*parseInt*(arg)); |

* + 1. Petlje while i do-while

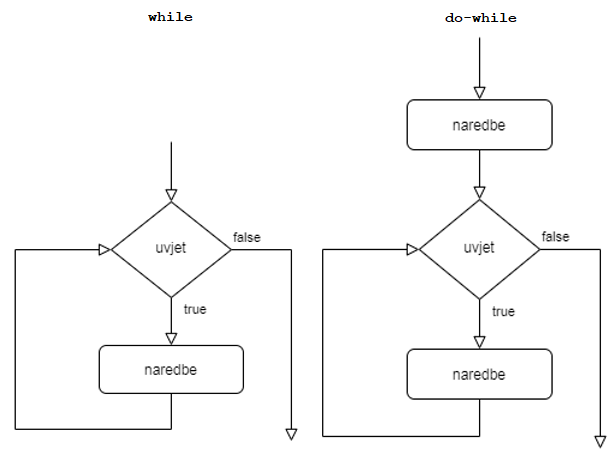
Petlja ***while*** razlikuje se od *for* samo po zaglavlju. Njen opći oblik je:

**while (uvjet) {  
 naredbe;  
}**

pri čemu se tijelo petlje izvršava sve dok zadani uvjet nije zadovoljen. Ova petlja je jednostavnija jer nema kontrolnu varijablu niti izraz prirasta u zaglavlju. Važno je primijetiti da se tijelo petlje neće izvršiti **nijednom** ako uvjet nije zadovoljen od početka. S druge strane, petlja ***do-while*** oblika:

**do {  
 naredbe;  
}** **while (uvjet);**

izvršit će naredbe prije provjeravanja uvjeta čime se osigurava minimalno jedan prolaz kroz tijelo petlje. Razlika između tih dviju petlji može se vizualno prikazati dijagramima toka:



U sljedećem primjeru prikazano je korištenje petlje *do-while*. Minimalnim preinakama moguće je postići istu funkcionalnost koja će koristiti običnu *while* petlju.

**Binarni zapis broja**

Razredi *Integer* i *Long* imaju ugrađenu metodu ***toBinaryString()*** koja daje jednak rezultat kao prikazani primjer, npr.

***Integer.toBinaryString(87)***

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  /\*\*  \* Program ucitava broj preko argumenata komandne  \* linije te ispisuje njegov zapis u binarnom  \* brojevnom sustavu.  \*/  **public** **class** BinarniZapis {  **public** **static** **void** main(String[] args) {  **int** broj = Integer.*parseInt*(args[0]);  String binarni = "";  **do** {  binarni = ((broj % 2 == 0) ? "0" : "1")  + binarni;  broj /= 2;  } **while** (broj > 0);  System.***out***.println(binarni);  }  } |
| Ulaz:  0  Izlaz:  0 |
| Ulaz:  32  Izlaz:  100000 |
| Ulaz:  87  Izlaz:  1010111 |

* + 1. Naredbe break i continue

Kao i mnogi drugi programski jezici, *Java* nudi naredbe ***break*** i ***continue*** koje služe za kontrolu toka izvođenja. Prethodno je naredba *break* prikazana u kontekstu upravljačke strukture ***switch*** pri čemu je služila za izlaz iz strukture. Jednako tako se kod petlji može koristiti za prekid ponavljanja. Nakon izvođenja naredbe ***break*** unutar petlje, program izlazi iz petlje te nastavlja s izvođenjem na prvoj naredbi iza tijela petlje.

Naredba ***continue*** preskače trenutnu iteraciju petlje te odmah prelazi na ispitivanje uvjeta i početak novog ciklusa petlje. U slučaju *for* petlje, izraz prirasta također će se izvršiti prije novog ispitivanja uvjeta. Naredba *continue* najčešće se koristi kad je u petlji potrebno preskočiti određene slučajeve.

U nastavku je prikazan primjer korištenja naredbi *break* i *continue*:

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  /\*\*  \* Program ucitava pozitivne brojeve iz argumenata  \* komandne linije. Za sve parne brojeve ispisuje se  \* zbroj znamenki, a u slucaju citanja negativnog  \* broja ispisuje se odgovarajuca poruka.  \*/  **public** **class** ZbrojZnamenki {  **public** **static** **void** main(String[] args) {  **for** (String arg : args) {  **int** broj = Integer.*parseInt*(arg);  **if** (broj < 0) {  System.***out***.println("Pogreska: Unesen je"  + " negativni broj " + broj + ".");  **break**;  }  **if** (broj % 2 != 0)  **continue**;  **int** zbroj = 0;  **for** (**int** tBroj = broj; tBroj > 0;  tBroj /= 10)  zbroj += tBroj % 10;  System.***out***.println("Zbroj znamenki "  + "parnog broja " + broj + " je "  + zbroj + ".");  }  }  } |
| Ulaz:  31 55 92 1204 1001 -341 7 21 80  Izlaz:  Zbroj znamenki parnog broja 92 je 11.  Zbroj znamenki parnog broja 1204 je 7.  Pogreska: Unesen je negativni broj -341. |

Naredbe *break* i *continue* utječu samo na **unutarnju (najviše ugniježđenu) petlju**, ali se njihova funkcionalnost može proširiti korištenjem **etiketa** (engl. ***label***). Etiketom se može označiti bilo koja naredba, ali se uglavnom koristi za označavanje petlji na sljedeći način:

**Oprez**

Prekomjerno korištenje etiketa nije preporučeno zbog smanjene čitljivosti kôda te predstavlja loš dizajn programa.

Etikete su se povijesno koristile uz naredbu ***goto*** u mnogim jezicima. Ključna riječ *goto* postoji u *Java* jeziku, ali se ne koristi.

Poznati nizozemski znanstvenik **Edsger W. Dijkstra** napisao je članak *„A Case against the GO TO statement“* u kojoj žestoko kritizira spomenutu naredbu.

**etiketa: while (uvjet) {  
 naredbe;  
}**

Unutar petlje može se izvršiti jedna od naredbi:

**break etiketa;  
continue etiketa;**

pri čemu će se naredba odnositi na petlju koja je označena s navedenom etiketom. Ovakav oblik naredbi *break* i *continue* koristi se u slučaju ugniježđenih petlji kako bi se utjecalo na vanjsku petlju iz tijela unutarnje petlje. Primjer korištenja naredbi *break* i *continue* s etiketama prikazan je u nastavku.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  /\*\*  \* Program trazi prvo pojavljivanje znaka 'a' u  \* rijecima dobivenih preko argumenata komandne  \* linije. Pritom znak 'a' ne smije biti ispred  \* znaka 'b'. Usput se ispisuju i rijeci koje ne  \* sadrze znak 'b'.  \*/  **public** **class** TraziZnak {  **public** **static** **void** main(String[] args) {  vanjska: **for** (String arg : args) {  **for** (**int** i = 0; i < arg.length(); ++i) {  **switch** (arg.charAt(i)) {  **case** 'a':  System.***out***.println(  "Pronasao znak 'a' u rijeci "  + arg + ".");  **break** vanjska;  **case** 'b':  **continue** vanjska;  }  }  System.***out***.println("Rijec " + arg  + " nema znak 'b'.");  }  }  } |
| Ulaz:  korito banana lubanja zec automobil pile  Izlaz:  Rijec korito nema znak 'b'.  Rijec zec nema znak 'b'.  Pronasao znak 'a' u rijeci automobil. |

* 1. Ulazni i izlazni tokovi

Svaki korisni računalni program komunicira s vanjskim svijetom (ljudima ili pak drugim programima) kako bi pročitao ulazne parametre, ispisao rezultate izvođenja ili prijavio grešku. Čitanje i pisanje obavlja se preko **ulaznih i izlaznih tokova podataka** (engl. ***input/output streams***). Izvori ulazno-izlaznih podataka su raznoliki, na primjer, tipkovnica, datoteka (engl. *file*), priključnica (engl. *socket*) i dr.

**Standardna greška**

U slučaju da se u programu treba prijaviti pogreška, preporuka je koristiti standardni tok podataka za greške (***System.err***).

Prethodni primjeri zbog jednostavnosti nisu to primjenjivali.

U programiranju postoje tri **standardna toka podataka** koji su unaprijed otvoreni i dostupni programima od početka njihova izvršavanja. To su **standardni ulaz** (engl. ***standard input, stdin***), **standardni izlaz** (engl. ***standard output, stdout***) i **standardna greška** (engl. ***standard error, stderr***). Za klasične konzolne aplikacije kojima se ovaj tečaj bavi, standardni tokovi bit će vezani za tekstualni terminal komandne linije. To znači da će podaci tih tokova biti u obliku niza znakova (*String*).

U *Javi* standardni tokovi podataka dostupni su kroz objekte ***System.in*** (standardni ulaz), ***System.out*** (standardni izlaz) i ***System.err*** (standardna greška).

Standardni izlaz i standardna greška zajedničkog su tipa ***PrintStream*** koji nudi mnoštvo metoda za ispisivanje. Najvažnije od njih su:

* **println** – ispisuje argument te na kraju prelazi u novi red
* **print** – ispisuje argument, ali ne prelazi u novi red
* **format**/**printf** – formatirano ispisivanje teksta, nalik metodi iz programskog jezika C

Primjer korištenja metoda za ispisivanje prikazan je u nastavku:

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  /\*\*  \* Program ucitava polumjer kruga iz argumenata  \* komandne linije te ispisuje opseg i povrsinu.  \*/  **public** **class** OpsegPovrsinaKruga {  **public** **static** **void** main(String[] args) {  **if** (args.length != 1) {  System.***err***.print("Greska, ");  System.***err***.println("nije predan polumjer.");  **return**;  }  **double** polumjer = Double.*parseDouble*(args[0]);  System.***out***.println("Podaci o krugu:");  System.***out***.printf("Polumjer: %.3f\n",  polumjer);  System.***out***.printf("Opseg: %f\n",  2.0 \* polumjer \* Math.***PI***);  System.***out***.format("Povrsina: %f",  polumjer \* polumjer \* Math.***PI***);  }  } |
| Ulaz:  27.823915  Izlaz:  Podaci o krugu:  Polumjer: 27.824  Opseg: 174.822814  Povrsina: 2432.127557 |
| Ulaz:  *{Ništa}*  Izlaz:  Greska, nije predan polumjer. |

Standardni ulaz je tipa ***InputStream*** koji nudi metode za čitanje podataka, ali te metode nisu praktične jer čitaju oktete (engl. *bytes*) koje je komplicirano interpretirati. Njih bi najprije trebalo pretvoriti u odgovarajuće *String* objekte te ih na kraju pretvoriti u npr. *double* u slučaju da na ulazu očekujemo decimalni broj. Iz tog razloga koriste se ***razredi omotači*** (engl. ***wrapper classes***) koji omataju *InputStream* s razredom više razine čime se proširuje funkcionalnost omotanoga razreda.

Jedan od mogućih omotača standardnog ulaza je razred ***Scanner***. To je jednostavni parser koji omogućuje čitanje primitivnih tipova podataka i stringova. *Scanner* nije dio standardne biblioteke programskoga jezika *Java* te ga zato najprije treba **uvesti** (engl. ***import***) kako bi bio vidljiv trenutnom paketu. Uvođenje se obavlja korištenjem naredbe ***import*** koja se navodi na početku izvorne datoteke (nakon definicije paketa) na sljedeći način:

**Uvođenje biblioteka**

Svi dosad korišteni razredi i metode pripadaju paketu ***java.lang*** koji je uvijek uključen.

Korištenjem znaka \* mogu se uključiti cijeli paketi, npr:

***import java.util.\*;***

Ne preporučuje se uključivanje cijelih paketa ako nije potrebno jer se time stvara „gužva“ u trenutnom imenskom prostoru te može doći do kolizije imena razreda, metoda i sl.

**import** java.util.Scanner;

Za korištenje toga razreda potrebno je stvoriti odgovarajući objekt te mu prilikom stvaranja proslijediti objekt standardnog ulaza:

Scanner sc = **new** Scanner(System.***in***);

Prilikom završetka izvođenja programa, standardni tokovi podataka se najčešće **automatski zatvaraju** čime se otpuštaju prethodno zauzeti resursi. Međutim, u slučaju korištenja razreda Scanner, taj tok podataka potrebno je eksplicitno zatvoriti korištenjem naredbe:

sc.close();

*Scanner* će ulazne podatke (u obliku niza znakova) podijeliti na tokene ili segmente (engl. ***tokens***) koristeći **bjeline** (razmaci, tabulatori i novi redovi) kao **graničnik** (engl. ***delimiter***).

Svaki od segmenata može se zasebno učitati koristeći niz dostupnih metoda poput ***nextInt***, ***nextShort***, ***nextString*** ili ***nextLine***, npr:

**long** broj = sc.nextLong();

Praktični primjer korištenja razreda *Scanner* prikazan je u nastavku:

**Prosti brojevi**

**Prost broj** (engl. ***prime number***) *M* je prirodni broj veći od 1 koji ima točno dva djelitelja (1 i *M*).

Na primjer, broj 7 je prost, ali broj 6 nije (jer je djeljiv s 1,2,3 i 6).

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **import** java.util.Scanner;  /\*\*  \* Program ucitava N te nakon toga N prirodnih  \* brojeva. Za svaki broj se provjerava je li prost  \* te se ispisuje odgovarajuca poruka.  \*/  **public** **class** ProstiBrojevi {  **public** **static** **void** main(String[] args) {  Scanner sc = **new** Scanner(System.***in***);  **int** N = sc.nextInt();  **for** (**int** i = 0; i < N; ++i) {  **long** broj = sc.nextLong();  **boolean** prost = **true**;    **if** (broj < 2)  prost = **false**;    **for** (**long** j = 2; j < broj; ++j) {  **if** (broj % j == 0) {  prost = **false**;  **break**;  }  }  System.***out***.println(broj  + (prost ? " je" : " nije")  + " prost.");  }  sc.close();  }  } |
| Ulaz:  6  1 3 12 37 87 9999907  Izlaz:  1 nije prost.  3 je prost.  12 nije prost.  37 je prost.  87 nije prost.  9999907 je prost. |

* + 1. Čitanje i pisanje datoteka

Osim standardnih tokova podataka, programi u *Javi* mogu čitati i pisati iz **datoteka** (engl. ***files***). Postoji mnoštvo razreda za čitanje i pisanje datoteka, ali će se radi jednostavnosti ovdje koristiti postojeći razredi ***PrintStream*** i ***Scanner***.

Programski jezik *Java* u paketu ***java.io*** sadrži brojne razrede za manipulaciju datotekama. Jedan od važnijih je razred ***File*** koji predstavlja apstraktnu reprezentaciju putanja do datoteka i direktorija. Preduvjet za čitanje i pisanje datoteka uz pomoć razreda *PrintStream* i *Scanner* je da im se prilikom stvaranja proslijedi objekt tipa ***File*** koji čuva lokaciju do tražene datoteke. To se može napraviti na sljedeći način:

Scanner sc = **new** Scanner(**new** File("dat1.txt"));

PrintStream ps = **new** PrintStream(**new** File("dat2.txt"));

Prilikom izvođenja prethodnih naredbi može se dogoditi da dane datoteke ne postoje. To je **iznimno ponašanje** te će se u tom slučaju podići **iznimka** java.io.FileNotFoundException. *Java* **forsira** da se iznimke na neki način obrade, inače dolazi do pogreške pri prevođenju. Zasad je dovoljno koristiti ključnu riječ **throws** u zaglavlju metode main (vidi primjer) pri čemu se objavljuje da ta metoda može uzrokovati navedenu iznimku.

Detaljne informacije o iznimkama i njihovom obrađivanju bit će navedene u zasebnom poglavlju.

U nastavku je prikazan modificirani program za provjeravanje prostih brojeva koji za čitanje i pisanje koristi datoteke umjesto standardnih tokova podataka.

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **import** java.io.File;  **import** java.io.FileNotFoundException;  **import** java.io.PrintStream;  **import** java.util.Scanner;  /\*\*  \* Program prima putanje do dvije datoteke preko  \* argumenata komandne linije. Program ucitava sve  \* brojeve iz prve datoteke te za svaki provjerava  \* je li prost. Odgovarajuce poruke ispisuju se u  \* drugu datoteku.  \*/  **public** **class** ProstiBrojeviDatoteka {  **public** **static** **void** main(String[] args)  **throws** FileNotFoundException {  Scanner sc = **new** Scanner(**new** File(args[0]));  PrintStream ps = **new** PrintStream(  **new** File(args[1]));  **while** (sc.hasNextLong()) {  **long** broj = sc.nextLong();  **boolean** prost = **true**;  **if** (broj < 2)  prost = **false**;  **for** (**long** j = 2; j < broj; ++j) {  **if** (broj % j == 0) {  prost = **false**;  **break**;  }  }  ps.println(broj  + (prost ? " je" : " nije")  + " prost.");  }  sc.close();  }  } |
| ulazna\_datoteka.txt:  1 3 12 37 87 9999907  izlazna\_datoteka.txt:  1 nije prost.  3 je prost.  12 nije prost.  37 je prost.  87 nije prost.  9999907 je prost. |

* 1. Pseudoslučajni brojevi

Programski jezik *Java*, kao i mnogu drugi jezici, nudi mogućnost generiranja **pseudoslučajnih brojeva** (engl. ***pseudorandom numbers***). Za generirane brojeve kaže se da su pseudoslučajni jer nisu u potpunosti slučajni već se generiraju preko formule koja vraća brojeve iz (približno) vjerojatnosnih distribucija kao što su uniformna ili Gaussova distribucija.

Za generiranje pseudoslučajnih brojeva u *Javi* najčešće se koristi razred **Random** iz paketa **java.util**. Sljedeća tablica prikazuje neke od funkcija tog razreda koje vraćaju različite primitivne tipove podataka čije su vrijednosti generirane iz **naizgled uniformne vjerojatnosne distribucije**, što znači da svaka vrijednost ima jednaku vjerojatnost pojavljivanja.

|  |  |
| --- | --- |
| **Funkcija** | **Opis** |
| boolean **nextBoolean**() | Vraća slučajnu boolean vrijednost true ili false. |
| double **nextDouble**() | Vraća slučajnu double vrijednost iz raspona [0.0, 1.0] |
| float **nextFloat**() | Vraća slučajnu float vrijednost iz raspona [0.0, 1.0] |
| int **nextInt**(int bound) | Vraća slučajnu int vrijednost iz raspona [0, bound> |

U sljedećem primjeru prikazano je korištenje razreda Random gdje program pokušava pogoditi broj koji je učitan iz argumenata komandne linije (u rasponu od 0 do 9).

|  |
| --- |
| **package** hr.unizg.srce.d470.opjj;  **import** java.util.Random;  **public** **class** PseudoRandom {  **public** **static** **void** main(String[] args) {  **int** trazeniBroj = Integer.*parseInt*(args[0]);  Random rand = **new** Random();  **int** pokusaj;  **for** (pokusaj = rand.nextInt(10);  pokusaj != trazeniBroj;  pokusaj = rand.nextInt(10)) {  System.***out***.println("Promasaj.. pokusaj = "  + pokusaj);  }  System.***out***.println("Pogodak! pokusaj = "  + pokusaj);  }  } |
| Ulaz:  9  Izlaz:  Promasaj.. pokusaj = 8  Promasaj.. pokusaj = 1  Promasaj.. pokusaj = 3  Promasaj.. pokusaj = 0  Promasaj.. pokusaj = 4  Promasaj.. pokusaj = 4  Promasaj.. pokusaj = 1  Pogodak! pokusaj = 9 |

* 1. Vježba: Osnove programskoga jezika *Java*

Ulazni podaci za sljedeće zadatke dani su u datoteci ***studenti.txt*** koja ima više redaka pri čemu svaki redak sadrži podatke o određenom studentu. Podaci su odvojeni razmacima te sadrže redom: **Prezime** (*String*), **Ime** (*String*), **Broj ECTS bodova** (*int*), **Prosjek ocjena** (*double*). Važno je napomenuti da su podaci **sortirani uzlazno** s obzirom na prezime.

1. Napišite program koji preko ulaznih argumenata komandne linije prima putanju do datoteke. Program treba pročitati datoteku te ispisati ukupnu sumu ECTS bodova.
2. Dopunite prethodni program da ispisuje podatke o odlikašima (studenti s prosjekom 5.0).
3. Dopunite prethodni program da ispisuje prezime koje se najčešće pojavljuje te srednju ocjenu svih studenata s tim prezimenom.
   1. Pitanja za ponavljanje: Osnove programskoga jezika *Java*
4. Nabrojite osnovne tipove podataka.
5. Što je sakupljač smeća u *Javi* i koju funkciju obavlja?
6. Čemu služe razredi omotači primitivnih tipova podataka? Koja je prednost njihovog korištenja?
7. Kako se dobije informacija o veličini polja u *Javi*?
8. Nabrojite strukture za uvjetno izvođenje kôda.
9. Navedite prednosti i mane korištenja tzv. *foreach* petlje.
10. Koji razredi se koriste za čitanje i pisanje datoteka?
11. Razredi i objekti

Po završetku ovoga poglavlja polaznik će moći:

definirati vlastite razrede i stvarati objekte

koristiti različite modifikatore pristupa

pisati konstruktore za inicijalizaciju objekata

pisati preopterećene konstruktore i metode

definirati i koristiti statičke članove.

U ovom poglavlju definiraju se osnovni pojmovi vezani za **objektno orijentirano programiranje** (kraće **OOP**). To je stil razvoja programskoga kôda (programska paradigma) pri čemu se programi projektiraju kao skup **objekata** koji međusobno komuniciraju. Ovakav pristup je zbog svoje prirodne modularnosti idealan za razvoj većih projekata te je stoga vrlo popularan u industriji.

Središnji pojam u OOP-u je **razred** koji predstavlja predložak ili shematski plan (engl. *blueprint*) za stvaranje objekata. Primjeri razreda mogu biti auto, životinja, voće itd. **Objekti** predstavljaju konkretne primjerke (engl. *instances*) razreda, npr. pas, jež, zebra, banana, jabuka, itd.

Razredi sadrže **atribute** i **metode** čime se oblikuju svojstva i ponašanje objekata. Atributi mogu biti ime, boja, broj kotača itd. te se definiraju kao varijable unutar razreda. Metode sadrže logiku za manipuliranje objektima te se definiraju kao funkcije. Primjeri metoda mogu biti ubrzaj(), promijeniBoju() itd.

1. 1. Definiranje razreda i stvaranje objekata

U programskom jeziku *Java* razredi se definiraju korištenjem ključne riječi **class** nakon koje slijedi ime razreda te blok naredbi kojima se definiraju atributi i metode. U nastavku je prikazana definicija razreda Vozilo koji sadrži atribute boja, tezina, brojKotaca i brzina te dvije metode koje služe za mijenjanje brzine vozila.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** Vozilo {  String boja;  **double** tezina;  **int** brojKotaca;  **double** brzina;  **void** ubrzaj(**double** x) {  brzina += x;  }  **void** uspori(**double** x) {  brzina -= x;  }  } |

Sljedeći primjer prikazuje glavni program koji koristi razred Vozilo na način da stvori dva objekta (bicikl i brod) te im mijenja atribute i poziva metode.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** VozniPark {  **private** **static** Vozilo *bicikl*, *brod*;  **public** **static** **void** main(String[] args) {  *bicikl* = **new** Vozilo();  *bicikl*.boja = "crvena";  *bicikl*.brojKotaca = 2;  *bicikl*.tezina = 8.5;  *bicikl*.ubrzaj(3.5);  *bicikl*.uspori(1.0);  *brod* = **new** Vozilo();  *brod*.boja = "plava";  *brod*.tezina = 20000.0;  *brod*.ubrzaj(40.0);  System.***out***.println("Boja bicikla: "  + *bicikl*.boja);  System.***out***.println("Boja broda: " + *brod*.boja);  System.***out***.println("Broj kotaca bicikla: "  + *bicikl*.brojKotaca);  System.***out***.println("Broj kotaca broda: "  + *brod*.brojKotaca);  System.***out***.println("Trenutna brzina bicikla: "  + *bicikl*.brzina);  System.***out***.println("Trenutna brzina broda: "  + *brod*.brzina);  }  } |
| Izlaz:  Boja bicikla: crvena  Boja broda: plava  Broj kotaca bicikla: 2  Broj kotaca broda: 0  Trenutna brzina bicikla: 2.5  Trenutna brzina broda: 40.0 |

Objekti pojednih razreda stvaraju se korištenjem operatora **new**, kao što je vidljivo u prethodnom primjeru. Operator new alocirat će potrebnu memoriju na memorijskom prostoru zvanom **gomila** (engl. ***heap***). Alocirana memorija sadrži prostor za pohranjivanje vrijednosti svih atributa pa tako svaki objekt sadrži **vlastite kopije atributa**. Stoga će npr. brod i bicikl imati različite vrijednosti atributa brojKotaca.

Operator new vratit će **referencu** određenoga tipa koja sadrži adresu na stvoreni objekt. U prethodnom primjeru reference se spremaju u varijable bicikl i brod koje su tipa Vozilo.

Važno je napomenuti da će atributi stvorenih objekata biti inicijalizirani na **nulu** za primitivne tipove podataka ili **null** u slučaju da je atribut referenca. Vrijednost **null** predstavlja referencu koja ne sadrži adresu konkretnog objekta. Za takvu referencu kaže se da pokazuje na „ništa“.

Reference služe za pristupanje atributima i metodama određenog objekta. To se obavlja korištenjem **operatora točka** (.) čime se može pročitati ili promijeniti vrijednost atributa, pozvati metoda itd. U prethodnom primjeru prikazane su brojne situacije gdje se koristi spomenuti operator.

Prilikom pristupanja atributima i metodama treba biti oprezan da referenca nema vrijednost null jer će to izazvati grešku koja će prekinuti normalno izvođenje programa.

* + 1. Enumeracije

U programskom jeziku *Java* postoje specijalne vrste razreda **enumeracije** koje predstavljaju skupinu **nepromijenjivih** **vrijednosti**. Enumeracije se definiraju korištenjem ključne riječi **enum** nakon koje slijedi ime enumeracije te njeno tijelo. Sljedeći primjer prikazuje definiciju enumeracije koja predstavlja nekoliko različitih boja.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **enum** Boja {  ***CRVENA***, ***ZUTA***, ***PLAVA***  } |

Različite vrijednosti enumeracije odvojene su zarezima te predstavljaju nepromjenjive atribute (varijable). Enumeracije se najčešće koriste za definiranje različitih boja, karata, dana, mjeseca itd. U nastavku je prikazan primjer koji koristi prethodno definiranu enumeraciju.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** PaletaBoja {  **public** **static** **void** main(String[] args) {  Boja boja = Boja.***PLAVA***;  System.***out***.print("Moguce boje: ");  **for** (Boja b : Boja.*values*())  System.***out***.print(b + " ");  System.***out***.print("\nVarijabla sadrzi ");  **switch** (boja) {  **case** ***CRVENA***:  System.***out***.print("crvenu");  **break**;  **case** ***ZUTA***:  System.***out***.print("zutu");  **break**;  **case** ***PLAVA***:  System.***out***.print("plavu");  **break**;  }  System.***out***.println(" boju.");  }  } |
| Izlaz:  Moguce boje: CRVENA ZUTA PLAVA  Varijabla sadrzi plavu boju. |

Važno je napomenuti da **nije moguće** stvoriti novi objekt tipa Boja, već se koriste postojeći objekti spremljeni u elementima enumeracije. Njima se pristupa korištenjem **operatora točka** nad imenom razreda, ali im nije moguće promijeniti vrijednost jer su interno definirani kao javni, statički i nepromjenjivi atributi – više o tome u nastavku poglavlja.

Kao što je prikazano u prethodnom primjeru, moguće je elegantno provjeravati vrijednosti enumeracije korištenjem **switch** konstrukcije. Također, moguće je iterirati kroz sve vrijednosti enumeracije korištenjem **for** petlje.

* 1. Dijagrami razreda

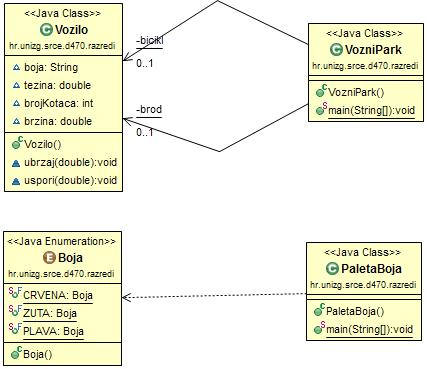
Programski sustavi u praksi su često jako veliki te sadrže milijune linija kôda koje su napisali deseci ili čak stotine inženjera. U tako velikim sustavima vrlo je važna **dokumentacija** koja opisuje strukturu i ponašanje sustava. Bez dokumentacije inženjeri bi morali dešifrirati logiku iz kôda čime bi se njihov rad uvelike otežao i usporio.

Jedan od poznatih alata za dokumentiranje je **UML** (***Unified Modeling Language***) – jezik za modeliranje koji predstavlja standardizirani način vizualizacije dizajna sustava. UML nudi mogućnost crtanja brojnih strukturalnih, ponašajnih i interakcijskih **dijagrama** koji daju jasan pregled funkcionalnosti sustava.

U kontekstu objektno orijentiranoga dizajna, jedan od najčešće korištenih UML dijagrama jest **dijagram razreda** (engl. ***class diagram***) koji vizualno opisuje strukturu sustava prikazivanjem razreda te njihovih atributa i metoda. Pritom se vizualiziraju i veze između razreda kao što su **ovisnost**, **kompozicija** i **generalizacija**.

**Ovisnost** (engl. ***dependency***) predstavlja vezu gdje jedan razred koristi objekte drugoga razreda, na primjer preko argumenata ili povratne vrijednosti metoda. **Kompozicija** je veza gdje jedan razred sadrži atribut koji je objekt drugog razreda. **Generalizacija** se veže za koncept nasljeđivanja koji je obrađen u sljedećem poglavlju.

U nastavku je prikazan dijagram razreda prethodnih primjera, generiran korištenjem ***ObjectAid*** alata koji se može ugraditi u *Eclipse* razvojno okruženje [7]. Ovisnost je prikazana iscrtkanom strelicom, dok je kompozicija prikazana običnom strelicom s nazivom atributa te označenom količinom objekata. Količina ***0..1*** predstavlja nula (null) ili jedan objekt, dok količina ***0..\**** označava proizvoljan broj objekata (npr. polje objekata).



* 1. Enkapsulacija

**Enkapsulacija** ili **učahurivanje** (engl. ***encapsulation***) je jedan od glavnih koncepta objektno orijentirane paradigme. To je mehanizam **skrivanja atributa** pri čemu se pristup atributima dozvoljava isključivo preko metoda istoga razreda. Prednost enkapsulacije je u tome što razred zadržava **potpunu kontrolu** nad vrijednostima atributa. Time se lako može spriječiti postavljanje nevaljane vrijednosti atributa ili generalno pristupanje određenom atributu.

**Automatsko učahurivanje**

U razvojnom okruženju *Eclipse* moguće je automatski enkapsulirati atribute razreda desnim klikom na razred te odabirom ***Source*** 🡪 ***Generate Getters and Setters***...

Enkapsulacija se postiže deklariranjem **privatnih** **atributa** korištenjem ključne riječi **private** te definiranjem tzv. **javnih** ***getter/setter*** metoda koje služe za čitanje i pisanje vrijednosti atributa. Osim čitanja i pisanja, *getter/setter* metode mogu sadržavati dodatnu logiku za, na primjer, provjeravanje valjanosti nove vrijednosti atributa. U nastavku je prikazan modificirani razred s početka poglavlja pri čemu su svi atributi učahureni.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** VoziloUcahureno {  **private** String boja;  **private** **double** tezina;  **private** **int** brojKotaca;  **private** **double** brzina;  **void** ubrzaj(**double** x) {  setBrzina(getBrzina() + x);  }  **void** uspori(**double** x) {  setBrzina(getBrzina() - x);  }  **public** String getBoja() {  **return** boja;  }  **public** **void** setBoja(String boja) {  **this**.boja = boja;  }  **public** **double** getTezina() {  **return** tezina;  }  **public** **void** setTezina(**double** tezina) {  **if** (tezina < 0.0)  **this**.tezina = 0.0;  **else**  **this**.tezina = tezina;  }  **public** **int** getBrojKotaca() {  **return** brojKotaca;  }  **public** **void** setBrojKotaca(**int** brojKotaca) {  **this**.brojKotaca = brojKotaca;  }  **public** **double** getBrzina() {  **return** brzina;  }  **public** **void** setBrzina(**double** brzina) {  **this**.brzina = brzina;  }  } |

U prethodnom primjeru moguće je uočiti ključnu riječ **this**. To je varijabla koja je uvijek dostupna unutar (ne-statičkih) metoda razreda te predstavlja referencu objekta nad kojim je ta metoda pozvana. U ovom slučaju koristi se za razlikovanje atributa i argumenta metode, s obzirom na to da imaju isto ime.

U nastavku je prikazan primjer korištenja prethodno definiranoga razreda.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** VozniParkUcahureno {  **private** **static** VoziloUcahureno *bicikl*, *brod*;  **public** **static** **void** main(String[] args) {  *bicikl* = **new** VoziloUcahureno();  *bicikl*.setBoja("crvena");  *bicikl*.setBrojKotaca(2);  *bicikl*.setTezina(8.5);  *bicikl*.ubrzaj(3.5);  *bicikl*.uspori(1.0);  *brod* = **new** VoziloUcahureno();  *brod*.setBoja("plava");  *brod*.setTezina(20000.0);  *brod*.ubrzaj(40.0);  System.***out***.println("Boja bicikla: "  + *bicikl*.getBoja());  System.***out***.println("Boja broda: "  + *brod*.getBoja());  System.***out***.println("Broj kotaca bicikla: "  + *bicikl*.getBrojKotaca());  System.***out***.println("Broj kotaca broda: "  + *brod*.getBrojKotaca());  System.***out***.println("Trenutna brzina bicikla: "  + *bicikl*.getBrzina());  System.***out***.println("Trenutna brzina broda: "  + *brod*.getBrzina());  }  } |
| Izlaz:  Boja bicikla: crvena  Boja broda: plava  Broj kotaca bicikla: 2  Broj kotaca broda: 0  Trenutna brzina bicikla: 2.5  Trenutna brzina broda: 40.0 |

* 1. Modifikatori pristupa

U prethodnim primjerima moguće je uočiti ključne riječi **public** i **private**. One predstavljaju neke od **modifikatora pristupa** kojima se definira dostupnost odnosno **vidljivost** **članova** (razreda, atributa, metoda itd.). Sljedeća tablica prikazuje različite modifikatore pristupa i njihove vidljivosti.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modifikator** | **Razred** | **Paket** | **Podrazred** | **Svijet** |
| public | ✓ | ✓ | ✓ | ✓ |
| protected | ✓ | ✓ | ✓ | 🗶 |
| (bez modifikatora) | ✓ | ✓ | 🗶 | 🗶 |
| private | ✓ | 🗶 | 🗶 | 🗶 |

Drugi stupac tablice prikazuje dostupnost unutar razreda gdje su definirani. Bez obzira na modifikator, svi članovi bit će vidljivi ostalim članovima unutar istog razreda. Treći stupac prikazuje vidljivost unutar paketa – svi članovi osim privatnih bit će vidljivi ostalim razredima unutar istog paketa. Četvrti stupac prikazuje vidljivost iz perspektive izvedenoga razreda (podrazreda), dok peti stupac prikazuje vidljivost iz perspektive svih ostalih razreda.

* 1. Konstruktori

Pri korištenju prethodno definiranoga razreda Vozilo, u glavnom programu je nakon stvaranja objekta (npr. bicikl ili brod) bilo potrebno postaviti sve atribute na odgovarajuće vrijednosti (npr. dva kotača za bicikl). U slučaju većega broja atributa taj posao postaje veoma naporan jer je za svaki stvoreni objekt potrebno pisati više linija kôda za inicijalizaciju tog objekta.

Kako bi inicijalizacija bila jednostavnija, u *Javi* postoje **konstruktori** koji predstavljaju posebnu vrstu metoda koje se automatski pozivaju prilikom stvaranja objekata te koje služe za njihovu inicijalizaciju.

Sljedeći primjer prikazuje isječak modificiranoga razreda Vozilo koji sadrži i odgovarajući konstruktor za inicijalizaciju objekata.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** VoziloKonstruktor {  **private** String boja;  **private** **double** tezina;  **private** **int** brojKotaca;  **private** **double** brzina;  **public** VoziloKonstruktor(  String boja,  **double** tezina,  **int** brojKotaca,  **double** brzina) {  **this**.boja = boja;  **this**.tezina = tezina;  **this**.brojKotaca = brojKotaca;  **this**.brzina = brzina;  }    ...  } |

Konstruktor se definira kao metoda bez specificirane povratne vrijednosti te istog imena kao i razred. Na sljedeći način moguće je stvoriti objekt korištenjem prethodno definiranoga konstruktora:

|  |
| --- |
| VoziloKonstruktor bicikl = **new**  VoziloKonstruktor("crvena", 8.5, 2, 0.0); |

Važno je napomenuti da svaki razred inicijalno sadrži **pretpostavljeni konstruktor** koji nema argumenata niti naredbe u svom tijelu (tzv. prazni konstruktor). U slučaju definiranja vlastitoga konstruktora, pretpostavljeni konstruktor bit će **uklonjen**, odnosno **zamijenjen**.

* 1. Preopterećenje (engl. *Overloading*)

Za jedan razred moguće je definirati više konstruktora koji se razlikuju po broju i/ili tipovima argumenata. Za takav konstruktor kaže se da je **preopterećen** (engl. ***overloaded***). Prilikom stvaranja objekata moguće je koristiti bilo koji od definiranih konstruktora, pod uvjetom da njihovi modifikatori pristupa to dozvoljavaju. Na primjer, definiranjem privatnoga konstruktora može se spriječiti stvaranje objekata nekoga razreda.

Također, konstruktori se međusobno mogu pozivati korištenjem ključne riječi **this** čime se postiže **ulančavanje konstruktora**. Prednost ulančavanja je u tome što se zajednička inicijalizacija može obaviti na jednom mjestu bez potrebe za dupliciranjem kôda. Sljedeći isječak kôda prikazuje primjer razreda s više definiranih konstruktora.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** VoziloPreopterecenje {  **private** String boja;  **private** **double** tezina;  **private** **int** brojKotaca;  **private** **double** brzina;  **public** VoziloPreopterecenje() {  **this**(**null**, 0.0, 0, 0.0);  }  **public** VoziloPreopterecenje(String boja,  **double** tezina,  **int** brojKotaca) {  **this**(boja, tezina, brojKotaca, 0.0);  }  **public** VoziloPreopterecenje(String boja,  **double** tezina,  **int** brojKotaca,  **double** brzina) {  **this**.boja = boja;  **this**.tezina = tezina;  **this**.brojKotaca = brojKotaca;  **this**.brzina = brzina;  }  ...  } |

Važno je napomenuti da poziv drugoga konstruktora, ako postoji, mora biti napisan **na samom početku (prva linija)** konstruktora, u suprotnom će prevodilac prijaviti grešku.

Preopterećenje se može primijeniti i na obične metode. Sljedeći primjer prikazuje preopterećenje metode mnozi(a, b) pri čemu svaka varijanta koristi različite tipove podataka argumenata. Ovakvo preopterećenje metoda predstavlja primjer **statičkoga polimorfizma**. To znači da se **prilikom prevođenja programa** za svaki poziv preopterećene metode određuje točna varijanta metode koja će se zapravo pozvati.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** Matematika {  **public** **int** zbroji(**int** a, **int** b) {  **return** a + b;  }  **public** **double** zbroji(**double** a, **double** b) {  **return** a + b;  }  **public** String zbroji(String a, String b) {  **return** a + b;  }  } |

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** Zbrajanje {  **public** **static** **void** main(String[] args) {  Matematika matematika = **new** Matematika();  System.***out***.println(matematika.zbroji(2, 3));  System.***out***.println(matematika.zbroji(10.5,  12.3));  System.***out***.println(matematika.zbroji("13",  "26"));  }  } |
| Izlaz:  5  22.8  1326 |

* 1. Statički članovi

U programskom jeziku *Java* postoji ključna riječ **static** koja se može koristiti prilikom definiranja razreda, atributa ili metoda.

Razred je moguće definirati kao statički samo ako je ugniježđen tj. definiran unutar tijela drugoga razreda. Iz tog razloga statički razredi bit će obrađeni kasnije, u poglavlju 5.6.

Statičkim članovima moguće je pristupiti **bez potrebe za objektom** razreda kojem ti članovi pripadaju. Razlog tome je što su statički članovi zajednički za sve objekte razreda. Statički atributi bit će alocirani u memoriji na samo jednom mjestu te će dijeliti istu vrijednost za sve objekte.

Statičkim atributima i metodama moguće je pristupiti korištenjem imena razreda na sljedeći način:

ImeRazreda.imeStatickogAtributa  
ImeRazreda.imeStatickeMetode()

Primjer statičke metode je i metoda **main** koja služi kao početna točka izvođenja svakoga programa.

Važno je napomenuti da u statičkim metodama nije moguće direktno pristupiti ne-statičkim atributima i metodama, ali obratno je moguće. U ne-statičkim metodama moguće je pristupiti statičkim atributima i metodama te u tom slučaju nije potrebno koristiti ime razreda.

Sljedeći primjer prikazuje korištenje statičkih atributa i metoda. Razred Kutija sadrži obični atribut volumen te statički atribut brojKutija koji služi kao brojač stvorenih kutija. U glavnom razredu BrojacKutija stvara se polje kutija te se inicijaliziraju i ispisuju informacije o svakoj kutiji.

Važno je primijetiti da neposredno nakon definiranja polja kutija atribut brojKutija ima vrijednost nula. Razlog tome je što nijedna kutija još nije alocirana u memoriji, jer polje u tom trenutku sadrži deset referenci koje imaju vrijednost null.

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** Kutija {  **private** **static** **int** *brojKutija*;  **private** **double** volumen;  **public** **static** **int** getBrojKutija() {  **return** *brojKutija*;  }  **public** **double** getVolumen() {  **return** volumen;  }  **public** **void** setVolumen(**double** volumen) {  **this**.volumen = volumen;  }  **public** Kutija(**double** volumen) {  ++*brojKutija*;  setVolumen(volumen);  }  } |

|  |
| --- |
| **package** hr.unizg.srce.d470.razredi;  **public** **class** BrojacKutija {  **public** **static** **void** ispisiBrojKutija() {  System.***out***.println("Broj kutija: " +  Kutija.*getBrojKutija*());  }  **public** **static** **void** main(String[] args) {  Kutija[] kutije = **new** Kutija[10];  *ispisiBrojKutija*();  **for** (**int** i = 0; i < kutije.length; ++i) {  kutije[i] = **new** Kutija(10 + i);  System.***out***.println("Volumen kutije "  + i + " je "  + kutije[i].getVolumen());  *ispisiBrojKutija*();  }  }  } |
| Izlaz:  Broj kutija: 0  Volumen kutije 0 je 10.0  Broj kutija: 1  Volumen kutije 1 je 11.0  Broj kutija: 2  Volumen kutije 2 je 12.0  Broj kutija: 3  Volumen kutije 3 je 13.0  Broj kutija: 4  Volumen kutije 4 je 14.0  Broj kutija: 5  Volumen kutije 5 je 15.0  Broj kutija: 6  Volumen kutije 6 je 16.0  Broj kutija: 7  Volumen kutije 7 je 17.0  Broj kutija: 8  Volumen kutije 8 je 18.0  Broj kutija: 9  Volumen kutije 9 je 19.0  Broj kutija: 10 |

Osim atributa i metoda, moguće je deklarirati i **statičke blokove** koji služe za inicijalizaciju statičkih atributa. Sljedeći isječak kôda prikazuje dva načina inicijalizacije – direktno te korištenjem **static** bloka. Moguće je definirati i više statičkih blokova koji će izvršavati slijedno prilikom inicijalizacije razreda.

|  |
| --- |
| **public** **class** Kutija {  **private** **static** **int** *brojKutija* = 5;  ... |

|  |
| --- |
| **public** **class** Kutija {  **private** **static** **int** *brojKutija*;    **static** {  *brojKutija* = 5;  }  ... |

* 1. Vježba: Razredi i objekti

**Prije rješavanja sljedećih zadataka pročitajte opis završne vježbe.**

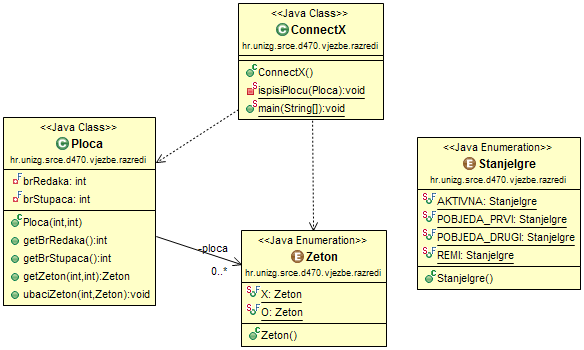
Ova vježba (kao i većina vježbi iz preostalih poglavlja) služi kao priprema za završnu vježbu u kojoj polaznici implementiraju igru ***ConnectX***.

1. Implementirajte enumeraciju **StanjeIgre** s članovima koji označavaju različita stanja igre. Igra može biti aktivna, remi ili pobijeđena od strane prvog ili drugog igrača.
2. Implementirajte enumeraciju **Zeton** s članovima **X** i **O** koji predstavljaju dvije različite boje žetona.
3. Implementirajte razred **Ploca** koji sadrži atribute **brRedaka** (broj redaka), **brStupaca** (broj stupaca) te dvodimenzionalno polje **ploca** u kojem će biti spremljeni žetoni.

* Dodajte konstruktor koji preko argumenata prima dimenzije ploče. Konstruktor mora zapisati primljene vrijednosti te inicijalizirati dvodimenzionalno polje s odgovarajućom veličinom.
* Dodajte *getter* metode za broj redaka i stupaca. Napišite metodu getZeton(redak,stupac) koja će vratiti element polja iz danog retka i stupca.
* Napišite metodu ubaciZeton(stupac,zeton) koja će ubaciti dani žeton na vrh određenoga stupca ploče.

1. Napišite razred **ConnectX** koji sadrži glavnu metodu main koja preko ulaznih parametara prima dimenzije ploče. Program treba stvoriti objekt ploče odgovarajućih dimenzija te učitavati iz standardnog ulaza indekse stupaca u koje je potrebno naizmjenično ubacivati žetone X i O. Nakon svakog ubacivanja potrebno je ispisati ploču na standardni izlaz. U slučaju da određeni redak i stupac ploče ne sadrži žeton, potrebno je ispisati znak točku. Program treba završiti u trenutku kad na standardnom ulazu više nema brojeva.

U nastavku je prikazan dijagram razreda rješenja koji može poslužiti kao pomoć prilikom rješavanja vježbe.



* 1. Pitanja za ponavljanje: Razredi i objekti

1. Detaljno opišite djelovanje sljedećeg isječka kôda:

|  |
| --- |
| Kutija kutija;  kutija = **new** Kutija(15); |

1. Što su enumeracije i koja im je glavna primjena?
2. Kako se postiže enkapsulacija atributa te zašto se koristi?
3. Navedite moguće modifikatore pristupa i njihove razine vidljivosti.
4. Čemu služe konstruktori te kako se definiraju?
5. Što znači da je neka metoda preopterećena?
6. Koja je razlika između običnih i statičkih atributa?

1. Nasljeđivanje

Po završetku ovoga poglavlja polaznik će moći:

pisati razrede koji koriste koncepte nasljeđivanja i nadjačavanja

koristiti ukalupljivanje tipova podataka

koristiti apstraktne razrede i sučelja

pisati ugniježđene, lokalne i anonimne razrede

definirati funkcijska sučelja te ih implementirati uporabom lambda izraza.

Jedan od važnijih pojmova u objektno orijentiranoj paradigmi jest **nasljeđivanje** (engl. ***inheritance***). Taj pojam predstavlja mogućnost novih razreda da naslijede postojeće čime se stvara hijerarhijska struktura razreda. Pritom se novi razred naziva **izvedeni razred** ili **podrazred** (engl. ***subclass***), dok se postojeći naziva **bazni razred** ili **nadrazred** (engl. ***superclass***). Uz nasljeđivanje se također vežu pojmovi **generalizacija** i **specijalizacija**. Bazni razred je generalizacija izvedenoga, dok je izvedeni razred specijalizacija baznoga razreda.

U svijetu postoje brojni primjeri hijerarhije razreda. Na primjer, slon je podrazred životinja, bicikl je podrazred vozila itd. U nastavku je obrađeno nasljeđivanje u kontekstu programskoga jezika *Java*.

1. 1. Hijerarhijska struktura razreda

Nasljeđivanje će biti objašnjeno na primjeru proizvoda koji se prodaju u dućanu. U nastavku je prikazan razred **Proizvod** koji sadrži nekoliko enkapsuliranih atributa te dvije metode: ispisiInformacije() i preracunajEuro().

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **class** Proizvod {  **private** **int** id;  **private** String naziv;  **private** **double** cijena;  **public** Proizvod(**int** id, String naziv,  **double** cijena) {  **this**.setId(id);  **this**.setNaziv(naziv);  **this**.setCijena(cijena);  }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getNaziv() {  **return** naziv;  }  **public** **void** setNaziv(String naziv) {  **this**.naziv = naziv;  }  **public** **double** getCijena() {  **return** cijena;  }  **public** **void** setCijena(**double** cijena) {  **this**.cijena = cijena;  }  **private** **double** preracunajEuro() {  **return** getCijena() / 7.5;  }  **public** **void** ispisiInformacije() {  System.***out***.println(getId() + ", " + getNaziv()  + ", " + preracunajEuro()  + " eura");  }  } |

Dućan prodaje dvije vrste proizvoda – **sudopere** i **pločice**. Obje vrste imaju zajedničke atribute (šifru proizvoda, naziv i cijenu), ali i vlastite specifične atribute. Sudoperi imaju tip (ugradbeni ili nadgradni), dok pločice imaju svoju dimenziju (duljinu i širinu).

Korištenjem nasljeđivanja može se lako postići odgovarajuća hijerarhijska struktura. Sljedeći primjer prikazuje razrede **Sudoper** i **Plocica** koji **nasljeđuju** razred Proizvod te dodaju vlastite atribute.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **enum** TipSudopera {  ***NADGRADNI***, ***UGRADBENI***  }  **public** **class** Sudoper **extends** Proizvod {  **private** TipSudopera tipSudopera;  **public** Sudoper(**int** id, String naziv,  **double** cijena,  TipSudopera tipSudopera) {  **super**(id, naziv, cijena);  **this**.setTipSudopera(tipSudopera);  }  **public** TipSudopera getTipSudopera() {  **return** tipSudopera;  }  **public** **void** setTipSudopera(  TipSudopera tipSudopera) {  **this**.tipSudopera = tipSudopera;  }  }  **public** **class** Plocica **extends** Proizvod {  **private** **double** duljina, sirina;  **public** Plocica(**int** id, String naziv,  **double** cijena, **double** duljina,  **double** sirina) {  **super**(id, naziv, cijena);  **this**.setDuljina(duljina);  **this**.setSirina(sirina);  }  **public** **double** getDuljina() {  **return** duljina;  }  **public** **void** setDuljina(**double** duljina) {  **this**.duljina = duljina;  }  **public** **double** getSirina() {  **return** sirina;  }  **public** **void** setSirina(**double** sirina) {  **this**.sirina = sirina;  }  } |

**Višestruko nasljeđivanje**

Razredi u *Javi* mogu naslijediti **najviše jedan razred**, za razliku od drugih jezika kao što je *C++* gdje je omogućeno **višestruko nasljeđivanje** prilikom čega razred može imati više direktnih (neposrednih) nadrazreda.

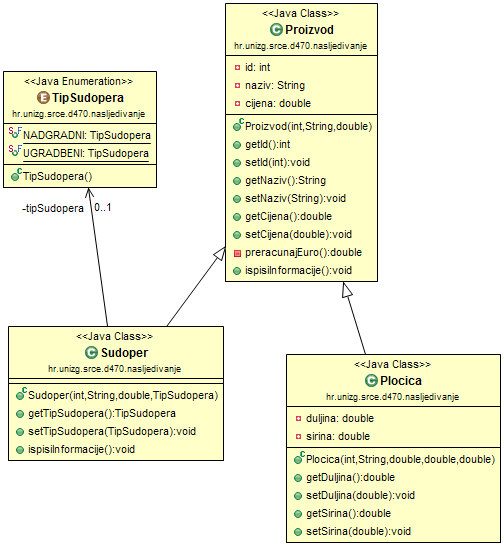
S druge strane, *Java* razredi mogu naslijediti (tj. implementirati) više **sučelja** – specijalnih tipova koji se obrađuju u nastavku poglavlja.

Nasljeđivanje se u *Javi* implementira korištenjem ključne riječi **extends** nakon koje slijedi ime baznoga razreda. Izvedeni razred nasljeđivanjem preuzima **sve atribute i metode** baznoga razreda pa će tako objekt izvedenoga razreda sadržavati vrijednosti svih atributa, uključujući one koji su definirani u baznom razredu.

Konstruktori baznoga razreda se ne nasljeđuju, ali se mogu pozivati iz konstruktora izvedenoga razreda korištenjem ključne riječi **super** koja predstavlja referencu na **objekt baznoga razreda**, slično kao ključna riječ **this** koja predstavlja referencu na **objekt trenutnoga razreda**.

Važno je napomenuti da će izvedeni razred imati pristup **samo vidljivim** atributima i metodama (npr. atributi definirani kao **public** ili **protected**). Tako će razred Sudoper morati koristiti javnu *getter* metodu getCijena() kako bi dohvatio vrijednost atributa cijena koji je naslijedio iz razreda Proizvod, jer je sami atribut definiran kao private. Također, razred Sudoper ne može pristupiti metodi preracunajEuro() jer je privatna.

Hijerarhija razreda može se elegantno vizualizirati korištenjem UML dijagrama razreda na sljedeći način:



* 1. Nadjačavanje (engl. *Overriding*)

**Automatsko nadjačavanje**

U razvojnom okruženju *Eclipse* moguće je automatski nadjačati metode baznoga razreda desnim klikom na izvedeni razred te odabirom ***Source*** 🡪 ***Override/Implement Methods***...

Izvedeni razredi mogu **redefinirati**, tj. **nadjačati** (engl. ***override***) metode baznoga razreda, pritom im mijenjajući ponašanje. U sljedećem primjeru razred Sudoper nadjačava metodu ispisiInformacije() na način da ispiše tip sudopera prije ostatka informacija.

|  |
| --- |
| **public** **class** Sudoper **extends** Proizvod {  **private** TipSudopera tipSudopera;  **public** Sudoper(**int** id, String naziv,  **double** cijena,  TipSudopera tipSudopera) {  **super**(id, naziv, cijena);  **this**.setTipSudopera(tipSudopera);  }  **public** TipSudopera getTipSudopera() {  **return** tipSudopera;  }  **public** **void** setTipSudopera(  TipSudopera tipSudopera) {  **this**.tipSudopera = tipSudopera;  }  @Override  **public** **void** ispisiInformacije() {  System.***out***.print(getTipSudopera()  + " sudoper, ");  **super**.ispisiInformacije();  }  } |

Prilikom nadjačavanja, nova verzija metode mora imati identičan **potpis**, što znači da treba imati isto ime, broj parametara te tipove parametara i povratne vrijednosti. U suprotnom će nova verzija biti definirana kao potpuno nova metoda bez ikakve poveznice s metodom iz baznog razreda. Ista situacija može se dogoditi i u slučaju da se naknadno promijeni potpis bazne metode. Kako bi se to izbjeglo, koristi se **anotacija** (engl. ***annotation***) **@Override** koja eksplicitno najavljuje da je sljedeća metoda nadjačana verzija neke metode baznoga razreda [8]. U slučaju da prevodilac ne može pronaći baznu verziju, prijavit će pogrešku.

Također, moguće je pozvati baznu verziju nadjačane metode korištenjem ključne riječi **super**, kao što je to slučaj u prethodnom primjeru. Za razliku od poziva baznoga konstruktora, poziv bazne metode **nije potrebno navoditi** na samom početku nadjačane metode.

Kako bi se prikazala djelotvornost nadjačavanja metoda, u sljedećem primjeru stvaraju se tri različita proizvoda te se ispisuju informacije o svakom od njih. Prilikom pozivanja metode ispisiInformacije() nad objektom razreda Sudoper, izvršava se nova verzija metode iz izvedenog razreda.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **class** Ducan {  **public** **static** **void** main(String[] args) {  Proizvod proizvod = **new** Proizvod(1, "papir",  30);  Sudoper sudoper = **new** Sudoper(2,  "GranitX", 900,  TipSudopera.***UGRADBENI***);  Plocica plocica = **new** Plocica(3,  "May Ceramics",  120, 60, 60);  proizvod.ispisiInformacije();  sudoper.ispisiInformacije();  plocica.ispisiInformacije();  }  } |
| Izlaz:  1, papir, 4.0 eura  UGRADBENI sudoper, 2, GranitX, 120.0 eura  3, May Ceramics, 16.0 eura |

Važno je napomenuti kako je moguće spriječiti nadjačavanje na način da se metoda u baznom razredu deklarira kao **nepromjenjiva**, tj. **final**. Pokušaj nadjačavanja takve metode rezultirat će pogreškom pri prevođenju. Na jednak način može se spriječiti i nasljeđivanje – deklariranjem željenoga razreda kao **final** **class**.

* 1. Dinamički polimorfizam

U *Javi* je referenci baznoga razreda moguće dodijeliti vrijednost objekta izvedenoga razreda. Tako je sljedeća linija kôda potpuno valjana:

Proizvod mojProizvod = **new** Sudoper(2,

"GranitX", 900,

TipSudopera.***UGRADBENI***);  
  
Razlog tome je što objekt razreda Sudoper između ostalog sadrži sve atribute i metode razreda Proizvod te se može tretirati kao takav. Ovakva pretvorba tipova podataka naziva se **ukalupljivanje** (engl. ***casting***).

Moguća je pretvorba i u suprotnom smjeru, ali pritom treba koristiti tzv. **eksplicitno ukalupljivanje** gdje se u zagradama navodi točan razred u koji se pretvara.

Sudoper mojSudoper = (Sudoper) mojProizvod;

Ovakav smjer ukalupljivanja **nije siguran** jer može dovesti do pogreške, ako mojProizvod zapravo sadrži objekt tipa Plocica. Više o tome bit će obrađeno u narednim poglavljima.

Zanimljiva situacija događa se prilikom pozivanja metoda nad ukalupljenim objektom, kao u sljedećoj naredbi:

mojProizvod.ispisiInformacije();

S obzirom na to da varijabla mojProizvod može sadržavati objekt bilo kojeg izvedenog razreda, *Javin* virtualni stroj će **prilikom izvođenja programa** odrediti točnu verziju metode koju treba pozvati na temelju stvarnoga tipa podataka koji se krije u danoj varijabli. U prethodnoj naredbi pozvat će se nadjačana verzija metode ispisiInformacije() koja je definirana unutar razreda Sudoper. Ovakav poziv metode predstavlja **dinamički polimorfizam**.

U nastavku je prikazan praktičan primjer korištenja ukalupljivanja i dinamičkoga polimorfizma.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **class** Nalog {  **public** **void** obradi(Proizvod[] proizvodi) {  }  }  **public** **class** PlatniNalog **extends** Nalog {  @Override  **public** **void** obradi(Proizvod[] proizvodi) {  **double** ukupnaCijena = 0.0;  **for** (Proizvod proizvod : proizvodi)  ukupnaCijena += proizvod.getCijena();  System.***out***.println("Ukupna cijena je: "  + ukupnaCijena);  }  }  **public** **class** Obrada {  **public** **static** **void** main(String[] args) {  Proizvod[] proizvodi = {  **new** Proizvod(1, "papir", 30),  **new** Sudoper(2, "GranitX", 900,  TipSudopera.***UGRADBENI***),  **new** Plocica(3, "May Ceramics", 120, 60, 60)  };  Nalog nalog = **new** PlatniNalog();  nalog.obradi(proizvodi);  }  } |
| Izlaz:  Ukupna cijena je: 1050.0 |

* + 1. Operator instanceof

U *Javi* postoji poseban binarni operator **instanceof** koji služi za ispitivanje pripadnosti objekta određenom razredu. Oblik korištenja operatora instanceof je sljedeći:

objekt **instanceof** Razred

Operator će vratiti tip **boolean** – true ako je objekt tipa Razred, inače false. Sljedeći primjer prikazuje korištenje operatora instanceof.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **class** OperatorInstanceOf {  **public** **static** **void** main(String[] args) {  Proizvod proizvod = **new** Plocica(1,  "May Ceramics", 120, 60, 60);  System.***out***.println(  proizvod **instanceof** Proizvod);  System.***out***.println(  proizvod **instanceof** Plocica);  System.***out***.println(  proizvod **instanceof** Sudoper);  }  } |
| Izlaz:  true  true  false |

Inače, ovaj operator se u praksi **rijetko koristi** te najčešće ukazuje na probleme u dizajnu objektno orijentiranoga sustava.

* 1. Bazni razred *Object*

U programskom jeziku *Java* svi razredi direktno ili indirektno nasljeđuju razred **Object** koji predstavlja **korijen hijerarhije** svih razreda. Taj razred sadrži nekoliko korisnih metoda od kojih su najvažnije **equals(Object o)** i **toString()**.

**Automatsko nadjačavanje metoda razreda Object**

U razvojnom okruženju *Eclipse* moguće je automatski nadjačati metode baznog razreda Object desnim klikom na vlastiti razred te odabirom ***Source*** 🡪 ***Generate hashCode() and equals() / toString()***

Metoda **equals(Object o)** predstavlja standardnu metodu za **uspoređivanje objekata**. Razred Object implementira ovu metodu na način da usporedi reference objekata operatorom == koji će jednostavno usporediti adrese na koje reference pokazuju. Takvo uspoređivanje može rezultirati **logičkom greškom** jer dva objekta istoga razreda s jednakim vrijednostima atributa mogu biti proglašeni različitima ako se nalaze na različitim adresama u memoriji. Iz tog razloga se ova metoda kod većine razreda nadjačava kako bi se omogućilo pravilno uspoređivanje. Jedan od najpoznatijih razreda koji implementiraju ovaj mehanizam je razred String.

Metoda **toString()** služi za pretvaranje objekta u tekstualni zapis (String) te se automatski poziva u slučajevima kad program očekuje objekt tipa String, ali mu je predan objekt nekoga drugog razreda. Najčešći slučaj toga može se vidjeti prilikom ispisivanja na standardni izlaz gdje metoda System.out.println() očekuje argument tipa String. Razred Object implementira metodu toString() tako da ispiše puno ime razreda te adresu u memoriji na kojoj se nalazi objekt.

U sljedećem primjeru razred Tocka nadjačava spomenute metode koje se zatim u glavnom razredu KoordinatniSustav pozivaju.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **class** Tocka {  **private** **int** x, y;  **public** Tocka(**int** x, **int** y) {  **this**.x = x;  **this**.y = y;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj)  **return** **true**;  **if** (!(obj **instanceof** Tocka))  **return** **false**;  Tocka other = (Tocka) obj;  **if** (x != other.x)  **return** **false**;  **if** (y != other.y)  **return** **false**;  **return** **true**;  }  @Override  **public** String toString() {  **return** "(" + x + ", " + y + ")";  }  }  **public** **class** KoordinatniSustav {  **public** **static** **void** main(String[] args) {  Tocka t1 = **new** Tocka(3, 5);  Tocka t2 = **new** Tocka(3, 5);  Tocka t3 = **new** Tocka(4, 5);  System.***out***.println(t1);  System.***out***.println(t1 == t2);  System.***out***.println(t1.equals(t2));  System.***out***.println(t1.equals(t3));  }  } |
| Izlaz:  (3, 5)  false  true  false |

* 1. Apstraktni razredi i sučelja

U *Javi* je moguće definirati **apstraktne razrede** (engl. ***abstract classes***) – posebne vrste razreda koje **nije moguće instancirati**. Apstraktni razredi služe kao bazni razred za porodicu izvedenih razreda koji imaju djelomično zajednička svojstva.

Idealan kandidat za apstraktni razred je Proizvod koji je definiran na početku poglavlja te predstavlja generički proizvod koji je logički gledano „apstraktan“. Apstraktni razred definira se ključnom riječju **abstract** na sljedeći način:

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **abstract** **class** ApstraktniProizvod {  **private** **int** id;  **private** String naziv;  **private** **double** cijena;  **public** ApstraktniProizvod(**int** id, String naziv,  **double** cijena) {  **this**.setId(id);  **this**.setNaziv(naziv);  **this**.setCijena(cijena);  }  ... |

Apstraktni razred može imati atribute, metode i konstruktore kao i svaki drugi razred, ali će pokušaj stvaranja objekta rezultirati pogreškom pri prevođenju. Apstraktni razred može imati i **apstraktne metode** – metode definirane bez tijela, kao što je prikazano u nastavku:

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **abstract** **class** ApstraktniProizvod {  **private** **int** id;  **private** String naziv;  **private** **double** cijena;  **public** ApstraktniProizvod(**int** id, String naziv,  **double** cijena) {  **this**.setId(id);  **this**.setNaziv(naziv);  **this**.setCijena(cijena);  }  **public** **abstract** **void** ispisiInformacije();  ...  } |

Razred koji sadrži barem jednu apstraktnu metodu **mora biti** definiran kao apstraktni razred. Time su izvedeni razredi prisiljeni nadjačati tj. implementirati apstraktne metode ako žele stvarati objekte.

Osim razreda, u *Javi* postoje postoje tipovi podataka zvani **sučelja** (engl. ***interfaces***) koji su vrlo slični apstraktnim razredima po tome što nije moguće stvarati njihove objekte te mogu sadržavati metode bez implementacije.

Apstraktni razredi najviše se koriste za definiranje hijerarhije usko povezanih razreda, na primjer, apstraktni razred Auto može služiti kao baza za porodicu različitih vrsta automobila. S druge strane, sučelja služe za definiranje **ponašanja objekata** koji nisu nužno usko povezani. Na primjer, *Java* sadrži sučelje Comparable kojime se oblikuje funkcija usporedbe dvaju objekata. Njime se može implementirati uspoređivanje različitih vrsta objekata kao što su automobili, životinje, kutije itd.

Glavna tehnička razlika je u tome što sučelja mogu sadržavati samo **javne statičke nepromjenjive atribute i javne metode**. Svi atributi i metode definirani unutar sučelja će **automatski (implicitno)** poprimiti spomenuta svojstva. Kao što je već spomenuto, razred može naslijediti samo jedan bazni razred, ali s druge strane može implementirati brojna sučelja.

Sljedeći primjer prikazuje korištenje sučelja gdje je prethodno definirani razred Nalog pretvoren u sučelje. Sučelje se definira ključnom riječju **interface**, dok se kod drugih razreda implementacija sučelja navodi korištenjem ključne riječi **implements**.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **interface** Nalog {  **public** **void** obradi(Proizvod[] proizvodi);  }  **public** **class** PlatniNalog **implements** Nalog {  @Override  **public** **void** obradi(Proizvod[] proizvodi) {  **double** ukupnaCijena = 0.0;  **for** (Proizvod proizvod : proizvodi)  ukupnaCijena += proizvod.getCijena();  System.***out***.println("Ukupna cijena je: "  + ukupnaCijena);  }  }  **public** **class** Obrada {  **public** **static** **void** main(String[] args) {  Proizvod[] proizvodi = {  **new** Proizvod(1, "papir", 30),  **new** Sudoper(2, "GranitX", 900,  TipSudopera.***UGRADBENI***),  **new** Plocica(3, "May Ceramics", 120, 60, 60)  };  Nalog nalog = **new** PlatniNalog();  nalog.obradi(proizvodi);  }  } |
| Izlaz:  Ukupna cijena je: 1050.0 |

* 1. Ugniježđeni, lokalni i anonimni razredi

U nastavku je opisano nekoliko različitih mehanizama uz pomoć kojih je moguće na koncizniji način naslijediti određeni bazni razred ili implementirati određeno sučelje. U prethodnom primjeru, sučelje Nalog moguće je drugačije implementirati korištenjem **ugniježđenih, lokalnih ili anonimnih razreda**, ili pak korištenjem **lambda izraza**.

Za određeni razred kaže se da je **ugniježđen** (engl. ***nested***) ako je definiran unutar tijela drugog razreda. Ugniježđene razrede moguće je deklarirati kao **static** te im dodijeliti različite modifikatore vidljivosti. Važno je napomenuti da ne-statički ugniježđeni razredi mogu pristupiti atributima razreda unutar kojeg su definirani, bez obzira na to jesu li privatni. Objekte statičkih ugniježđenih razreda moguće je stvarati bez potrebe za objektom vanjskoga razreda.

Ugniježđeni razredi najčešće se koriste za grupiranje logički povezanih razreda pri čemu se može postići dodatna enkapsulacija i čitljivost kôda. U nastavku je prepisan prethodni primjer korištenjem ugniježđenoga razreda.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **class** NestedObrada {  **private** **static** **class** UgnijezdeniPlatniNalog  **implements** Nalog {  @Override  **public** **void** obradi(Proizvod[] proz) {  **double** ukupnaCijena = 0.0;  **for** (Proizvod proizvod : proz)  ukupnaCijena += proizvod.getCijena();  System.***out***.println("Ukupna cijena je: "  + ukupnaCijena);  }  }  **public** **static** **void** main(String[] args) {  Proizvod[] proizvodi = {  **new** Proizvod(1, "papir", 30),  **new** Sudoper(2, "GranitX", 900,  TipSudopera.***UGRADBENI***),  **new** Plocica(3, "May Ceramics", 120, 60, 60)  };  Nalog nalog = **new** UgnijezdeniPlatniNalog();  nalog.obradi(proizvodi);  }  } |
| Izlaz:  Ukupna cijena je: 1050.0 |

**Lokalni razred** može se definirati unutar bilo kojeg bloka naredbi (npr. tijelo for petlje) te je vidljiv i valjan samo unutar zadanog bloka. Također, njegove metode imaju pristup svim varijablama bloka koji ga okružuje, uključujući i argumente metoda u slučaju kad je lokalni razred definiran unutar metode. U praksi se lokalni razredi ne koriste često. U nastavku je prepisan prethodni primjer korištenjem lokalnoga razreda.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **class** LokalnaObrada {  **public** **static** **void** main(String[] args) {  Proizvod[] proizvodi = {  **new** Proizvod(1, "papir", 30),  **new** Sudoper(2, "GranitX", 900,  TipSudopera.***UGRADBENI***),  **new** Plocica(3, "May Ceramics", 120, 60, 60)  };  **class** LokalniPlatniNalog **implements** Nalog {  @Override  **public** **void** obradi(Proizvod[] proz) {  **double** ukupnaCijena = 0.0;  **for** (Proizvod proizvod : proz)  ukupnaCijena += proizvod.getCijena();  System.***out***.println("Ukupna cijena je: "  + ukupnaCijena);  }  }  Nalog nalog = **new** LokalniPlatniNalog();  nalog.obradi(proizvodi);  }  } |
| Izlaz:  Ukupna cijena je: 1050.0 |

Definiciju lokalnoga razreda moguće je kraće napisati korištenjem **bezimenog** ili **anonimnog** **razreda**. Anonimni razredi definiraju se prilikom stvaranja objekta te se koriste u slučajevima kad je potrebno stvoriti samo jedan objekt traženoga razreda. Sljedeći primjer prikazuje korištenje anonimnih razreda.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **class** AnonimnaObrada {  **public** **static** **void** main(String[] args) {  Proizvod[] proizvodi = {  **new** Proizvod(1, "papir", 30),  **new** Sudoper(2, "GranitX", 900,  TipSudopera.***UGRADBENI***),  **new** Plocica(3, "May Ceramics", 120, 60, 60)  };  Nalog nalog = **new** Nalog() {  @Override  **public** **void** obradi(Proizvod[] proz) {  **double** ukupnaCijena = 0.0;  **for** (Proizvod proizvod : proz)  ukupnaCijena += proizvod.getCijena();  System.***out***.println("Ukupna cijena je: "  + ukupnaCijena);  }  };  nalog.obradi(proizvodi);  }  } |
| Izlaz:  Ukupna cijena je: 1050.0 |

U slučaju da je potrebno implementirati tzv. **funkcijsko sučelje** – sučelje koje sadrži **točno jednu** apstraktnu metodu, moguće je to napraviti korištenjem **lambda izraza** (engl. ***lambda expressions***).

Lambda izrazi su kratki blokovi kôda slični metodama koji primaju parametre za koje izvršavaju određene naredbe. Pojavljuju se u mnogim drugim programskim jezicima gdje se vrlo često koriste jer je uz pomoć njih moguće pisati vrlo kratak, koncizan i elegantan kôd. Mogući načini pisanja lambda izraza navedeni su u nastavku:

**parametar -> izraz  
( parametar1, parametar2 ) -> izraz  
( parametar1, parametar2 ) -> { blok kôda; }**

U najjednostavnijem slučaju lambda izraz sadrži jedan parametar za koji se izvršava naredba dana izrazom **izraz**. U slučaju više parametara, oni se navode unutar običnih zagrada. Također, moguće je pisati kompleksnije lambda izraze tako da se dodaju vitičaste zagrade koje mogu sadržavati više naredbi.

Tehnički gledano, lambda izrazi su zapravo **skraćeni zapisi anonimnih razreda ili sučelja** koji sadrže točno jednu apstraktnu metodu. Sljedeći primjer prikazuje korištenje lambda izraza kao zamjena za anonimne razrede.

|  |
| --- |
| **package** hr.unizg.srce.d470.nasljedivanje;  **public** **class** LambdaObrada {  **public** **static** **void** main(String[] args) {  Proizvod[] proizvodi = {  **new** Proizvod(1, "papir", 30),  **new** Sudoper(2, "GranitX", 900,  TipSudopera.***UGRADBENI***),  **new** Plocica(3, "May Ceramics", 120, 60, 60)  };  Nalog nalog = (Proizvod[] proz) -> {  **double** ukupnaCijena = 0.0;  **for** (Proizvod proizvod : proz)  ukupnaCijena += proizvod.getCijena();  System.***out***.println("Ukupna cijena je: "  + ukupnaCijena);  };  nalog.obradi(proizvodi);  }  } |
| Izlaz:  Ukupna cijena je: 1050.0 |

* 1. Vježba: Nasljeđivanje

Ova vježba nadograđuje prethodnu vježbu korištenjem novostečenoga znanja iz ovog poglavlja. U ovoj vježbi oblikuju se igrači i potezi igre ***ConnectX***.

1. U razredu Ploca nadjačajte metodu **toString()** tako da vrati tekstualni prikaz ploče. Iskoristite ovu metodu u glavnom programu ConnectX kako biste pojednostavili ispisivanje ploče.
2. Napišite apstraktni razred **Potez** koji predstavlja valjani potez u igri *ConnectX*, na primjer, ubacivanje žetona u određeni stupac ploče.

* Dodajte atribut **indeksStupca** koji označava indeks stupca nad kojim se potez treba odigrati. Implementirajte konstruktor koji inicijalizira indeksStupca te *getter* metodu koja će vratiti vrijednost atributa.
* Dodajte apstraktnu metodu **igraj(Ploca)** koja će predstavljati logiku odigravanja poteza nad danom pločom.

1. Implementirajte razred **PotezUbaci** koji nasljeđuje razred Potez.

* Dodajte atribut **zeton** koji predstavlja žeton koji je potrebno ubaciti. U konstruktoru inicijalizirajte žeton i indeks stupca te dodajte *getter* metodu za novi atribut.
* Nadjačajte metode igraj(Ploca) i toString().

1. Stvorite apstraktni razred **Igrac** s atributima **ime** i **zeton** koji predstavljaju ime igrača te tip žetona s kojima igra.

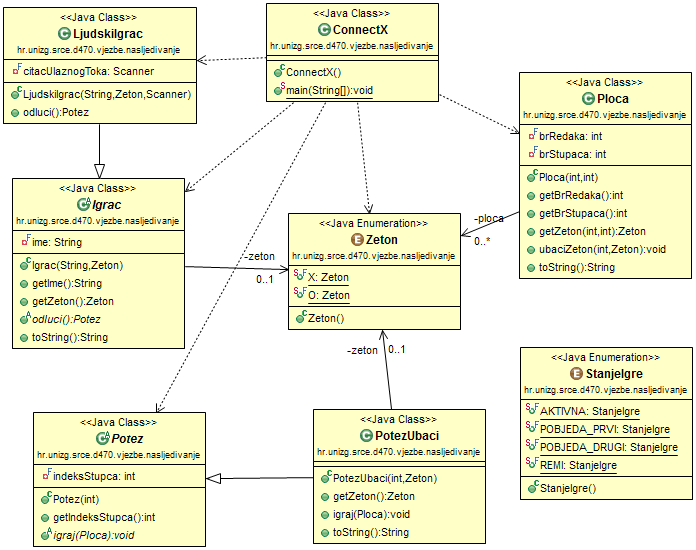
* Inicijalizirajte atribute u konstruktoru te dodajte odgovarajuće *getter* metode.
* Nadjačajte metodu **toString()** tako da vraća ime igrača.
* Dodajte apstraktnu metodu **odluci()** koja vraća potez koji je igrač odlučio odigrati.

1. Implementirajte razred **LjudskiIgrac** koji nasljeđuje razred Igrac.

* Dodajte atribut **citacUlaznogToka** tipa Scanner te ga inicijalizirajte u konstruktoru.
* Nadjačajte metodu odluci() na način da pročita naredbu oblika *„****ubaci X****“* gdje je X indeks stupca ploče. Metoda mora vratiti objekt tipa PotezUbaci s odgovarajućim vrijednostima. Zasad pretpostavite da će naredba uvijek biti valjanog oblika.

1. Modificirajte glavni razred ConnectX da dodatno učitava imena dvaju igrača sa standardnog ulaza. Inicijalizirajte igrače različitim vrstama žetona te u petlji naizmjenično pozivajte metodu odluci() svakog igrača. Odigrajte potez za koji se igrač odlučio te nakon toga ispišite informaciju o tome koji je igrač odigrao koji potez. Nakon toga ispišite novo stanje ploče.

U nastavku je prikazan dijagram razreda rješenja koji može poslužiti kao pomoć prilikom rješavanja vježbe.



* 1. Pitanja za ponavljanje: Nasljeđivanje

1. Koji su članovi baznih razreda vidljivi izvedenom razredu prilikom nasljeđivanja?
2. Kako se nadjačavaju metode baznoga razreda?
3. Što predstavlja ključna riječ super i kako se koristi?
4. Što je dinamički polimorfizam?
5. Nabrojite najpoznatije metode razreda Object te objasnite njihovo djelovanje.
6. Što su apstraktni razredi i čemu služe?
7. Što su funkcijska sučelja?

1. Iznimke

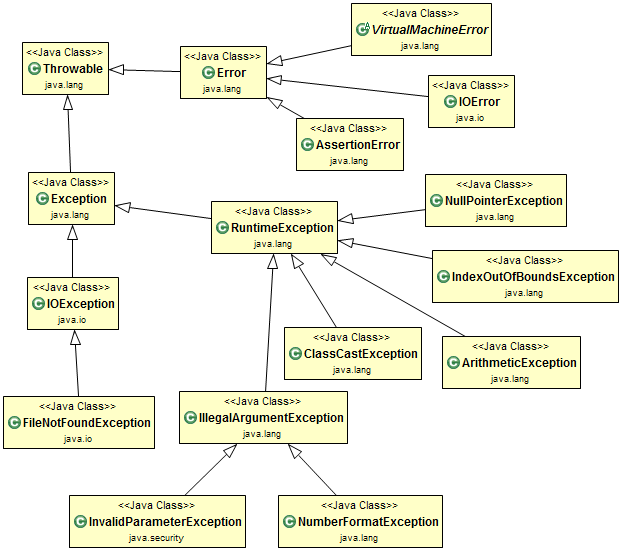
Po završetku ovoga poglavlja polaznik će moći:

definirati iznimke te navesti primjere u Javi

primijeniti koncepte obrađivanja i podizanja iznimki

definirati i koristiti vlastite iznimke.

Tijekom izvođenja programa mogu se dogoditi iznimne situacije koje najčešće prekidaju normalni tijek izvođenja, na primjer, dijeljenje s nulom, čitanje nepostojeće datoteke ili poziv metode nad referencom koja ima vrijednost null. Takve situacije nazivaju se **iznimke** (engl. ***exceptions***), a programi ih obično trebaju posebno obrađivati. Moderni programski jezici sadrže mehanizme za obradu iznimki (engl. *exception handling*). Programski jezik *Java* sadrži hijerarhiju razreda koji predstavljaju različite vrste iznimki, a najvažniji od njih prikazani su na sljedećem dijagramu razreda.



Na vrhu hijerarhije nalazi se bazni razred **Throwable** koji predstavlja iznimnu situaciju. Svaki Throwable objekt sadrži **poruku o grešci** te informacije o lokaciji gdje se greška dogodila (engl. ***stack trace***). Te informacije dostupne su kroz metode getMessage() i getStackTrace().

Razred Throwable nasljeđuju razredi Error i Exception. Razred Error predstavlja ozbiljnije iznimne situacije koje programi ne bi trebali obrađivati, dok Exception predstavlja sve ostale.

Iznimke se u *Javi* dijele na **provjeravane** (engl. ***checked***) i **neprovjeravane** (engl. ***unchecked***). Provjeravane iznimke moraju se eksplicitno obrađivati u kôdu, u suprotnom će prevodilac prijaviti grešku. Primjer takve iznimke jest **IOException** koja je već spomenuta u 3. poglavlju prilikom čega je bilo potrebno eksplicitno ju „obraditi“ koristeći ključnu riječ throws. Neprovjeravane iznimke nije potrebno eksplicitno obrađivati, a njih predstavljaju razredi **Error** i **RuntimeException** te svi razredi izvedeni iz njih.

U nastavku su prikazani primjeri najčešćih situacija kod kojih se događaju iznimke.

1. 1. Primjeri iznimki

Sljedeći primjer prikazuje jednu od najčešćih pogreški – pristup elementima polja koristeći indeks koji premašuje veličinu polja. U tom slučaju baca se neprovjeravana iznimka **ArrayIndexOutOfBoundsException**.

|  |
| --- |
| **package** hr.unizg.srce.d470.iznimke;  **public** **class** IznimkaPolje {  **public** **static** **void** main(String[] args) {  **int**[] polje = **new** **int**[] { 2, 3, 4 };  polje[5] = 5;  }  } |
| Izlaz:  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3 at hr.unizg.srce.d470.iznimke.Iznimke.main(Iznimke.java:7) |

U trenutku kad se dogodi iznimna situacija, odgovarajuća iznimka se baca, a uobičajeni tok programa se prekida. Iznimka se **propagira** „uvis“ te se traži najbliži blok koji ju može obraditi. U slučaju da se iznimka ne obradi unutar programa, naposljetku će ju obraditi **JVM** koji će prekinuti izvođenje programa te ispisati odgovarajuću poruku o grešci.

Sljedeći primjer prikazuje slučaj dijeljenja s nulom pri čemu se baca **ArithmeticException**. Važno je primijetiti da se iznimka ne bi dogodila u slučaju korištenja tipova podataka double, jer je u tom slučaju rezultat valjan te iznosi Double.POSITIVE\_INFINITY.

|  |
| --- |
| **package** hr.unizg.srce.d470.iznimke;  **public** **class** IznimkaDijeljenjeNulom {  **public** **static** **void** main(String[] args) {  **int** x = 3 / 0;  System.***out***.println(x);  }  } |
| Izlaz:  Exception in thread "main" java.lang.ArithmeticException: / by zero at hr.unizg.srce.d470.iznimke.IznimkaDijeljenjeNulom.main(IznimkaDijeljenjeNulom.java:6) |

* 1. Korištenje iznimki

Programski jezik *Java* omogućuje programeru različite mehanizme za rad s iznimkama. U nastavku su opisani načini korištenja iznimki.

* + 1. Proslijeđivanje iznimki

Ako u tijelu neke metode može doći do iznimne situacije koja rezultira bacanjem iznimke, tada je u deklaraciju metode moguće dodati popis iznimki koje se mogu dogoditi korištenjem ključne riječi **throws**:

<deklaracija\_metode> **throws** TipIznimke1,   
 TipIznimke2, ... {  
 <tijelo\_metode>  
 }

Proslijeđivanje iznimki najčešće se koristi kod provjeravanih iznimki gdje je nužno obraditi iznimku. Kada obrađivanje nije prikladno obaviti u trenutnoj metodi, tada se iznimka može proslijediti.

U nastavku je prikazan primjer proslijeđivanja iznimki prilikom otvaranja i čitanja datoteke. Metoda otvoriDatoteku može uzrokovati provjeravanu iznimku tipa FileNotFoundException prilikom stvaranja objekta tipa Scanner. Iznimka se ne obrađuje, već se proslijeđuje, pa zato metoda main koja poziva metodu otvoriDatoteku **neizravno uzrokuje istu iznimku**. Iz tog razloga potrebno je u metodi main obraditi ili, kao u prikazanom primjeru, proslijediti iznimku FileNotFoundException.

|  |
| --- |
| **package** hr.unizg.srce.d470.iznimke;  **import** java.io.File;  **import** java.io.FileNotFoundException;  **import** java.util.Scanner;  **public** **class** IznimnoCitanjeDatoteka {  **public** **static** Scanner otvoriDatoteku(String  imeDatoteke) **throws** FileNotFoundException {  **return** **new** Scanner(**new** File(imeDatoteke));  }  **public** **static** **void** main(String[] args) **throws**  FileNotFoundException {  Scanner sc = *otvoriDatoteku*(args[0]);  System.***out***.println(sc.nextLong());  sc.close();  }  } |
| Izlaz:  Exception in thread "main" java.io.FileNotFoundException: asd.txt (The system cannot find the file specified) at java.base/java.io.FileInputStream.open0(Native Method) at java.base/java.io.FileInputStream.open(FileInputStream.java:213) at java.base/java.io.FileInputStream.<init>(FileInputStream.java:155) at java.base/java.util.Scanner.<init>(Scanner.java:639) at hr.unizg.srce.d470.iznimke.IznimnoCitanjeDatoteka.otvoriDatoteku(IznimnoCitanjeDatoteka.java:10) at hr.unizg.srce.d470.iznimke.IznimnoCitanjeDatoteka.main(IznimnoCitanjeDatoteka.java:14) |

* + 1. Hvatanje iznimki

Kako bi se spriječio prekid izvođenja programa, bačene iznimke moguće je uhvatiti korištenjem try-catch blokova, čiji je opći oblik prikazan u nastavku.

**try** {   
 <naredbe\_koje\_mogu\_uzrokovati\_iznimku>  
 } **catch** (TipIznimke objektIznimke) {   
 <obrada\_iznimke>   
 }

Prethodno napisani blok catch uhvatit će iznimku razreda TipIznimke ili bilo kojeg njenog podrazreda. Objekt uhvaćene iznimke bit će spremljen u varijablu objektIznimke kako bi se mogao koristiti u nastavku. Preporuča se hvatanje što specifičnijih razreda iznimki kako bi se izbjeglo slučajno hvatanje neželjenih iznimki, jer je takve greške **vrlo teško** otkriti. Na primjer, hvatanje iznimki tipa Throwable rezultirat će hvatanjem svih vrsta iznimnih situacija, uključujući i one koje programer možda nije planirao uhvatiti.

Moguće je uhvatiti više vrsta iznimaka unutar jednog *catch* bloka korištenjem **operatora |** na sljedeći način:

**catch** (TipIznimke1 | TipIznimke2 objektIznimke)

Također, za jedan try blok moguće je navesti nekoliko catch blokova u nizu, prilikom čega se oni redom provjeravaju. Prevodilac će prijaviti grešku u slučaju kad je određeni catch blok teoretski nedohvatljiv (npr. hvatanje baznoga razreda Exception navedeno prije hvatanja podrazreda IllegalArgumentException).

Primjer hvatanja iznimaka prikazan je u nastavku. Program pokušava pročitati dva broja iz ulaznih parametara te ih podijeliti. Moguće iznimke se hvataju pri čemu se ispisuju odgovarajuće poruke.

|  |
| --- |
| **package** hr.unizg.srce.d470.iznimke;  **public** **class** HvatanjeIznimki {  **public** **static** **void** main(String[] args) {  **try** {  **int** a = Integer.*parseInt*(args[0]);  **int** b = Integer.*parseInt*(args[1]);  System.***out***.println(a / b);  } **catch** (ArrayIndexOutOfBoundsException ex) {  System.***err***.println("Ulaz ne sadrži " +  "barem dva parametra.");  } **catch** (NumberFormatException nfe) {  System.***err***.println("Ulazni parametar nije" +  "broj.");  } **catch** (ArithmeticException ae) {  System.***err***.println("Dijeljenje s nulom.");  }  }  } |
| Ulaz:  1  Izlaz:  Ulaz ne sadrži barem dva parametra. |
| Ulaz:  A B  Izlaz:  Ulazni parametar nije broj. |
| Ulaz:  10 0  Izlaz:  Dijeljenje s nulom. |
| Ulaz:  32 2  Izlaz:  16 |

* + 1. Bacanje iznimki

Osim hvatanja, iznimke je moguće i **baciti** korištenjem naredbe throw na sljedeći način:

**throw** <ThrowableObjekt>;

Naredba throw prekida normalno izvođenje programa te započinje potragu za najbližim blokom koji može obraditi danu iznimku. Naredba throw kao argument prima objekt tipa Throwable koji sadrži iznimku koju treba baciti (npr. objekt tipa IllegalArgumentException).

Sljedeći primjer računa zbroj **sretnih** **brojeva** koje čita iz ulaznih parametara programa. Inače, sretan broj je cijeli broj koji sadrži samo znamenke 4 ili 7 (npr. 47, 777, 4). U slučaju da ulazni broj nije sretan, baca se postojeća iznimka IllegalArgumentException sa specifičnom porukom greške.

|  |
| --- |
| **package** hr.unizg.srce.d470.iznimke;  **public** **class** SretniBrojevi {  **public** **static** **void** main(String[] args) {  **int** suma = 0;  **try** {  **for** (String arg : args) {  **int** a = Integer.*parseInt*(arg);  **for** (**int** i = a; i > 0; i /= 10)  **if** (i % 10 != 4 && i % 10 != 7)  **throw** **new**  IllegalArgumentException(a +  " nije sretan broj.");  suma += a;  }  System.***out***.println(suma);  } **catch** (NumberFormatException nfe) {  System.***err***.println("Ulazni parametar nije"  + "cijeli broj.");  } **catch** (IllegalArgumentException iae) {  System.***err***.println(iae.getMessage());  }  }  } |
| Ulaz:  744 44 37 7777  Izlaz:  37 nije sretan broj. |
| Ulaz:  A 47  Izlaz:  Ulazni parametar nije broj. |
| Ulaz:  4 47 74 7  Izlaz:  132 |

* + 1. Završne akcije

Ponekad je programima važno izvršiti određene naredbe prije završetka izvođenja, kao što je zatvaranje korištenih datoteka ili konekcija na bazu. Neoslobađanje korištenih resursa može dovesti do blokiranja resursa, na primjer, datoteka koja je ostala otvorena neće se naknadno moći ponovno otvoriti i čitati.

S obzirom na to da iznimke prekidaju normalno izvođenje programa, *Java* na raspolaganju ima blok **finally** koji se dodaje na postojeći try blok te čije će se naredbe **uvijek izvršiti** neovisno o potencijalnim iznimnim situacijama unutar try bloka. Time se osigurava pravovremeno oslobađanje resursa. Osnovni način korištenja bloka finally prikazan je u nastavku.

**try** {  
 <naredbe1>  
} **catch**(<iznimka>) {  
 <naredbe2>  
} **finally** {  
 <naredbe3>  
}

Blok finally izvršit će se nakon naredbi bloka try ili, u slučaju hvatanja iznimke, nakon naredbi bloka catch. Važno je napomenuti da blok catch nije nužno navoditi. Također, naredbe bloka finally izvršit će se i u slučaju naglog iskakanja iz try bloka korištenjem naredbi **return**, **continue** ili **break**. Primjer korištenja bloka finally prikazan je u nastavku.

**Standardni tokovi**

Ovaj primjer iznimno ispisuje poruke o greškama u standardni tok za izlaz umjesto standardni tok za grešku kako bi osigurao prikazivanje točnoga redoslijeda poruka na ekranu.

S obzirom na to da se radi o dvama različitim tokovima podataka, razvojno okruženje Eclipse prilikom spajanja tokova može promiješati poruke čime se smanjuje preglednost primjera.

|  |
| --- |
| **package** hr.unizg.srce.d470.iznimke;  **import** java.io.File;  **import** java.io.FileNotFoundException;  **import** java.util.NoSuchElementException;  **import** java.util.Scanner;  **public** **class** ZavrsneAkcije {  **public** **static** **void** main(String[] args) {  Scanner sc = **null**;  **for** (String imeDatoteke : args) {  **try** {  System.***out***.format("Otvaram datoteku " +  "%s.\n", imeDatoteke);  sc = **new** Scanner(**new** File(imeDatoteke));  System.***out***.format("Datoteka sadrzi " +  "broj: %d.\n",  sc.nextInt());  } **catch** (FileNotFoundException e) {  System.***out***.format("Datoteka %s ne " +  "postoji.\n",  imeDatoteke);  } **catch** (NoSuchElementException exception) {  System.***out***.format("Datoteka %s ne " +  "sadrži broj.\n",  imeDatoteke);  } **finally** {  System.***out***.println("Izvrsavam blok " +  "finally.");  **if** (sc != **null**)  sc.close();  }  }  }  } |
| brojA.txt:  5  brojB.txt:  12  brojC.txt:  C15  Ulaz:  brojA.txt brojB.txt brojC.txt brojD.txt  Izlaz:  Otvaram datoteku brojA.txt.  Datoteka sadrzi broj: 5.  Izvrsavam blok finally.  Otvaram datoteku brojB.txt.  Datoteka sadrzi broj: 12.  Izvrsavam blok finally.  Otvaram datoteku brojC.txt.  Datoteka brojC.txt ne sadrži broj.  Izvrsavam blok finally.  Otvaram datoteku brojD.txt.  Datoteka brojD.txt ne postoji.  Izvrsavam blok finally. |

Prethodni primjer učitava datoteke čija su imena predana preko ulaznih parametara programa. Program pokušava otvoriti svaku od datoteka te pročitati iz nje jedan cijeli broj. U ovom primjeru važno je uočiti da se blok finally izvršava neovisno o tome je li program uspješno otvorio datoteku i pročitao broj.

Osim bloka finally, postoji i drugi oblik bloka try (tzv. ***try-with-resources***) prikazan u nastavku:

**try** (<deklaracije\_resursa>) {  
 <naredbe>  
}

Ovaj blok kombinira funkcionalnosti blokova try i finally. Blok koristi deklarirane i inicijalizirane resurse koji će se u slučaju iznimne situacije automatski zatvoriti. Važno je napomenuti da deklarirani resursi moraju implementirati sučelje **AutoCloseable**. Neki od razreda koji implementiraju navedeno sučelje već su prethodno obrađeni, poput razreda InputStream, OutputStream, Scanner i PrintStream.

U nastavku je prikazan prethodni primjer, ali napisan korištenjem bloka *try-with-resources*.

|  |
| --- |
| **package** hr.unizg.srce.d470.iznimke;  **import** java.io.File;  **import** java.io.FileNotFoundException;  **import** java.util.NoSuchElementException;  **import** java.util.Scanner;  **public** **class** ZavrsneAkcijeResursi {  **public** **static** **void** main(String[] args) {  **for** (String imeDatoteke : args) {  **try** (Scanner sc =  **new** Scanner(**new** File(imeDatoteke))) {  System.***out***.format("Otvaram datoteku " +  "%s.\n", imeDatoteke);  sc = **new** Scanner(**new** File(imeDatoteke));  System.***out***.format("Datoteka sadrzi " +  "broj: %d.\n",  sc.nextInt());  } **catch** (FileNotFoundException e) {  System.***out***.format("Datoteka %s ne " +  "postoji.\n",  imeDatoteke);  } **catch** (NoSuchElementException exception) {  System.***out***.format("Datoteka %s ne " +  "sadrži broj.\n",  imeDatoteke);  }  }  }  } |
| brojA.txt:  5  brojB.txt:  12  brojC.txt:  C15  Ulaz:  brojA.txt brojB.txt brojC.txt brojD.txt  Izlaz:  Otvaram datoteku brojA.txt.  Datoteka sadrzi broj: 5.  Otvaram datoteku brojB.txt.  Datoteka sadrzi broj: 12.  Otvaram datoteku brojC.txt.  Datoteka brojC.txt ne sadrži broj.  Datoteka brojD.txt ne postoji. |

* 1. Stvaranje vlastitih iznimaka

Osim korištenja postojećih, moguće je definirati i vlastite iznimke **nasljeđivanjem postojećih razreda iznimaka**. Prednost korištenja vlastitih iznimaka jest mogućnost dodavanja atributa i metoda koje nisu dio standardnih *Java* iznimaka. Također, vlastite iznimke omogućuju vrlo specifično obrađivanje, čime se smanjuje mogućnost slučajnoga hvatanja neželjenih iznimaka.

U nastavku je prikazan primjer vlastite iznimke koja predstavlja iznimnu situaciju obrađivanja brojeva koji nisu prethodno opisani **sretni brojevi**.

|  |
| --- |
| **package** hr.unizg.srce.d470.iznimke;  **public** **class** NesretanBrojException **extends** Exception {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **public** NesretanBrojException() {  }  **public** NesretanBrojException(String message) {  **super**(message);  }  **public** NesretanBrojException(Throwable cause) {  **super**(cause);  }  **public** NesretanBrojException(String message,  Throwable cause) {  **super**(message, cause);  }  **public** NesretanBrojException(String message,  Throwable cause,  **boolean** enableSuppression,  **boolean** writeableStackTrace) {  **super**(message, cause, enableSuppression,  writeableStackTrace);  }  } |

Razred NesretanBrojException nasljeđuje razred Exception što znači da predstavlja **provjeravanu** iznimku. Također, ovaj razred implementira konstruktore baznoga razreda kako bi omogućio stvaranje objekata na jednak način.

Razvojno okruženje *Eclipse* prijavit će upozorenje u slučaju da nije definiran atribut **serialVersionUID** koji se koristi prilikom **serijalizacije** objekata koju bazni razred Throwable podržava implementiranjem sučelja **Serializable**. Za potrebe ovoga tečaja dovoljno je definirati te inicijalizirati spomenuti atribut s bilo kojom vrijednosti. Za više informacija o serijalizaciji objekata potrebno je pogledati dokument *Java* API [3] u kojem se nalazi specifikacija sučelja Serializable s detaljnim objašnjenjima.

U nastavku je prikazan prethodno opisani primjer računanja zbroja sretnih brojeva, ali korištenjem vlastite iznimke.

|  |
| --- |
| **package** hr.unizg.srce.d470.iznimke;  **public** **class** SretniBrojeviVlastitaIznimka {  **public** **static** **void** main(String[] args) {  **int** suma = 0;  **try** {  **for** (String arg : args) {  **int** a = Integer.*parseInt*(arg);  **for** (**int** i = a; i > 0; i /= 10)  **if** (i % 10 != 4 && i % 10 != 7)  **throw** **new** NesretanBrojException(a +  " nije sretan broj.");  suma += a;  }  System.***out***.println(suma);  } **catch** (NumberFormatException nfe) {  System.***err***.println("Ulazni parametar nije"  + "cijeli broj.");  } **catch** (NesretanBrojException ex) {  System.***err***.println(ex.getMessage());  }  }  } |
| Ulaz:  744 44 37 7777  Izlaz:  37 nije sretan broj. |
| Ulaz:  A 47  Izlaz:  Ulazni parametar nije broj. |
| Ulaz:  4 47 74 7  Izlaz:  132 |

* 1. Vježba: Iznimke

U ovoj vježbi potrebno je dodati obrađivanje iznimnih situacija koje se mogu dogoditi prilikom izvođenja vježbe iz prethodnog poglavlja.

1. U razredu LjudskiIgrac obradite slučajeve kad pročitani indeks pokušaja nije broj ili nije pozitivan broj. Za svaki od slučajeva bacite iznimku tipa **IllegalArgumentException** s različitim porukama pogreške.
2. U razredu LjudskiIgrac obradite slučaj kad ulazna naredba ne započinje riječju „ubaci“. U tom slučaju bacite iznimku **IllegalArgumentException** porukom „Nevaljana vrsta poteza.“.
3. Stvorite razred **PopunjenStupacException** koji će služiti kao neprovjeravana iznimka u slučaju pokušaja ubacivanja žetona u popunjeni stupac.

* Implementirajte pomoćnu metodu **getVisinaStupca(stupac)** u razredu Ploca koja će vratiti broj žetona u danom stupcu.
* U metodi ubaciZeton(stupac, zeton) razreda Ploca u slučaju da je traženi stupac popunjen bacite prethodno stvorenu iznimku (s odgovarajućom porukom).

1. U glavnom razredu ConnectX uvedite petlju koja će za određenog igrača pokušavati pročitati i odigrati potez sve dok potez nije uspješno odigran. Pritom uhvatite moguće iznimke te ispišite njihove poruke o grešci.
2. Isprobajte svaku od mogućih iznimnih situacija kako biste se uvjerili u ispravnost rješenja.
   1. Pitanja za ponavljanje: Iznimke
3. Što su iznimke? Navedite nekoliko primjera *Java* iznimaka.
4. Koja je razlika između provjeravanih i neprovjeravanih iznimaka? Koji razredi pripadaju neprovjeravanim iznimkama?
5. Čemu služi ključna riječ throws i kako se koristi?
6. Koji blokovi služe za hvatanje iznimaka?
7. Što radi naredba throw?
8. Kako osigurati izvođenje određenih naredbi neovisno o iznimkama koje se mogu dogoditi?
9. Kako se definiraju vlastite iznimke? Za što se koriste?
10. Parametrizacija kôda

Po završetku ovoga poglavlja polaznik će moći:

definirati generičko programiranje i prednosti koje donosi

opisati važnost očuvanja integriteta tipova podataka

koristiti tehnologiju Java Generics za parametriziranje metoda, razreda i sučelja.

U programiranju je često potrebno implementirati određenu funkcionalnost koja ima mogućnost rada s **različitim tipovima podataka** (npr. strukturiranje podataka ili njihovo ispisivanje). Takav stil pisanja programa naziva se **generičko programiranje** čija je glavna odlika pisanje algoritama koji su **generični** s obzirom na tip podataka s kojim rade. Time se izbjegava dupliciranje kôda jer bi u suprotnom bilo nužno više puta pisati isti algoritam za svaki podržani tip podataka.

U *Javi* se to može postići korištenjem i ukalupljivanjem baznoga razreda **Object**, ali to predstavlja lošu praksu jer se u tom slučaju riskira pojavljivanje iznimke **ClassCastException** uslijed pogrešnog ukalupljivanja. Sljedeći primjer ilustrira ovaj pristup i njegove mane.

|  |
| --- |
| **package** hr.unizg.srce.d470.parametrizacija;  **public** **class** NaruseniIntegritet {  **public** **static** Object ispisiVrati(Object o) {  System.***out***.println("#" + o + "#");  **return** o;  }  **public** **static** **void** main(String[] args) {  Integer a = 3;  String b = "broj";  a = (Integer) *ispisiVrati*(a);  b = (String) *ispisiVrati*(b);  a = (Integer) *ispisiVrati*(b);  }  } |
| Izlaz:  #3#  #broj#  #broj#  Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer at hr.unizg.srce.d470.parametrizacija.NaruseniIntegritet.main(NaruseniIntegritet.java:15) |

Prethodni primjer uredno će se prevesti, ali će prilikom pokretanja izazvati neočekivanu iznimku. Za ovakav program kaže se da ima narušen **integritet tipova podataka** (engl. ***type safety***).

Kako bi sačuvao integritet tipova, programski jezik *Java* nudi tehnologiju ***Java Generics*** za pisanje generičnih programa pri čemu se prilikom prevođenja programa provodi **statička provjera** valjanosti tipova podataka. Pogrešno korištenje tipova podataka u tehnologiji *Java Generics* rezultirat će pogreškom **pri prevođenju** programa.

Generički kôd piše se korištenjem **parametara** umjesto tipova podataka. Zato se za takav kôd kaže da je **parametriziran**. *Java Generics* omogućuje parametrizaciju metoda, razreda i sučelja.

1. 1. Parametrizacija metoda

U nastavku je prepisan prethodni primjer korištenjem tehnologije *Java Generics* čime se metoda ispisiVrati parametrizira.

|  |
| --- |
| **package** hr.unizg.srce.d470.parametrizacija;  **public** **class** ParametrizacijaMetoda {  **public** **static** <T> T ispisiVrati(T obj) {  System.***out***.println("#" + obj + "#");  **return** obj;  }  **public** **static** **void** main(String[] args) {  Integer a = 3;  String b = "broj";  a = *ispisiVrati*(a);  b = *ispisiVrati*(b);  // GRESKA: a = ispisiVrati(b);  }  } |
| Izlaz:  #3#  #broj# |

U metodi ispisiVrati prije povratnog tipa podataka deklarira se **parametar (T)** unutar izlomljenih zagrada (**<>**). Parametar može biti proizvoljnog imena, ali se preporuča korištenje jednoga velikog slova (npr. T,U,V,K,E). Deklarirani parametar može se koristiti u tijelu metode ili deklaraciji kao povratni tip podataka ili tip podataka argumenta metode. Moguće je unutar izlomljenih zagrada deklarirati više parametara **odvojenih zarezima**, na primjer, **<K,V>**.

U glavnoj metodi main prikazano je pozivanje parametrizirane metode pri čemu je važno primijetiti da nije potrebno eksplicitno ukalupljivanje tipova podataka. Naredba (a = *ispisiVrati*(b);) koja je u početnom primjeru izazvala iznimku, u ovom bi slučaju uzrokovala pogrešku pri prevođenju: *Type mismatch: cannot convert from String to Integer*.

Parametrizirana metoda može se pozivati s različitim tipovima podataka pa se za nju kaže da je **polimorfna** s obzirom na tip podataka. Ovo je primjer tzv. **parametarskog polimorfizma**.

Prilikom prevođenja parametriziranoga kôda radi se **brisanje tipova** (engl. ***type erasure***) pri čemu se parametri zamjenjuju odgovarajućim baznim tipom podataka te se na potrebna mjesta ubacuje eksplicitno ukalupljivanje. Pretpostavljeni bazni tip parametara je **Object** pa će nakon prevođenja prethodni primjer izgledati **jednako** kao početni. Glavna razlika je u dodatnim provjerama prevodilaca koje osiguravaju sigurnost tipova podataka.

***Java Generics* i sigurnost tipova podataka**

Tehnologija *Java Generics* ne osigurava u potpunosti sigurnost tipova podataka. Otkriveno je nekoliko rijetkih slučajeva u kojima je moguće zavarati statičku provjeru [9].

* 1. Parametrizacija razreda i sučelja

Sljedeći primjer definira parametrizirani razred koji predstavlja **par (ključ, vrijednost)** pri čemu oba člana mogu biti proizvoljnoga tipa.

|  |
| --- |
| **package** hr.unizg.srce.d470.parametrizacija;  **public** **class** Par<K, V> {  **private** K kljuc;  **private** V vrijednost;  **public** Par() {  **this**(**null**, **null**);  }  **public** Par(K kljuc, V vrijednost) {  **this**.kljuc = kljuc;  **this**.vrijednost = vrijednost;  }  **public** K getKljuc() {  **return** kljuc;  }  **public** **void** setKljuc(K kljuc) {  **this**.kljuc = kljuc;  }  **public** V getVrijednost() {  **return** vrijednost;  }  **public** **void** setVrijednost(V vrijednost) {  **this**.vrijednost = vrijednost;  }  @Override  **public** String toString() {  **return** "(" + kljuc + ":" + vrijednost + ")";  }  } |

Razredi se parametriziraju deklaracijom parametara unutar izlomljenih zagrada koje slijede nakon imena razreda. Svi atributi i metode unutar parametriziranog razreda mogu koristiti deklarirane parametre.

***Oprez!***

Deklaracija parametrizirane metode unutar parametriziranog razreda korištenjem istog imena parametra izazvat će **skrivanje parametra** unutar metode. Prevodilac će u takvim situacijama prijaviti upozorenje.

Postoje određena **ograničenja korištenja parametara** [10] koja treba imati na umu, a najvažnija od njih su:

* Parametri ne mogu biti primitivni tipovi podataka — umjesto njih koriste se razredi omotači (npr. Integer, Boolean itd.).
* Nije dozvoljeno deklarirati statičku varijablu tipa T.
* Nije dozvoljeno stvoriti novi objekt tipa T.
* Nije dozvoljeno definirati polje tipa T.
* Nije dozvoljeno definirati polje objekata parametriziranoga razreda.

Miješanje polja i parametriziranih tipova podataka predstavlja **lošu praksu** zbog mogućeg narušavanja integriteta tipova podataka. Za čuvanje većega broja objekata parametriziranih razreda preporuka je koristiti **kolekcije** koje će biti obrađene u sljedećem poglavlju.

Sljedeći primjer prikazuje korištenje prethodno definiranoga parametriziranog razreda. Glavni program čita parove vrijednosti iz ulaznih parametara te ih sprema u objekte tipa Par<K,V>. Svaki od objekata prikladno se ispisuje koristeći parametriziranu metodu obradiPar.

|  |
| --- |
| **package** hr.unizg.srce.d470.parametrizacija;  **public** **class** Parovi {  **private** **static** <K, V> **void** obradiPar(  Par<K, V> par) {  K kljuc = par.getKljuc();  V vrijednost = par.getVrijednost();  System.***out***.format("Par %s sadrzi kljuc \"%s\""  + " i vrijednost \"%d\".\n",  par, kljuc, vrijednost);  }  **public** **static** **void** main(String[] args) {  **for** (**int** i = 0; i + 1 < args.length; i += 2) {  Par<String, Integer> par = **new** Par<>();  par.setKljuc(args[i]);  par.setVrijednost(  Integer.*parseInt*(args[i + 1]));  *obradiPar*(par);  }  }  } |
| Ulaz:  Ivan 10 Ana 15 Petar 3  Izlaz:  Par (Ivan:10) sadrzi kljuc "Ivan" i vrijednost "10".  Par (Ana:15) sadrzi kljuc "Ana" i vrijednost "15".  Par (Petar:3) sadrzi kljuc "Petar" i vrijednost "3". |

Prethodni primjer prikazuje deklariranje i stvaranje objekta parametriziranoga razreda **Par<K,V>** sljedećom naredbom:

|  |
| --- |
| Par<String, Integer> par = **new** Par<>(); |

Deklaracija varijable parametriziranoga tipa mora definirati konkretne razrede parametara. U ovom slučaju razredi **String** i **Integer** predstavljat će parametre **K** i **V**. U naredbi za stvaranje objekta s desne strane nije potrebno ponavljati razrede već je moguće koristiti tzv. **dijamant (<>)**. Prevodilac će u tom slučaju shvatiti o kojem tipu objekta se radi.

Važno je napomenuti da se parametrizirani tipovi razlikuju s obzirom na konkretne razrede kojima se deklariraju pa će tako sljedeći isječak kôda uzrokovati grešku pri prevođenju (*Type mismatch: cannot convert from Par<Integer,String> to Par<String,Integer>*).

|  |
| --- |
| Par<String, Integer> par1 = **new** Par<>();  Par<Integer, String> par2 = **new** Par<>();  par1 = par2; |

* + 1. Parametrizirana funkcijska sučelja

Sučelja se mogu parametrizirati na jednak način kao i razredi. Odlični primjeri parametriziranih sučelja nalaze se unutar paketa **java.util.function** koji sadrži mnoštvo korisnih funkcijskih sučelja od kojih su najvažnija navedena u sljedećoj tablici.

|  |  |
| --- | --- |
| **Sučelje** | **Opis** |
| **Predicate<T>** | Operacija koja prima jedan argument i vraća boolean. |
| **BiPredicate<T,U>** | Operacija koja prima dva argumenta različitoga tipa i vraća boolean. |
| **UnaryOperator<T>** | Operacija koja prima jedan argument i vraća rezultat istoga tipa. |
| **BinaryOperator<T>** | Operacija koja prima dva argumenta istog tipa i vraća rezultat istoga tipa. |
| **Supplier<T>** | Funkcija koja ne prima argumente, ali vraća rezultat određenoga tipa. |
| **Consumer<T>** | Funkcija koja prima argument određenoga tipa i ne vraća rezultat. |
| **BiConsumer<T,U>** | Operacija koja prima dva argumenta različitoga tipa i ne vraća rezultat. |
| **Function<T,R>** | Funkcija koja prima jedan argument određenoga tipa i vraća rezultat drugoga tipa. |
| **BiFunction<T,U,R>** | Funkcija koja prima dva argumenta različitoga tipa i vraća rezultat trećega tipa. |

Sljedeći primjer prikazuje korištenje parametriziranih sučelja. Parametrizirana metoda obradi prima polje cijelih brojeva te ih uz pomoć fleksibilnih parametriziranih funkcijskih sučelja transformira u drugi tip te na određeni način agregira. Glavna metoda main prikazuje različite načine korištenja metode obradi – zbrajanje brojeva, traženje maksimuma, zbrajanje u String domeni itd.

|  |
| --- |
| **package** hr.unizg.srce.d470.parametrizacija;  **import** java.util.function.BinaryOperator;  **import** java.util.function.Function;  **public** **class** FunkcijskaSucelja {  **private** **static** <T> T obradi(Integer[] brojevi,  Function<Integer, T> funkcija,  BinaryOperator<T> agregator) {  T rezultat = **null**;  **for** (**int** i = 0; i < brojevi.length; ++i) {  **if** (i > 0)  rezultat = agregator.apply(rezultat,  funkcija.apply(brojevi[i]));  **else**  rezultat = funkcija.apply(brojevi[i]);  }  **return** rezultat;  }  **public** **static** **void** main(String[] args) {  Integer[] brojevi = **new** Integer[args.length];  **for** (**int** i = 0; i < args.length; ++i)  brojevi[i] = Integer.*parseInt*(args[i]);  System.***out***.println("Zbroj: " +  *obradi*(brojevi, x -> x,  (x, y) -> x + y));  System.***out***.println("Maksimum: " +  *obradi*(brojevi, x -> x,  (x, y) -> Math.*max*(x, y)));  System.***out***.println("Minimum: " +  *obradi*(brojevi, x -> x,  (x, y) -> Math.*min*(x, y)));  System.***out***.println("Broj neparnih brojeva: " +  *obradi*(brojevi,  x -> x % 2,  (x, y) -> x + y));  System.***out***.println("Zbroj (String): " +  *obradi*(brojevi,  x -> x.toString(),  (x, y) -> x + y));  System.***out***.println("Svi brojevi " +  jednoznamenkasti: " +  *obradi*(brojevi,  x -> x / 10 == 0,  (x, y) -> x && y));  }  } |
| Ulaz:  1 3 5 7 9 8 6 4 2 0  Izlaz:  Zbroj: 45  Maksimum: 9  Minimum: 0  Broj neparnih brojeva: 5  Zbroj (String): 1357986420  Svi brojevi jednoznamenkasti: true |

* 1. Ograničavanje parametara i bezimeni parametri

Dosadašnji način definiranja parametara pretpostavlja da će parametri biti izvedeni iz tipa **Object**, što omogućuje parametriziranim metodama, razredima i sučeljima rad s bilo kojim tipom podataka. Loša strana toga je što će prilikom rada s objektima parametarskoga tipa biti dostupni samo atributi i metode razreda Object. Iz tog razloga moguće je **ograničiti parametar** tako da mu se u deklaraciji specificira bazni razred ili sučelje koji služi kao **gornja granica**, na primjer:

public class Razred <T **extends** BazniRazred>

Ovako parametrizirani razred može raditi samo s objektima koji su tipa BazniRazred ili nekoga njegovog podrazreda. Parametrizirani kôd s ograničenim parametrom može raditi s objektima parametarskoga tipa kao da su objekti baznoga razreda ili sučelja kojim je parametar ograničen.

Moguće je zadati nekoliko ograničenja odvojenih znakom „**&**“, ali u tom slučaju smije biti naveden maksimalno jedan razred i to na početku.

public class Razred <T **extends** BazniRazred **&** Sucelje1 **&** Sucelje2>

Primjer ovakvog ograničavanja parametara prikazan je u nastavku. Prikazani program kopira i ispisuje podatke o različitim razredima proizvoda koristeći parametriziranu metodu ispisiVrati koja ima ograničeni parametar. Ovaj primjer koristi razrede definirane u 5. poglavlju.

|  |
| --- |
| **package** hr.unizg.srce.d470.parametrizacija;  **import** hr.unizg.srce.d470.nasljedivanje.Plocica;  **import** hr.unizg.srce.d470.nasljedivanje.Proizvod;  **import** hr.unizg.srce.d470.nasljedivanje.Sudoper;  **import** hr.unizg.srce.d470.nasljedivanje.TipSudopera;  **public** **class** Ogranicavanje {  **private** **static** <T **extends** Proizvod> T  ispisiVrati(T proizvod) {  proizvod.ispisiInformacije();  **return** proizvod;  }  **public** **static** **void** main(String[] args) {  Sudoper[] sudoperi = {  **new** Sudoper(1, "XGranit 40cm", 800,  TipSudopera.***UGRADBENI***),  **new** Sudoper(2, "XGranit 60cm", 1100,  TipSudopera.***UGRADBENI***)  };  Plocica[] plocice = {  **new** Plocica(3, "May Ceramics mramor", 220,  120, 60),  **new** Plocica(4, "May Ceramics beton", 160,  60, 60)  };  Sudoper[] noviSudoperi = **new**  Sudoper[sudoperi.length];  Plocica[] novePlocice = **new**  Plocica[plocice.length];  **for** (**int** i = 0; i < sudoperi.length; ++i)  noviSudoperi[i] = *ispisiVrati*(sudoperi[i]);  **for** (**int** i = 0; i < plocice.length; ++i)  novePlocice[i] = *ispisiVrati*(plocice[i]);  }  } |
| Izlaz:  UGRADBENI sudoper, 1, XGranit 40cm,  106.66666666666667 eura  UGRADBENI sudoper, 2, XGranit 60cm,  146.66666666666666 eura  3, May Ceramics mramor, 29.333333333333332 eura  4, May Ceramics beton, 21.333333333333332 eura |

Ograničavanje parametara moguće je koristiti i prilikom definiranja varijabli parametriziranih tipova. U nastavku je prikazan još jedan primjer ograničavanja parametara nakon čega slijedi objašnjenje ključnih dijelova programa.

|  |
| --- |
| **package** hr.unizg.srce.d470.parametrizacija;  **public** **class** SadrzajKutije {  **private** **static** **class** Kutija<T> {  **private** T sadrzaj;  **public** Kutija() {  **this**(**null**);  }  **public** Kutija(T sadrzaj) {  **this**.sadrzaj = sadrzaj;  }  **public** T getSadrzaj() {  **return** sadrzaj;  }  **public** **void** setSadrzaj(T sadrzaj) {  **this**.sadrzaj = sadrzaj;  }  }  **public** **static** **void** main(String[] args) {  Kutija<Integer> cijeliBroj = **new** Kutija<>();  cijeliBroj.setSadrzaj(3);  Kutija<Double> decimalniBroj = **new**  Kutija<>(Math.***PI***);  // GRESKA: cijeliBroj = decimalniBroj;  Kutija<? **extends** Number> kutija = **new**  Kutija<>();  kutija = cijeliBroj;  kutija = decimalniBroj;  // GRESKA: kutija.setSadrzaj(3.0);  Number n = kutija.getSadrzaj();  System.***out***.println("Kutija sadrzi: " + n);  Kutija<? **super** Integer> kutija2 = **new**  Kutija<>();  kutija2 = cijeliBroj;  // GRESKA: kutija2 = decimalniBroj;  kutija2.setSadrzaj(3);  Object o = kutija2.getSadrzaj();  System.***out***.println("Kutija2 sadrzi: " + o);  }  } |
| Izlaz:  Kutija sadrzi: 3.141592653589793  Kutija2 sadrzi: 3 |

Program koristi parametrizirani razred Kutija proizvoljnoga sadržaja. Kao što je već spomenuto, objekti istoga razreda i različitih parametara tretiraju se kao objekti različitih tipova podataka pa je tako nemoguće napraviti pridruživanje cijeliBroj = decimalniBroj.

Iz tog razloga moguće je definirati **objekt ograničenoga parametra** Kutija<? **extends** Number> kojemu je moguće pridružiti objekte istoga tipa s parametrima koji su razreda Number ili nekoga njegovog podrazreda. Važno je primijetiti da se umjesto imena parametra (npr. T) koristi znak „**?**“ (engl. ***wildcard***).

Iz ovakvog objekta moguće je pročitati sadržaj tipa **Number**, ali **nije moguće upisati sadržaj** jer prevodilac ne može garantirati o kojem se točno tipu sadržaja radi.

Osim gornje granice, pri definiranju objekta parametriziranoga razreda moguće je postaviti i **donju granicu** na parametar korištenjem ključne riječi **super**. U prethodnom primjeru na taj način definira se objekt tipa Kutija<? **super** Integer> kojemu je moguće pridružiti objekte istoga tipa s parametrima koji su razreda Integer ili nekoga njegovog baznog razreda. U ovakav objekt moguće je upisivati sadržaj razreda **Integer** ili nekoga njegovog podrazreda, ali je iz njega moguće pročitati samo sadržaj tipa **Object**.

Važno je napomenuti da **nije moguće** u isto vrijeme definirati i gornju i donju granicu određenoga parametra.

* 1. Vježba: Parametrizacija kôda

U ovoj vježbi nastavlja se implementacija igre ***ConnectX***.

1. Parametrizirajte razred Ploca tako da umjesto objekata tipa Zeton koristi generičke objekte tipa **T**. S obzirom na to da nije moguće stvoriti polje tipa T, zasad koristite bazni razred Object. U idućem poglavlju bit će prikazano kako izbjeći korištenje razreda Object pri pohranjivanju generičkih podataka.

* Preimenujte postojeće metode getZeton i ubaciZeton u **getElement** i **ubaciElement** kako bi dodatno generalizirali razred.
* Po potrebi promijenite ostale razrede kako bi na pravilan način koristili parametrizirani razred Ploca<Zeton>.

1. Stvorite razred **Igra** s atributima **ploca**, **igraci** i **brZetonaZaredom**. Ovaj razred predstavljat će partiju igre *ConnectX* koja je definirana igračima, dimenzijama ploče te brojem žetona koji igrači trebaju skupiti zaredom kako bi pobijedili.

* Inicijalizirajte atribute u konstruktoru te dodajte odgovarajuće *getter* metode. Ploču inicijalizirajte preko dobivenih brojeva redaka i stupaca.
* Dodajte metodu **odigraj()** koja će odigrati jednu cijelu partiju te vratiti završno stanje igre. Metoda u (zasad beskonačnoj) petlji treba naizmjenično igrati poteze obaju igrača. Iskoristite postojeću logiku iz glavne metode main.

1. Modificirajte glavni razred ConnectX tako da koristi novi razred Igra.

* Dodajte učitavanje dodatnog ulaznog parametra **brZetonaZaredom**.
* Umjesto postojeće petlje iskoristite objekt tipa Igra kako biste stvorili i pokrenuli igru.
* Na ekran ispišite različite poruke ovisno o završnom stanju igre.
  1. Pitanja za ponavljanje: Parametrizacija kôda

1. Što je generičko programiranje?
2. Zašto je važno sačuvati integritet tipova podataka?
3. Koja je prednost korištenja tehnologije *Java Generics* u odnosu na korištenje baznoga razreda Object?
4. Navedite nekoliko ograničenja korištenja parametara.
5. Navedite nekoliko primjera funkcijskih sučelja iz paketa ***java.util.function***.
6. Čemu služi ograničavanje parametara?
7. Kolekcije

Po završetku ovoga poglavlja polaznik će moći:

definirati osnovne strukture podataka i njihove složenosti

koristiti podatkovne strukture implementirane u razvojnom okviru Java Collections

primijeniti Java kolekcije u rješavanju praktičnih problema.

Računalni programi najčešće imaju zadatak obraditi velike količine podataka. Kako bi bili što efikasniji, programi organiziraju podatke u **apstraktne strukture podataka** (engl. ***abstract data structures***) koje najčešće podržavaju osnovne operacije nad skupom podatkovnih elemenata:

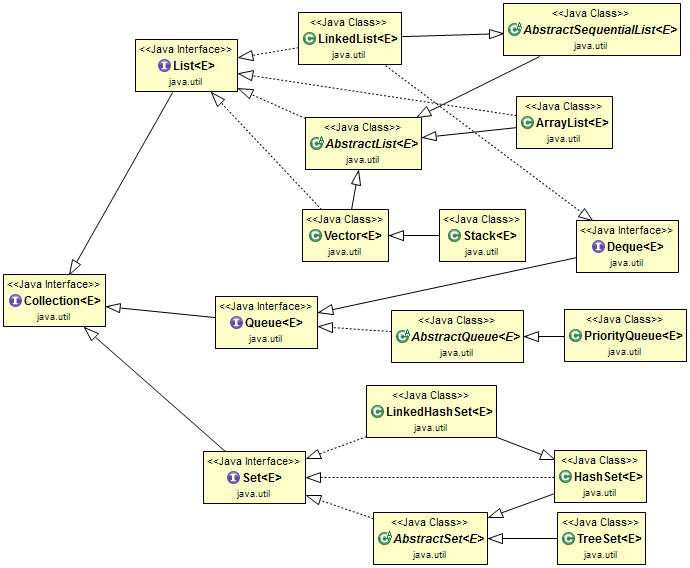
* dodavanje elemenata
* brisanje elemenata
* traženje određenog elementa

Za mjerenje efikasnosti pojedine operacije koristi se pojam **vremenska složenost** (engl. ***time complexity***) koji dolazi iz računske teorijske složenosti. Vremenska složenost opisuje trajanje algoritma s obzirom na veličinu ulaznih podataka te se najčešće zapisuje korištenjem **O-notacije**. Na primjer, složenost algoritma označava da će algoritam (u najgorem slučaju) izvršiti broj instrukcija linearno proporcionalan s veličinom ulaza ().

Jedna od najjednostavnijih struktura podataka — **polje**, već je prethodno spomenuta u priručniku. U polju su elementi složeni jedan iza drugoga u memoriji, a složenost osnovnih operacija dodavanja, brisanja i traženja elementa je što nije pretjerano pohvalno. S obzirom na to da je veličina polja fiksna, prilikom dodavanja novog elementa potrebno je definirati novo (veće) polje u koje zatim treba kopirati sve postojeće elemente uključujući i novi element. Kod brisanja, u slučaju da se obrisani element nalazi na sredini polja, preostale elemente potrebno je posmaknuti ulijevo kako bi se zadržao slijed elemenata polja. Također, s obzirom na to da elementi polja nisu posebno poredani, traženje određenog elementa zahtijeva obilazak polja i usporedbu sa svim postojećim elementima.

Kao što će biti prikazano u nastavku, postoje brojne druge strukture podataka koje imaju puno bolje performanse prilikom manipuliranja podacima. Programski jezik *Java* nudi razvojni okvir **Java Collections** koji sadrži skup razreda i sučelja koji implementiraju brojne strukture podataka odnosno **kolekcije**. Sve kolekcije nalaze se unutar paketa ***java.util***. Važno je napomenuti da Java kolekcije koriste tehnologiju *Java Generics* pa podržavaju rad s proizvoljnim tipovima podataka.

Dijagram najvažnijih razreda i sučelja unutar *Java Collections* okvira prikazan je u nastavku.



Sve strukture podataka odnosno kolekcije izvedene su iz sučelja **Collection<E>** koje se grana na različite podvrste kolekcija kao što su **liste** (sučelje List<E>) i **skupovi** (sučelje Set<E>). S obzirom na velik broj različitih kolekcija, u ovom priručniku obrađuju se samo osnovne kolekcije.

Osim konkretnih struktura podataka postoji i razred **Collections** koji sadrži mnoštvo korisnih metoda za rad s različitim vrstama kolekcija. Najvažnije metode toga razreda navedene su u nastavku.

|  |  |  |
| --- | --- | --- |
| **Metoda** | **Opis** | **Složenost** |
| **copy**(List<? super T> dest, List<? extends T> src) | Kopiranje elemenata iz jedne liste u drugu. |  |
| **disjoint**(Collection<?> c1, Collection<?> c2) | Provjera koja vraća istinu ako elementi ne sadrže zajedničke elemente. |  |
| **fill**(List<? super T>, T obj) | Zamjena svih elemenata s danim elementom. |  |
| **max**(Collection<? extends T> coll) | Traženje najvećeg elementa. |  |
| **min**(Collection<? extends T> coll) | Traženje najmanjeg elementa. |  |
| **replaceAll**(List<T> list, T oldVal, T newVal) | Zamjena svih pojavljivanja jednog elementa s drugim elementom. |  |
| **reverse**(List<?> list) | Okretanje redoslijeda elemenata. |  |
| **rotate**(List<?> list, int distance) | Rotiranje elemenata. |  |
| **shuffle**(List<?> list) | Nasumično permutiranje elemenata. |  |
| **sort**(List<T> list) | Sortiranje uzlaznim poretkom. |  |
| **swap**(List<?> list, int i, int j) | Zamjena dvaju elemenata. |  |

Sljedeća potpoglavlja detaljnije obrađuju osnovne podatkovne strukture iz *Java Collections* okvira.

1. 1. Lista

**Lista** (engl. ***List***) je apstraktna struktura podataka koja predstavlja konačan broj podataka poredanih u niz. U razvojnom okviru *Java Collections*, liste su definirane sučeljem **List<E>**.

Liste je moguće stvoriti direktno iz polja korištenjem metode **java.util.Arrays.asList()** na jedan od dvaju načina: prosljeđivanjem polja ili nabrajanjem elemenata u argumentima metode. Sljedeći primjer prikazuje spomenutu funkcionalnost.

|  |
| --- |
| **package** hr.unizg.srce.d470.kolekcije;  **import** java.util.Arrays;  **import** java.util.List;  **public** **class** StvoriListu {  **public** **static** **void** main(String[] args) {  Integer[] brojevi = { 2, 3, 4, 5 };  List<Integer> listaBrojeva =  Arrays.*asList*(brojevi);  List<String> voce = Arrays.*asList*("jabuka",  "banana",  "breskva");  System.***out***.println(listaBrojeva);  System.***out***.println(voce);  }  } |
| Izlaz:  [2, 3, 4, 5]  [jabuka, banana, breskva] |

Liste su najčešće implementirane korištenjem spomenutoga **polja** (engl. ***array***) ili pak korištenjem **vezane liste** (engl. ***linked list***). Postoji nekoliko konkretnih implementacija navedenoga sučelja, a najpoznatiji su razredi ArrayList<E> i LinkedList<E>.

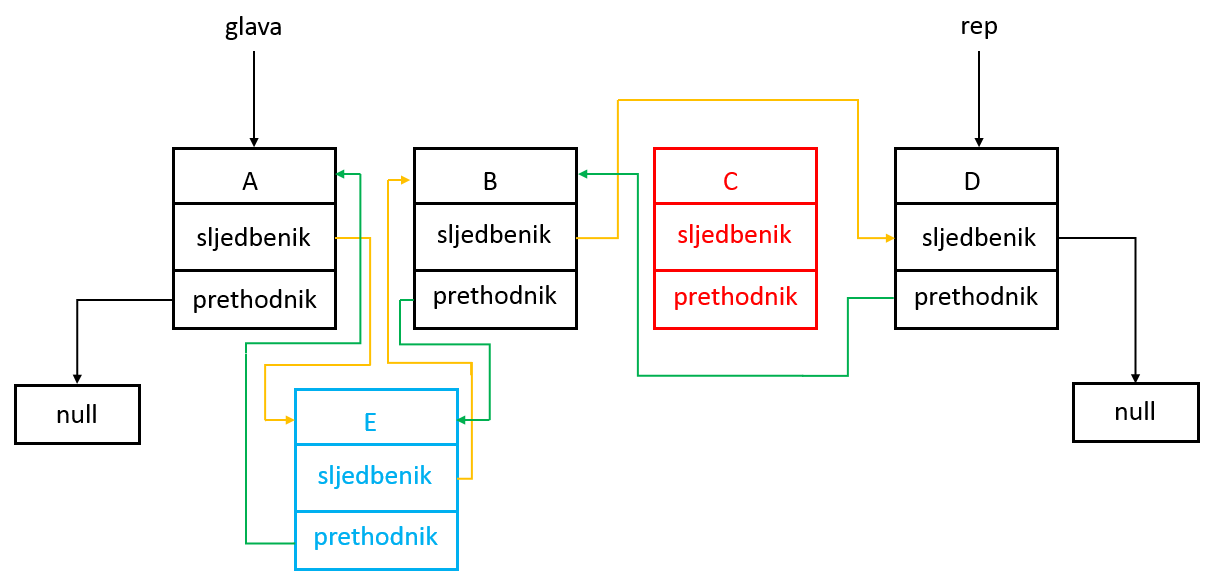
**ArrayList<E>** predstavlja listu implementiranu korištenjem polja, ali s ugrađenim automatskim proširivanjem kapaciteta u slučaju dodavanja elemenata. Sljedeća tablica prikazuje informacije o najvažnijim metodama ovoga razreda.

|  |  |  |
| --- | --- | --- |
| **Metoda** | **Opis** | **Složenost** |
| **add**(E e) | Dodavanje na kraj. |  |
| **add**(int index, E e) | Dodavanje na određenu poziciju. |  |
| **get**(int index) | Pristup određenom elementu s pozicije. |  |
| **contains**(Object o) | Traženje elementa po vrijednosti. |  |
| **remove**(int index) | Brisanje elementa s pozicije. |  |

**LinkedList<E>** predstavlja implementaciju **vezane liste**. Za razliku od polja, vezane liste nisu smještene sljedno u memoriji. Iz tog razloga nije moguće direktno pristupiti elementu liste preko numeričkog indeksa. Kako bi lista bila povezana, svaki element vezane liste sadrži referencu na sljedeći i/ili prethodni element, čime lista postaje **jednostruko** ili **dvostruko povezana**. Vezana lista najčešće čuva referencu na prvi i/ili zadnji element liste (**glava** i **rep**), a pristup određenom elementu obavlja se iteriranjem po elementima koristeći veze unutar elemenata. Sljedeća slika ilustrira dvostruko povezanu listu.



LinkedList<E> sadrži jednake metode navedene u prethodnoj tablici s jednakom složenosti. Glavna prednost vezanih listi jest mogućnost dodavanja i brisanja elemenata s bilo koje pozicije sa složenošću , **ako i samo ako** je u danom trenutku dostupna referenca na objekt koji treba ukloniti ili pored kojeg treba dodati novi element. U tom slučaju dovoljno je preusmjeriti reference susjednih elemenata, kao što je prikazano na sljedećoj slici.



* + 1. Iterator listi

Kako bi bio moguć učinkovit obilazak liste, dodavanje i brisanje elemenata, sučelje List<E> sadrži metodu **listIterator(int index)** koja vraća objekt tipa **ListIterator<E>**. Taj objekt služi za **dvosmjerni** obilazak liste te sadrži **kursor** koji predstavlja trenutnu poziciju iteratora. Važno je napomenuti da pozicije kursora predstavljaju lokacije **između** elemenata liste.

Najvažnije metode razreda ListIterator<E> navedene su u sljedećoj tablici sa složenostima za obje implementacije listi.

|  |  |  |  |
| --- | --- | --- | --- |
| **Metoda** | **Opis** | **Složenost (LinkedList)** | **Složenost (ArrayList)** |
| **add**(E e) | Dodavanje elementa na poziciju kursora. |  |  |
| **hasNext**() | Provjera ima li sljedećeg elementa. |  |  |
| **hasPrevious**() | Provjera ima li prethodnog elementa. |  |  |
| E **next**() | Vraća sljedeći element liste i pomiče kursor unaprijed. |  |  |
| E **previous**() | Vraća prethodni element liste i pomiče kursor unatrag. |  |  |
| **remove**() | Uklanja element koji je vratila posljednja next() ili previous() operacija. |  |  |
| **set**(E e) | Zamjena danog elementa s elementom koji je vratila posljednja next() ili previous() operacija. |  |  |

Primjer korištenja listi prikazan je u nastavku.

**Obilazak i ispis kolekcija**

Sve *Java* kolekcije implementiraju mehanizam koji omogućuje obilazak po elementima koristeći skraćeni oblik for petlje.

Također, kolekcije implementiraju metodu toString() koja daje jasan tekstualni prikaz kolekcije i njenih elemenata.

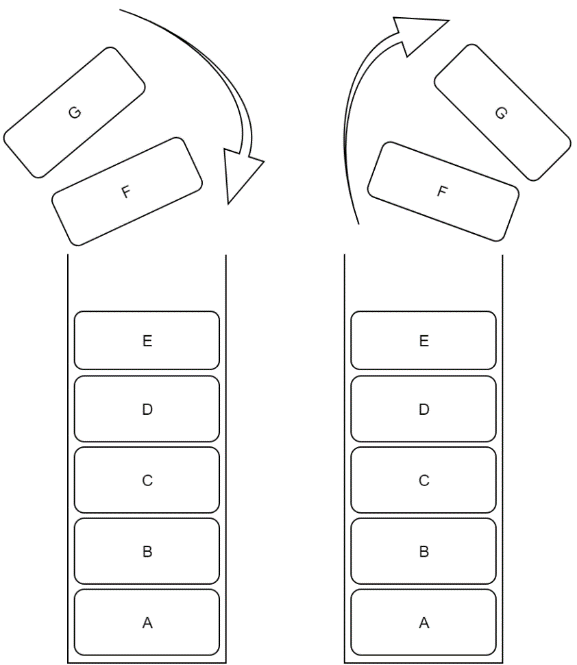
|  |
| --- |
| **package** hr.unizg.srce.d470.kolekcije;  **import** java.util.Collections;  **import** java.util.LinkedList;  **import** java.util.List;  **import** java.util.ListIterator;  **public** **class** Zivotinje {  **public** **static** **void** main(String[] args) {  List<String> lista = **new** LinkedList<>();  **for** (String arg : args)  lista.add(arg);  System.***out***.println("Lista: " + lista);  System.***out***.println("Prva zivotinja je: "  + lista.get(0));  System.***out***.println("Zadnja zivotinja je: "  + lista.get(lista.size() - 1));  **if** (lista.contains("ovan"))  lista.remove("ovan");  ListIterator<String> iterator =  lista.listIterator();  // Svim zivotinjama zapisi imena  // velikim slovima.  **while** (iterator.hasNext()) {  String zivotinja = iterator.next();  iterator.set(zivotinja.toUpperCase());  }  System.***out***.println(lista);  // Sortiraj listu.  Collections.*sort*(lista);  System.***out***.println("Sortirana lista: "  + lista);  // Izbrisi posljednje tri zivotinje.  **for** (**int** i = 0; i < 3 &&  iterator.hasPrevious(); ++i) {  iterator.remove();  iterator.previous();  System.***out***.println(lista);  }  }  } |
| Ulaz:  zec pas lav ovan konj tigar slon  Izlaz:  Lista: [zec, pas, lav, ovan, konj, tigar, slon]  Prva zivotinja je: zec  Zadnja zivotinja je: slon  [ZEC, PAS, LAV, KONJ, TIGAR, SLON]  Sortirana lista: [KONJ, LAV, PAS, SLON, TIGAR,  ZEC]  [KONJ, LAV, PAS, SLON, TIGAR]  [KONJ, LAV, PAS, SLON]  [KONJ, LAV, PAS] |

Liste mogu poslužiti kao baza za implementaciju drugih korisnih struktura kao što su **stog** i **red**.

* 1. Stog

**Stog** (engl. ***stack***) je apstraktna struktura podataka kod koje se svaki novi element dodaje na vrh strukture. Prilikom uklanjanja elemenata sa stoga, najprije se uzima element s vrha stoga, odnosno element koji je zadnji dodan na stog. Iz tog razloga se kaže da je stog **LIFO** struktura (engl. ***Last In First Out***).

U svakodnevnom životu primjer stoga bilo bi skladištenje tanjura za jelo. Nakon što se neki tanjur očisti, on se stavlja na vrh već postojeće hrpe tanjura, a kad trebamo tanjur, u tom slučaju uzimamo tanjur koji se nalazi na vrhu.[[1]](#footnote-1) [15]



*Java Collections* implementira stog u razredu **Stack<E>** koji sadrži brojne metode od kojih su najvažnije navedene u sljedećoj tablici. U nastavku je prikazan primjer korištenja stoga.

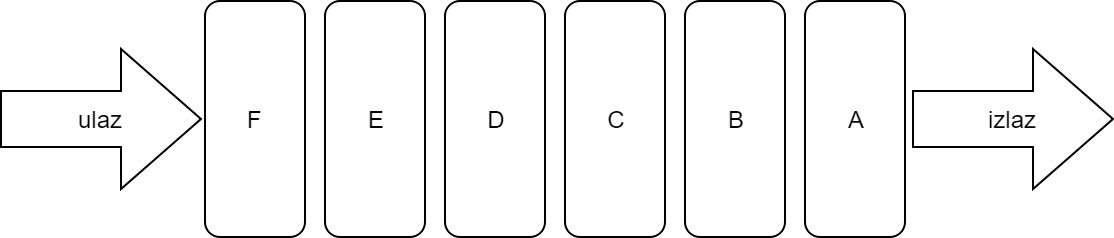
|  |  |  |
| --- | --- | --- |
| **Metoda** | **Opis** | **Složenost** |
| boolean **empty**() | Provjera je li stog prazan. |  |
| E **push**(E item) | Dodavanje na vrh. |  |
| E **peek**() | Dohvat elementa s vrha bez uklanjanja. U slučaju praznoga stoga dogodit će se EmptyStackException. |  |
| E **pop**() | Dohvat i uklanjanje elementa s vrha. U slučaju praznoga stoga dogodit će se EmptyStackException. |  |
| int **search**(Object o) | Traženje određenog elementa. |  |

|  |
| --- |
| **package** hr.unizg.srce.d470.kolekcije;  **import** java.util.Stack;  **public** **class** ObrniRedoslijed {  **public** **static** **void** main(String[] args) {  Stack<String> stog = **new** Stack<>();  **for** (String arg : args)  stog.push(arg);  **while** (!stog.empty())  System.***out***.print(stog.pop() + " ");  }  } |
| Ulaz:  1 2 3 4 5  Izlaz:  5 4 3 2 1 |

* 1. Red

**Red** (engl. ***queue***) je apstraktna struktura podataka kod koje se svaki novi element dodaje na kraj strukture. Prilikom uklanjanja elemenata iz reda, najprije se uzima element s početka reda, odnosno element koji je prvi dodan u red. Iz tog razloga se kaže da je red **FIFO** struktura (***First In First Out***).

U svakodnevnom životu primjer reda bilo bi čekanje u banci. Osoba koja je došla prva, bit će prva poslužena, dok će zadnja osoba biti zadnja poslužena, odnosno mora čekati dok sve prethodne osobe ne budu poslužene.[[2]](#footnote-2) [15]



*Java Collections* definira red kroz sučelje **Queue<E>** koje nudi brojne metode od kojih su najvažnije navedene u sljedećoj tablici. Konkretna implementacija reda nalazi se u prethodno spomenutom razredu **LinkedList<E>**. U nastavku je prikazan primjer korištenja reda.

|  |  |  |
| --- | --- | --- |
| **Metoda** | **Opis** | **Složenost** |
| boolean **isEmpty**() | Provjeravanje je li red prazan. |  |
| boolean **add**(E e) | Dodavanje na kraj reda. |  |
| E **element**() | Dohvat elementa s početka bez uklanjanja. U slučaju praznoga reda dogodit će se NoSuchElementException. |  |
| E **remove**() | Dohvat i uklanjanje elementa s početka. U slučaju praznoga reda dogodit će se NoSuchElementException. |  |
| boolean **contains**(Object o) | Traženje određenog elementa. |  |

|  |
| --- |
| **package** hr.unizg.srce.d470.kolekcije;  **import** java.util.LinkedList;  **import** java.util.Queue;  **public** **class** RedCekanja {  **public** **static** **void** main(String[] args) {  Queue<String> red = **new** LinkedList<>();  System.***out***.println(red);  red.add("Ana");  red.add("Ivan");  red.add("Luka");  **for** (String element : red)  System.***out***.print(element + " ");  System.***out***.println("");  System.***out***.println("Prva osoba u redu"  + " je: " + red.element());  red.remove();  red.remove();  System.***out***.println(red);  red.add("Petra");  red.add("Josip");  System.***out***.print("Red sada sadrzi: ");  **while** (!red.isEmpty())  System.***out***.print(red.remove() + " ");  }  } |
| Izlaz:  []  Ana Ivan Luka  Prva osoba u redu je: Ana  [Luka]  Red sada sadrzi: Luka Petra Josip |

* 1. Vježba: Kolekcije

1. (***ConnectX***) U razredu Ploca<T> iskoristite *Java* kolekcije za spremanje žetona. Zamjenite postojeće dvodimenzionalno polje objekata baznoga razreda Object čime će se dodatno osigurati integritet tipova podataka.
2. Napišite program koji računa rezultat matematičkog izraza koji je napisan u **postfiksnoj notaciji**. Na primjer, izraz „2 + 3 x 5“ će u postfiksnoj notaciji biti „2 3 5 x +“. Traženi izraz učitajte preko ulaznih argumenata programa. Potrebno je omogućiti podršku za matematičke izraze koji sadrže brojeve (operande) te osnovne operatore **zbrajanja**, **oduzimanja**, **množenja** i **dijeljenja**. Također, potrebno je obraditi moguće pogreške koje se mogu dogoditi unošenjem nepravilnoga matematičkog izraza. U nastavku je prikazano nekoliko primjera.   
   (Pomoć: Iskoristite strukturu podataka **stog**.)

|  |
| --- |
| Ulaz:  2 3 5 x +  Izlaz:  Rezultat je: 17.0 |
| Ulaz:  3 4 2 1 – x +  Izlaz:  Rezultat je: 7.0 |
| Ulaz:  3 4 2 x 1 5 - / 2 x 3 - +  Izlaz:  Rezultat je: -4.0 |
| Ulaz:  3 a +  Izlaz:  Nepoznati argument: "a" |
| Ulaz:  6  Izlaz:  Rezultat je 6.0 |
| Ulaz:  3 5  Izlaz:  Nevaljani matematicki izraz. |

* 1. Pitanja za ponavljanje: Kolekcije

1. Što su strukture podataka i koje osnovne operacije podržavaju?
2. Što je vremenska složenost algoritma i kako se izražava?
3. Koja je složenost dodavanja, brisanja i traženja elemenata u polju?
4. Navedite dvije osnovne vrste listi i njihove razlike.
5. Što je stog? Navedite primjer stoga iz svakodnevnog života.
6. Navedite neke od metoda koje sadrži sučelje Queue<E>.

1. Oblikovni obrasci

Po završetku ovoga poglavlja polaznik će moći:

definirati oblikovne obrasce i njihove primjene

implementirati obrasce Kompozit, Prototip, Iterator i Strategija.

Prilikom razvoja objektno orijentiranih sustava vrlo je lako osmisliti rješenje koje je neučinkovito te koje je vrlo teško nadograditi i održavati. Iz tog razloga treba poznavati **oblikovne obrasce** – formalizirane predloške isprobanih rješenja za najčešće probleme u razvoju programa [11]. Oblikovni obrasci pojavili su se krajem 20. stoljeća kad su četvorica autora detaljno opisala rješenja različitih problema koji su se u to vrijeme često pojavljivali u objektno orijentiranom razvoju programa [12]. Ta rješenja (obrasci) **nisu strogo definirana** u nekom programskom jeziku, već su opisana na konceptualnoj razini pa se konkretna implementacija prepušta pojedincima. Takav generalizirani pristup omogućuje primjenu obrazaca na cijele skupove sličnih problema. Oblikovne obrasce korisno je poznavati kako bismo u praksi mogli prepoznati različite probleme u dizajnu te ih efikasno riješiti.

Oblikovnih obrazaca ima mnoštvo te ih prema njihovoj primjeni možemo podijeliti u tri skupine:

1. **Strukturalni obrasci** (engl. ***Structural patterns***) koji definiraju različite načine za efikasno definiranje kompleksnih struktura podataka.
2. **Obrasci stvaranja** (engl. ***Creational patterns***) koji sadrže brojne mehanizme za fleksibilno i efikasno stvaranje objekata.
3. **Obrasci ponašanja** (engl. ***Behavioral patterns***) koji se brinu za funkcionalnost sustava i pravilno raspoređivanje odgovornosti među objektima.

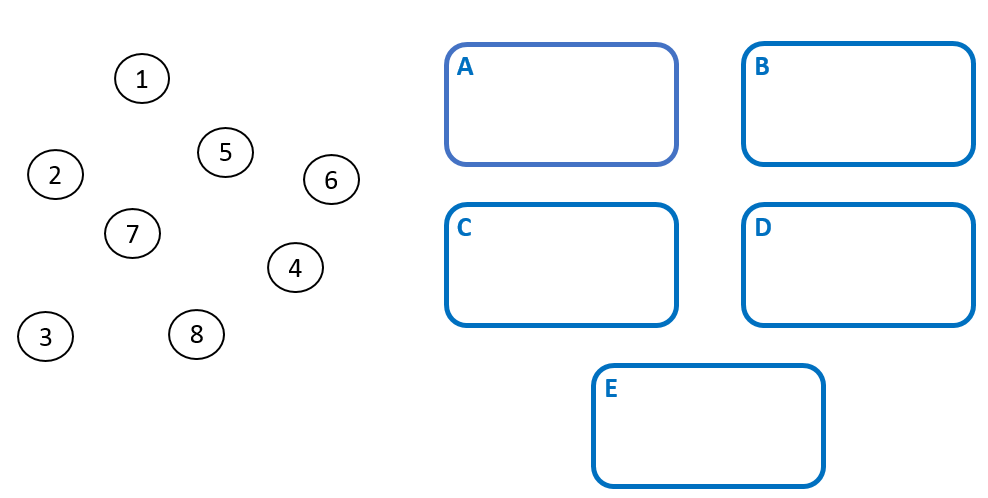
Važno je napomenuti da oblikovni obrasci **nisu nužno najbolje rješenje** za sve probleme. Svaki obrazac ima svoje prednosti i mane, a implementacija istih može unijeti kompleksnost u sustav koja nije nužno potrebna. Često se događa da početnici žele svugdje primijeniti obrasce, čak i u slučajevima gdje postoji jednostavnije rješenje.

U nastavku su kroz primjer detaljno obrađena četiri obrasca — **Kompozit**, **Prototip**, **Iterator** i **Strategija**.

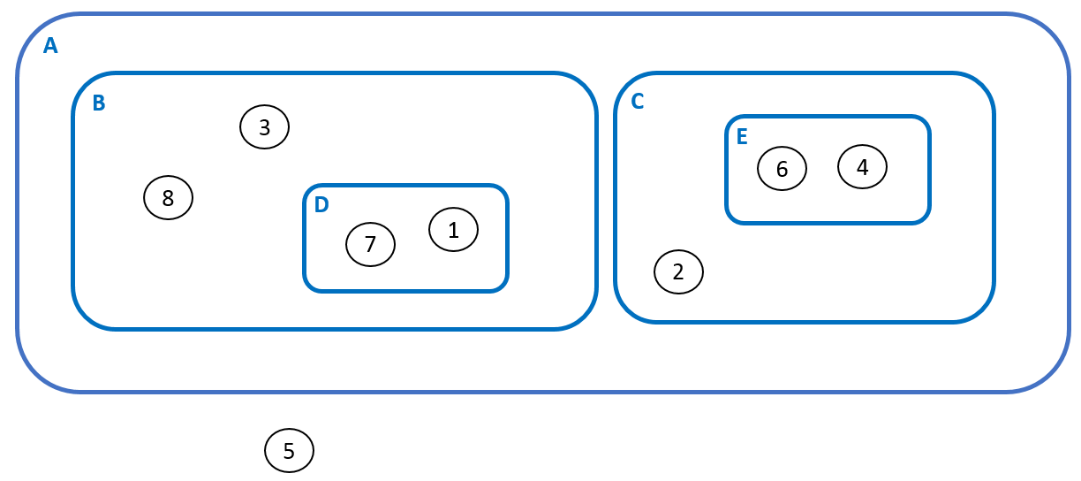
1. 1. Kompozit (engl. *Composite Pattern*)

**Kompozit** (engl. ***Composite Pattern***) je strukturalni oblikovni obrazac koji omogućuje povezivanje objekata u **stablaste strukture** (engl. ***tree structures***) te njihovo fleksibilno korištenje.

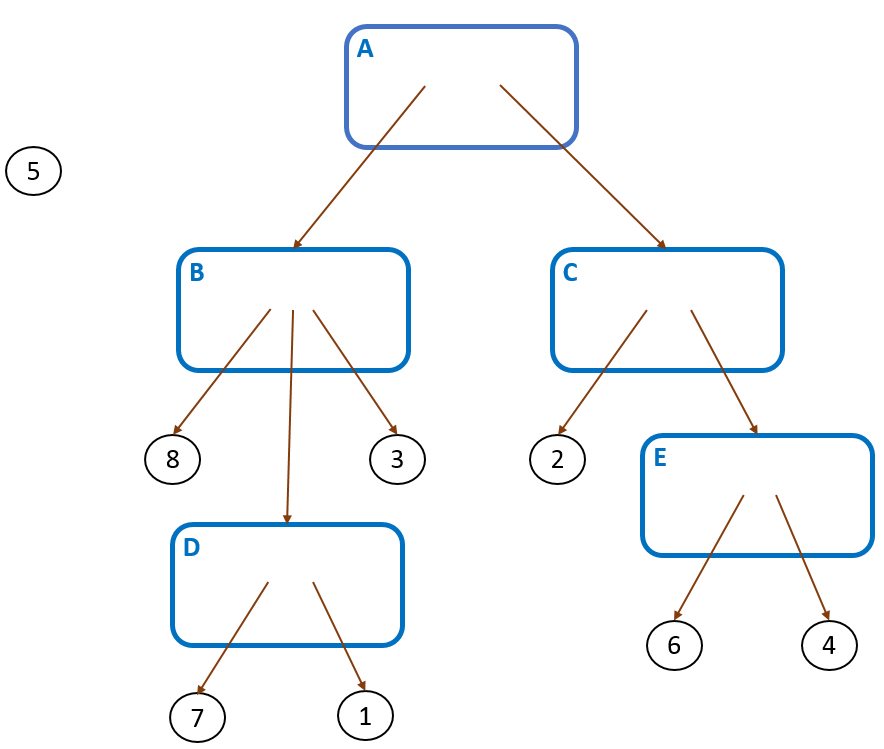
Ovaj obrazac sastoji se od osnovnih **elemenata** (**listova**) koji se mogu povezivati u **grupe** (**kompozite**) koje mogu sadržavati različit broj elemenata u sebi. Sljedeća slika prikazuje primjer elemenata i grupa pri čemu elementi sadrže **broj** (1 — 8), dok grupe imaju svoj **naziv** (A — E).



Elementi i grupe mogu se strukturirati kao na sljedećoj slici pri čemu grupe mogu sadržavati elemente ili čak druge grupe. Postoje brojni praktični primjeri ovakvoga strukturiranja poput raspoređivanja ljudi u prostorije ili predmeta u kutije. Grupe mogu predstavljati i menadžere koji su odgovorni za druge zaposlenike ili menadžere u kompaniji čime se oblikuje hijerarhija zaposlenika.



Prethodna struktura može se prikazati u obliku stabla kao što je vidljivo na sljedećoj slici. Strelice na slici predstavljaju sadržanost elemenata ili grupa unutar drugih grupa. Važno je primijetiti da osnovni elementi ne mogu sadržavati druge elemente ili grupe, zbog čega se za njih kaže da su **listovi stabla**.



Nad složenim strukturama potrebno je omogućiti obavljanje određenih **operacija**, npr. **traženje određenoga broja**. Prednost obrasca Kompozit jest u tome što omogućuje tretiranje osnovnih elemenata i složenih struktura na jednak način kroz **zajedničko sučelje**. Sljedeći isječak kôda prikazuje sučelje **Komponenta** koje sadrži jednu operaciju za traženje određenoga broja.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.kompozit;  **public** **interface** Komponenta {  **boolean** sadrzi(**int** broj);  } |

Implementirajući zajedničko sučelje moguće je efikasno definirati osnovne elemente i grupe kao što je vidljivo u sljedećem primjeru. Razred **Element** sadrži atribut **broj** te implementira metodu sučelja **sadrzi()** na jednostavan način tako da usporedi vlastiti atribut s traženim brojem.

S druge strane, razred **Kompozit** ne sadrži broj već **naziv grupe** i **listu komponenti** koja predstavlja sadržane elemente i/ili grupe. Implementacija metode **sadrzi()** je mrvicu složenija jer se u petlji redom ispituju sadržane komponente. U ovoj metodi važno je uočiti fleksibilnost ovog obrasca gdje se sadržani elementi i grupe tretiraju **na jednak način** kao objekti tipa **Komponenta**. Ova metoda **rekurzivno** će pretražiti cijelu strukturu bez obzira na njen oblik, dok će se metoda **sadrzi()** polimorfno pozivati ovisno o stvarnom tipu komponente (Element ili Kompozit).

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.kompozit;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** Element **implements** Komponenta {  **private** **final** **int** broj;  **public** Element(**int** broj) {  **this**.broj = broj;  }  **public** **int** getBroj() {  **return** broj;  }  @Override  **public** String toString() {  **return** "Element [broj=" + getBroj() + "]";  }  @Override  **public** **boolean** sadrzi(**int** broj) {  System.***out***.println(**this**);  **return** getBroj() == broj;  }  }  **public** **class** Kompozit **implements** Komponenta {  **private** String nazivGrupe;  **private** List<Komponenta> komponente;  **public** Kompozit(String nazivGrupe) {  **this**.nazivGrupe = nazivGrupe;  komponente = **new** ArrayList<Komponenta>();  }  **private** String getNazivGrupe() {  **return** nazivGrupe;  }  **private** List<Komponenta> getKomponente() {  **return** komponente;  }  **public** **void** dodaj(Komponenta komponenta) {  getKomponente().add(komponenta);  }  @Override  **public** String toString() {  **return** "Kompozit [nazivGrupe="  + getNazivGrupe() + "]";  }  @Override  **public** **boolean** sadrzi(**int** broj) {  System.***out***.println(**this**);  **for** (Komponenta komponenta : getKomponente())  **if** (komponenta.sadrzi(broj))  **return** **true**;  **return** **false**;  }  } |

Sljedeći primjer prikazuje korištenje prethodno definirane kompozitne strukture. Pomoćna metoda **stvoriStrukturu()** stvara elemente i grupe te ih međusobno povezuje u prethodno ilustrirano stablo. U glavnoj metodi **main()** stvorena struktura ispituje se za postojanje brojeva 5 i 2.

Na izlazu je prikazan **redoslijed pozivanja metode sadrzi()** unutar strukture te krajnji rezultat pretrage. U prvom slučaju, s obzirom na to da struktura ne sadrži broj 5, algoritam će obići sve komponente te na kraju ispisati **false**. U slučaju traženja broja 2, vidljivo je da se algoritam zaustavlja u trenutku pronalaska traženog elementa koji sadrži traženi broj.

**Pretraživanje u dubinu**

Zanimljivo je primijetiti specifičan način obilaska strukture koji je rezultat rekurzivnoga pozivanja metode sadrzi(). Taj algoritam naziva se **pretraživanje u dubinu** (engl. ***depth first search***, ***DFS***) te ima široku primjenu u računarskoj znanosti.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.kompozit;  **public** **class** GlavniProgram {  **private** **static** Komponenta stvoriStrukturu() {  Element[] elementi = **new** Element[8];  **for** (**int** i = 0; i < elementi.length; ++i)  elementi[i] = **new** Element(i + 1);  Kompozit[] kompoziti = **new** Kompozit[5];  **for** (**int** i = 0; i < kompoziti.length; ++i)  kompoziti[i] = **new** Kompozit(String.*valueOf*(  Character.*valueOf*((**char**) ('A' + i))));  // Veze  kompoziti[4].dodaj(elementi[5]);  kompoziti[4].dodaj(elementi[3]);  kompoziti[3].dodaj(elementi[6]);  kompoziti[3].dodaj(elementi[0]);  kompoziti[2].dodaj(elementi[1]);  kompoziti[2].dodaj(kompoziti[4]);  kompoziti[1].dodaj(elementi[7]);  kompoziti[1].dodaj(kompoziti[3]);  kompoziti[1].dodaj(elementi[2]);  kompoziti[0].dodaj(kompoziti[1]);  kompoziti[0].dodaj(kompoziti[2]);  **return** kompoziti[0];  }  **public** **static** **void** main(String[] args) {  Komponenta struktura = *stvoriStrukturu*();  System.***out***.println(struktura.sadrzi(5));  System.***out***.println(struktura.sadrzi(2));  }  } |
| Izlaz:  Kompozit [nazivGrupe=A]  Kompozit [nazivGrupe=B]  Element [broj=8]  Kompozit [nazivGrupe=D]  Element [broj=7]  Element [broj=1]  Element [broj=3]  Kompozit [nazivGrupe=C]  Element [broj=2]  Kompozit [nazivGrupe=E]  Element [broj=6]  Element [broj=4]  **false**  Kompozit [nazivGrupe=A]  Kompozit [nazivGrupe=B]  Element [broj=8]  Kompozit [nazivGrupe=D]  Element [broj=7]  Element [broj=1]  Element [broj=3]  Kompozit [nazivGrupe=C]  Element [broj=2]  **true** |

* 1. Prototip (engl. *Prototype Pattern*)

**Prototip** (engl. ***Prototype Pattern***) je obrazac stvaranja koji omogućuje fleksibilno kopiranje postojećih objekata bez uvođenja nepotrebnih ovisnosti između razreda.

Kako bi se određeni objekt kopirao, potrebno je kopirati sve njegove atribute, što ponekad nije moguće, na primjer, ako su pojedini atributi deklarirani kao privatni. Oblikovni obrazac **Prototip** delegira proces kloniranja samim objektima koje je potrebno kopirati. Kako bi se to postiglo, potrebno je deklarirati **zajedničko sučelje** za sve objekte koji podržavaju kloniranje.

U sljedećim primjerima omogućuje se kloniranje prethodno definirane kompozitne strukture. Najprije je potrebno definirati zajedničko sučelje **Prototip** koje sadrži metodu **kloniraj()** koja vraća objekt tipa Prototip. Zatim je potrebno proširiti sučelje **Komponenta** na način da naslijedi sučelje Prototip čime se deklarira mogućnost kloniranja komponenti strukture.

**Java sučelje *Cloneable***

Programski jezik Java već sadrži mehanizam za implementiranje obrasca Prototip korištenjem sučelja Cloneable koje sadrži metodu clone().

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.prototip;  **public** **interface** Prototip {  Prototip kloniraj();  }  **public** **interface** Komponenta **extends** Prototip {  **boolean** sadrzi(**int** broj);  } |

S obzirom na to da je prošireno sučelje Komponenta, potrebno je u razredima **Element** i **Kompozit** implementirati apstraktnu metodu **kloniraj()** kao što je prikazano u sljedećem primjeru. Implementacija se najčešće obavlja definiranjem tzv. **kopirajućega konstruktora** (engl. ***copy constructor***) koji preko argumenta prima objekt istoga tipa koji je potrebno kopirati. Ovaj konstruktor treba sadržavati svu logiku kopiranja, kopiranje atributa itd. U slučaju razreda Element kopiranje je jednostavno, dok je kod razreda Kompozit potrebno **rekurzivno klonirati** sadržane komponente. Na kraju je u metodi **kloniraj()** dovoljno pozvati novi konstruktor.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.prototip;  **public** **class** Element **implements** Komponenta {  **...**  **public** Element(Element element) {  **this**(element.getBroj());  }  @Override  **public** Prototip kloniraj() {  **return** **new** Element(**this**);  }  }  **public** **class** Kompozit **implements** Komponenta {  **...**  **public** Kompozit(Kompozit kompozit) {  **this**(kompozit.nazivGrupe);  **for** (Komponenta komponenta :  kompozit.getKomponente())  getKomponente().add((Komponenta)  komponenta.kloniraj());  }  @Override  **public** Prototip kloniraj() {  **return** **new** Kompozit(**this**);  }  } |

U sljedećem primjeru prikazano je kloniranje prethodno ilustrirane složene strukture koja sadrži više različitih komponenti, elemenata i grupa.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.prototip;  **public** **class** GlavniProgram {  **...**  **public** **static** **void** main(String[] args) {  Komponenta struktura = *stvoriStrukturu*();  Komponenta novaStruktura = (Komponenta)  struktura.kloniraj();  System.***out***.println(novaStruktura.sadrzi(5));  }  } |
| Izlaz:  Kompozit [nazivGrupe=A]  Kompozit [nazivGrupe=B]  Element [broj=8]  Kompozit [nazivGrupe=D]  Element [broj=7]  Element [broj=1]  Element [broj=3]  Kompozit [nazivGrupe=C]  Element [broj=2]  Kompozit [nazivGrupe=E]  Element [broj=6]  Element [broj=4]  **False** |

* 1. Iterator (engl. *Iterator Pattern*)

**Iterator** (engl. ***Iterator Pattern***) je obrazac ponašanja koji omogućuje obilazak elemenata unutar strukture podataka bez otkrivanja detalja o samoj strukturi. Iteratori su već djelomično spomenuti u poglavlju o kolekcijama gdje se obradio iterator listi.

Glavna ideja ovog obrasca jest odvajanje logike obilaska u **zasebni objekt (iterator)** koji će enkapsulirati detalje samog obilaska poput trenutnog elementa ili broj elemenata do kraja obilaska. Ovakav pristup omogućuje stvaranje nekoliko iteratora koji mogu obilaziti elemente strukture neovisno jedan o drugome.

Programski jezik *Java* sadrži mehanizam za implementiranje ovog obrasca korištenjem sučelja **Iterable<T>** i **java.util.Iterator<E>** čije su apstraktne metode navedene u sljedećoj tablici.

|  |  |
| --- | --- |
| **Sučelje** | **Metode** |
| Iterable<T> | Iterator<T> iterator() |
| Iterator<E> | boolean hasNext()  E next() |

Razredi koji implementiraju Iterable<T> predstavljaju strukture koje imaju mogućnost obilaska po elementima tipa **T**. Sljedeći primjer prikazuje postojeće sučelje Komponenta koje je sada prošireno sučeljem **Iterable<Integer>** kako bi se omogućilo obilaženje brojeva koji su sadržani u komponentama.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.iterator;  **public** **interface** Komponenta **extends** Prototip,  Iterable<Integer> {  **boolean** sadrzi(**int** broj);  } |

Sučelje Iterable<T> sadrži metodu **iterator()** koju izvedeni razredi moraju implementirati te koja mora vratiti objekt tipa Iterator<T>. Taj objekt predstavlja instancu iteratora koji mora sadržavati logiku obilaženja elemenata. S obzirom na to da postoji više različitih vrsta komponenti (element i kompozit), potrebno je definirati posebne iteratore za svaki od razreda.

Sljedeći primjer prikazuje prošireni razred Element koji sadrži definiciju ugniježđenoga razreda **ElementIterator** koji implementira Iterator<Integer>. S obzirom na to da se radi o **ugniježđenom razredu**, ElementIterator ima pristup **svim privatnim atributima** razreda Element, što uvelike olakšava implementaciju obilaska, kao što će biti vidljivo na primjeru razreda Kompozit.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.iterator;  **import** java.util.Iterator;  **public** **class** Element **implements** Komponenta {  ...  @Override  **public** Iterator<Integer> iterator() {  **return** **new** ElementIterator(**this**);  }  **private** **class** ElementIterator **implements**  Iterator<Integer> {  **private** **boolean** hasNext;  **private** Element element;  **public** ElementIterator(Element element) {  **this**.element = element;  hasNext = **true**;  }  @Override  **public** **boolean** hasNext() {  **return** hasNext;  }  @Override  **public** Integer next() {  hasNext = **false**;  **return** element.getBroj();  }  }  } |

Iteratori moraju imati pristup strukturi koju je potrebno obići pa je najčešća praksa **proslijediti objekt strukture preko konstruktora**, kao što je vidljivo na prethodnom primjeru. Također, razred ElementIterator mora implementirati dvije metode: **hasNext()** i **next()**. Metoda hasNext() provjerava postoji li još elemenata koje je potrebno obići, dok next() vraća sljedeći element. S obzirom na to da razred Element sadrži samo jedan broj, dovoljno je njega vratiti u prvom pozivu metode next(). Kako bi implementacija razreda Element bila potpuna, potrebno je naposljetku implementirati metodu iterator() gdje se jednostavno vraća novi objekt tipa ElementIterator.

Sljedeći primjer prikazuje implementaciju iteratora u slučaju razreda Kompozit. Ovaj primjer je složeniji jer je potrebno obilaziti listu sadržanih komponenti te svaku podkomponentu zasebno iterirati. Iz tog razloga potrebno je pamtiti indeks podkomponente u listi te iterator trenutne podkomponente. Rezultat cijele implementacije jest **rekurzivno obilaženje cijele strukture**.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.iterator;  **import** java.util.ArrayList;  **import** java.util.Iterator;  **import** java.util.List;  **public** **class** Kompozit **implements** Komponenta {  **...**  @Override  **public** Iterator<Integer> iterator() {  **return** **new** KompozitIterator(**this**);  }  **private** **class** KompozitIterator **implements**  Iterator<Integer> {  **private** Kompozit kompozit;  **private** **int** index;  **private** Iterator<Integer> podIterator;  **public** KompozitIterator(Kompozit kompozit) {  **this**.kompozit = kompozit;  }  @Override  **public** **boolean** hasNext() {  **return** index <  kompozit.getKomponente().size();  }  @Override  **public** Integer next() {  **if** (podIterator == **null**)  podIterator = kompozit.getKomponente()  .get(index)  .iterator();  Integer rezultat = podIterator.next();  **if** (!podIterator.hasNext()) {  ++index;  podIterator = **null**;  }  **return** rezultat;  }  }  } |

U nastavku je prikazan glavni program koji koristi prethodno implementirani obrazac kako bi redom ispisao brojeve sadržane unutar strukture.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.prototip;  **public** **class** GlavniProgram {  **...**  **public** **static** **void** main(String[] args) {  Komponenta struktura = *stvoriStrukturu*();  System.***out***.print("Struktura sadrzi brojeve:");  **for** (Integer broj : struktura) {  System.***out***.print(" " + broj);  }  }  } |
| Izlaz:  Struktura sadrzi brojeve: 8 7 1 3 2 6 4 |

Naizgled nije očito korištenje iteratora jer se stvarna logika iteratora odvija u pozadini skraćenog oblika petlje for (tzv. **foreach** petlje) pri čemu se automatski stvaraju iteratori te pozivaju metode hasNext() i next(). Istu logiku moguće je ručno napisati korištenjem obične **for** petlje na sljedeći način:

|  |
| --- |
| **for** (Iterator<Integer> it = struktura.iterator();  it.hasNext();) {  **int** broj = it.next();  System.***out***.print(" " + broj);  } |

* 1. Strategija (engl. *Strategy Pattern*)

**Strategija** (engl. ***Strategy Pattern***) je obrazac ponašanja koji omogućuje definiranje razreda algoritama (strategija) te njihovo fleksibilno korištenje. U prethodnim primjerima definirana je kompleksna struktura Komponenta koja sadrži skup brojeva. U ovom podpoglavlju potrebno je omogućiti različite operacije nad njima, poput traženja maksimuma ili računanje zbroja.

Ovaj obrazac definira skup algoritama kroz **zajedničko sučelje** koje sadrži samo jednu metodu za pokretanje algoritma. Sljedeći primjer prikazuje sučelje **Agregacija** koje predstavlja skup algoritama koji za danu komponentu vraćaju broj koji može biti zbroj, maksimum, umnožak, itd..

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.strategija;  **public** **interface** Agregacija {  **int** agregiraj(Komponenta komponenta);  } |

Konkretne strategije definiraju se implementacijom navedenoga sučelja. U nastavku su prikazane implementacije dviju strategija — **zbrajanje parnih elemenata** te **traženje maksimuma**. Obje strategije na jednostavan način obilaze elemenate komponente zbog činjenice da Komponenta implementira obrazac Iterator.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.strategija;  **public** **class** AgregacijaZbrojParnih **implements**  Agregacija {  @Override  **public** **int** agregiraj(Komponenta komponenta) {  **int** rezultat = 0;  **for** (**int** broj : komponenta) {  **if** (broj % 2 == 0)  rezultat += broj;  }  **return** rezultat;  }  }  **public** **class** AgregacijaMaksimum **implements**  Agregacija {  @Override  **public** **int** agregiraj(Komponenta komponenta) {  **int** rezultat = Integer.***MIN\_VALUE***;  **for** (**int** broj : komponenta) {  **if** (broj > rezultat)  rezultat = broj;  }  **return** rezultat;  }  } |

Sljedeći primjer prikazuje korištenje prethodno definiranih algoritama gdje se objekti konkretnih agregacija spremaju u varijable baznoga tipa Agregacija.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.prototip;  **public** **class** GlavniProgram {  **...**  **public** **static** **void** main(String[] args) {  Komponenta struktura = *stvoriStrukturu*();  Agregacija zbrojParnih = **new**  AgregacijaZbrojParnih();  Agregacija maksimum = **new** AgregacijaMaksimum();  System.***out***.println("Zbroj parnih elemenata je: "  + zbrojParnih.agregiraj(struktura));  System.***out***.println("Maksimum elemenata je: "  + maksimum.agregiraj(struktura));  }  } |
| Izlaz:  Zbroj parnih elemenata je: 20  Maksimum elemenata je: 8 |

Jedan od najčešćih načina korištenja ovog obrasca jest definiranje atributa tipa Agregacija koji može sadržavati bilo koju vrstu algoritma čime se dobiva **biranje strategije** unutar željenog razreda. Ovakav pristup potrebno je implementirati u sljedećoj vježbi.

Programski jezik *Java* implementira obrazac Strategija kroz brojna **funkcijska sučelja** iz paketa **java.util.function** koja su već spomenuta u prethodnim poglavljima. Tako je prethodni primjer moguće prepisati korištenjem sučelja **Function<T, R>**, kao što je vidljivo u sljedećem primjeru. Konkretne agregacije definirane su korištenjem **lambda izraza**.

|  |
| --- |
| **package** hr.unizg.srce.d470.obrasci.prototip;  **import** java.util.function.Function;  **public** **class** GlavniProgramFunction {  **...**  **public** **static** **void** main(String[] args) {  Komponenta struktura = *stvoriStrukturu*();  Function<Komponenta, Integer> zbrojParnih =  komponenta -> {  **int** rezultat = 0;  **for** (**int** broj : komponenta) {  **if** (broj % 2 == 0)  rezultat += broj;  }  **return** rezultat;  };  Function<Komponenta, Integer> maksimum =  komponenta -> {  **int** rezultat = Integer.***MIN\_VALUE***;  **for** (**int** broj : komponenta) {  **if** (broj > rezultat)  rezultat = broj;  }  **return** rezultat;  };  System.***out***.println("Zbroj parnih elemenata je: "  + zbrojParnih.apply(struktura));  System.***out***.println("Maksimum elemenata je: "  + maksimum.apply(struktura));  }  } |
| Izlaz:  Zbroj parnih elemenata je: 20  Maksimum elemenata je: 8 |

* 1. Vježba: Oblikovni obrasci

U ovoj vježbi nastavlja se implementacija igre ***ConnectX*** gdje je potrebno definirati **računalnog igrača** i osnovnu (slučajnu) **strategiju** koju takvi igrači mogu koristiti.

1. Stvorite apstraktni razred **Strategija** koji implementira sučelje BiFunction<Igra, Igrac, Potez>. Ovim razredom oblikuje se skup algoritama koji će za danu igru i igrača vratiti potez koji igrač treba odigrati.

* Napišite pomoćnu metodu **mogucnosti()** koja vraća listu valjanih poteza koje je moguće odigrati s danim žetonom na danoj ploči.
* Metodu **apply()** implementiranoga sučelja nije potrebno implementirati jer se to ostavlja konkretnim izvedenim razredima.

1. Stvorite razred **SlucajnaStrategija** koji će naslijediti prethodno definirani razred te implementirati metodu **apply()** na način da vrati nasumično odabrani **valjani potez**.
2. U razredu Igrac modificirajte metodu odluci() tako da kroz argument prima objekt tipa Igra. Ova promjena ima smisla jer bi svaki igrač trebao raspolagati informacijama o igri prilikom donošenja odluke o vlastitom potezu. Istu promjenu napravite i u izvedenom razredu LjudskiIgrac.
3. Implementirajte razred **RacunalniIgrac** koji nasljeđuje razred Igrac te koji dodatno sadrži atribut **strategija** tipa Strategija.

* Inicijalizirajte novi atribut preko konstruktora. Nemojte zaboraviti inicijalizirati i atribute baznoga razreda.
* Nadjačajte metodu **odluci()** na način da se poziva vlastita strategija te se vraća njezin rezultat.

1. U glavnom razredu ConnectX zamjenite jednoga postojećeg igrača s računalnim kako biste mogli isprobati rješenje.
   1. Pitanja za ponavljanje: Oblikovni obrasci
2. Što su oblikovni obrasci i čemu služe?
3. Navedite osnovne skupine oblikovnih obrazaca.
4. Nabrojite sudionike koji su definirani obrascem Kompozit.
5. Čemu služi obrazac Prototip?
6. Kako se u *Javi* implementira obrazac Iterator?
7. Kako biste oblikovali sustav koji omogućuje kriptiranje datoteka korištenjem različitih algoritama kao što su AES, 3DES ili RSA?

Završna vježba

Cilj završne vježbe jest kroz praktični primjer objediniti prethodno prikupljeno znanje objektno orijentiranoga programiranja u programskom jeziku *Java*.

U ovoj vježbi potrebno je implementirati generaliziranu varijantu igre ***Connect Four*** [13] pod nazivom ***ConnectX***. To je igra u kojoj se dva igrača izmjenjuju ubacivanjem žetona u vertikalno postavljenu ploču dimenzija ***NxM***. Žetoni se ubacuju u određeni stupac pri čemu padaju na dno stupca. Cilj igre je horizontalno, vertikalno ili dijagonalno skupiti ***X*** vlastitih žetona zaredom. Sljedeća slika [13] prikazuje igru *Connect Four* u kojoj je na ploči dimenzija *6 x 7* potrebno skupiti *4* žetona zaredom.



Kako bi igra bila zanimljivija, *ConnectX* koristi varijantu pravila ***PopOut*** [13] u kojoj igrač umjesto ubacivanja može izbaciti vlastiti žeton ako se on nalazi u prvom redu (tj. na dnu) određenoga stupca.

Važno je primijetiti da se u ovoj varijanti pravila izbacivanjem jednoga žetona može dogoditi da oba igrača pobjede u isto vrijeme. *ConnectX* takvu situaciju proglašava **remijem** odnosno neriješenim rezultatom partije.

Prije početka pisanja programa korisno je navesti sve elemente igre kako bismo lakše vizualizirali vrste objekata koji sudjeluju u igri:

* **Žeton** — igrači ga ubacuju u ploču, može biti različitih boja
* **Ploča** — sadrži žetone te podržava njihovo ubacivanje i izbacivanje
* **Potez** — vrsta akcije koju igrač može napraviti, tj. ubaciti ili izbaciti žeton
* **Stanje igre** — aktivna (u tijeku), pobjeda prvog igrača, pobjeda drugog igrača ili remi
* **Igrač** — čovjek ili računalo, sadrži ime i žetone koje koristi.
* **Strategija** — algoritam koji računalni igrač može koristiti
* **Igra** — sadrži informacije o jednoj partiji te kontrolira logiku i tijek jedne partije
* **Glavni program** — učitava informacije o igračima, pokreće igru te ispisuje rezultat partije.

Nekoliko vježbi prethodnih poglavlja već je postavilo temelje za razvoj ove igre pri čemu su mnogi razredi već implementirani. Završna vježba sastoji se od triju skupina zadataka koje najprije zaokružuju postojeće dijelove, a potom unaprijeđuju igru dodatnim funkcionalnostima. **Nakon svake skupine zadataka isprobajte program kako biste se uvjerili u točnost rješenja.**

Cilj prve skupine zadataka jest dovršiti osnovnu verziju igre ***ConnectX*** gdje dva igrača igraju jedan protiv drugoga pri čemu je dozvoljeno samo ubacivanje žetona.

1. U razredu Igra napišite metodu **stanjeIgre()** koja će provjeriti trenutačno stanje na ploči te vratiti jednu od vrijednosti enumeracije StanjeIgre.

* U metodi odigraj() nakon svakog odigranog poteza pozovite stanjeIgre() te prekinite igru u slučaju da je završila.

1. U razredu ConnectX prije stvaranja igre slučajnim odabirom odredite igrača koji će napraviti prvi potez.

Druga skupina zadataka uvodi ***PopOut*** varijantu igre pri čemu se dodaje mogućnost izbacivanja vlastitih žetona s dna stupaca.

1. Implementirajte razred **PotezIzbaci** koji nasljeđuje razred Potez.

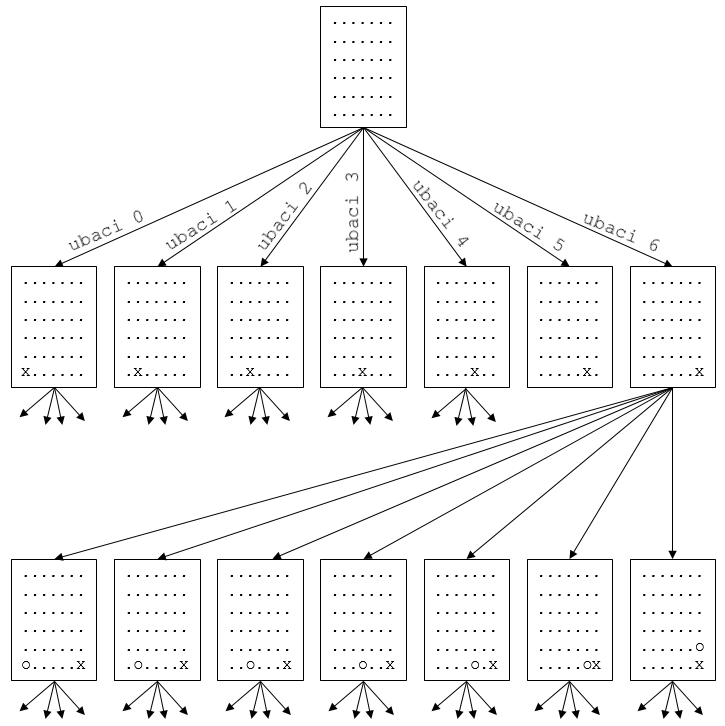
* Nadjačajte metode igraj(Ploca) i toString().

1. U razredu Ploca<T> dodajte metodu **izbaciElement(stupac)** koja izbacuje i vraća element s dna traženoga stupca.
2. U razredu LjudskiIgrac omogućite unošenje naredbe oblika ***„izbaci X“*** pri čemu je X indeks stupca iz kojeg treba izbaciti žeton.

* Napišite dodatnu provjeru za slučajeve kada učitani indeks nije u odgovarajućem rasponu vrijednosti. Također, provjerite je li moguće izbaciti žeton iz danog stupca. U slučaju greške, u obama slučajevima bacite iznimku tipa **IllegalArgumentException** s različitim porukama o grešci.

1. U razredu Strategija modificirajte metodu mogucnosti() kako biste uzeli u obzir mogućnost izbacivanja žetona iz ploče.

Posljednja skupina zadataka dodaje naprednu računalnu strategiju koja koristi algoritam **minimax** [14]. Ovaj algoritam koristi se u umjetnoj inteligenciji i teoriji igara kao pravilo odlučivanja koje **minimizira maksimalni gubitak** ili pak **maksimizira minimalnu dobit**. U kontekstu igre dvaju igrača, to znači da će algoritam **simulirati** igru i pritom uzimati u obzir najbolje poteze obaju igrača. U svakom potezu analizirat će se **sve mogućnosti** trenutnog igrača što rezultira velikim **prostorom pretraživanja** koje se može vizualizirati kao stablo. Sljedeća slika prikazuje početak stabla pretraživanja za primjer igre ***ConnectX***.



S obzirom na to da je prostor pretraživanja golem, nije praktično simulirati igru do samog završetka, već je potrebno **zaustaviti algoritam** na određenoj dubini te evaluirati trenutačno stanje na ploči. Za evaluiranje stanja na ploči mogu se koristiti sljedeće vrijednosti gdje veći broj označava povoljnije stanje.

* **4** – Pobjeda.
* **2** – Remi.
* **1** – Igra je još u tijeku.
* **0** – Poraz.

Iz prethodnih vrijednosti može se uočiti da će računalo radije odabrati potez koji vodi u remiziranu poziciju nego otići u smjeru gdje je rezultat neizvjesan. Postavljene vrijednosti mogu uvelike utjecati na strategiju — u ovom slučaju računalo će imati donekle **miroljubivu strategiju**.

U nastavku je prikazan **pseudokôd algoritma minimax** za igru *ConnectX*:

**Poboljšanja algoritma**

U ovoj vježbi prikazana je osnovna verzija algoritma minimax. U praksi postoje brojna poboljšanja algoritma poput **alfa-beta podrezivanja** koje može drastično smanjiti prostor pretraživanja čime se uvelike ubrzava algoritam. Za više detalja potrebno je pogledati dodatnu literaturu [14].

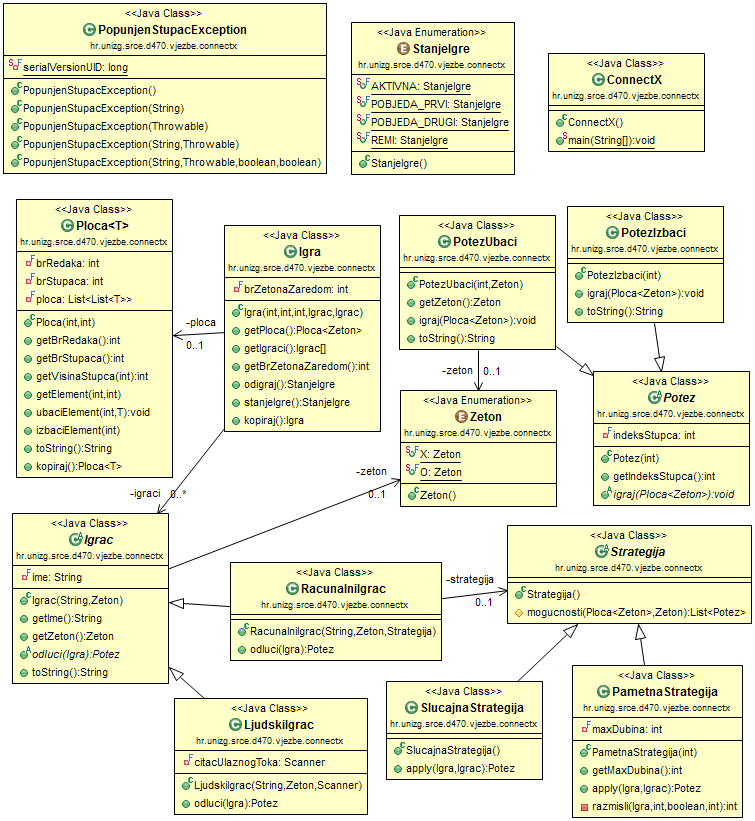
|  |
| --- |
| **minimax(igra, maksimiziraj, dubina):**  izracunaj stanje i evaluaciju igre  **if** dubina >= maxDubina ili je igra gotova:  **return** evaluacijaIgre  **if** maksimiziraj:  rezultat = -∞  **for** moguciPotez : mogucnosti():  kopiraj igru  odigraj potez na novoj ploci  rezultat = **max**(rezultat, **minimax**(novaIgra,  false,  dubina+1)  **return** rezultat  **else**:  rezultat = +∞  **for** moguciPotez : mogucnosti():  kopiraj igru  odigraj potez na novoj ploci  rezultat = **min**(rezultat, **minimax**(novaIgra,  true,  dubina+1)  **return** rezultat |

1. U razredima Ploca<T> i Igra implementirajte metodu **kopiraj()** koja će vratiti kopiju objekata tih razreda. Kopiranje je potrebno za pretraživanje jer nije praktično simulirati različite mogućnosti na istoj instanci ploče.
2. Stvorite razred **PametnaStrategija** koji će predstavljati strategiju koja koristi minimax algoritam. Dodajte atribut **maxDubina** koji predstavlja ograničenje dosega pretraživanja.

* Implementirajte metodu **apply()** tako da za svaki mogući potez pozove pomoćnu metodu **razmisli()** koja će korištenjem algoritma minimax vratiti evaluaciju poteza. Metoda apply() treba vratiti potez koji ima najveću evaluaciju. U slučaju više poteza s jednako velikom evaluacijom, potrebno je nasumično odabrati jedan od njih.
* Metodu **razmisli()** implementirajte po uzoru na pseudokôd algoritma minimax.

1. U glavnom razredu ConnectX zamjenite jednoga postojećeg igrača s računalnim kako biste mogli isprobati rješenje. Možete pokušati i druge kombinacije, poput sparivanja dvaju računalnih igrača koji koriste iste ili različite strategije.

U nastavku je prikazan dijagram razreda rješenja koji može poslužiti kao pomoć prilikom rješavanja vježbe.



Literatura

1. Čupić, M. Programiranje u Javi. 2015. URL: <http://java.zemris.fer.hr/nastava/opjj/book-2015-09-30.pdf>. 1.11.2020.
2. Java SE Specifications. URL: <https://docs.oracle.com/javase/specs/index.html>. 1.11.2020.
3. Java API (Java SE 14). URL: <https://docs.oracle.com/en/java/javase/14/docs/api/index.html>. 1.11.2020.
4. Java SE Downloads. URL: <https://www.oracle.com/java/technologies/javase-downloads.html>. 1.11.2020.
5. How to Write Doc Comments for the Javadoc Tool. URL:   
   <https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html>. 1.11.2020.
6. Help – Eclipse Platform. URL: <https://help.eclipse.org/>. 1.11.2020.
7. ObjectAid UML Explorer. URL: <https://www.objectaid.com/home>. 1.11.2020.
8. Predefined Annotation Types. URL: <https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html>. 1.11.2020.
9. Ross, T.; Amin, N. *Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers*. 2016.
10. Restrictions on Generics. URL: <https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html>. 1.11.2020.
11. Design Patterns. URL: <https://refactoring.guru/design-patterns>. 1.11.2020.
12. E. Gamma; R. Helm; R. Johnson; J. Vilssides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.
13. Connect Four. URL: <https://en.wikipedia.org/wiki/Connect_Four>. 1.11.2020.
14. S. J. Russell; P. Norvig. *Artificial Intelligence: A Modern Approach*. 2001.
15. Hruška, M. *Programiranje u Pythonu*, Osnovni tečaj Srca, verzija priručnika D460-20190904

***Bilješke:***

1. Tekst i prateća slika preuzeti iz tečaja „Programiranje u *Pythonu*“ [↑](#footnote-ref-1)
2. Tekst i prateća slika preuzeti iz tečaja „Programiranje u *Pythonu*“ [↑](#footnote-ref-2)