

# **Java od začetka**

**Luka Fürst**

2023

Univerza v Ljubljani  
Fakulteta za računalništvo in informatiko

Katalogni zapis o publikaciji (CIP) pripravili v Narodni in univerzitetni knjižnici  
v Ljubljani

COBISS.SI-ID=[144609027](#)

ISBN 978-961-7059-13-7 (pdf)

Copyright © 2023 Založba UL FRI. All rights reserved.

Elektronska izdaja knjige je na voljo na:

URL: <http://zalozba.fri.uni-lj.si/fuerst2023.pdf>

DOI: [10.51939/0003](#)

Recenzenta: prof. dr. Viljan Mahnič, prof. dr. Borut Robič

Založnik: Založba UL FRI, Ljubljana

Izdajatelj: UL Fakulteta za računalništvo in informatiko, Ljubljana

Urednik: prof. dr. Franc Solina

# Predgovor

O javi, programskem jeziku, ki že vrsto let kroji vrh lestvice najbolj priljubljenih,<sup>1</sup> obstaja ogromno literature, tako tiskane kot elektronske. Zakaj potemtakem potrebujemo še *eno* knjigo?

Prvič zato, ker je velika večina te literature angleške. Slovenci sicer premoremo nekaj knjig (npr. Mesojedec in Fabjan (2004); Mahnič in sod. (2008); Florjančič (2017)), a se povečini nanašajo na starejše različice jave. V času od svojega nastanka pa se je java gromozansko razvila in razširila. Njena knjižnica (Java Class Library) je v prvi različici obsegala dobrih dvesto razredov, sedaj pa jih več kot 4000. Tudi sam jezik se neprestano bogati: različica 5 nam je prinesla generike, različica 8 nas je razveselila z nekaterimi elementi funkcijskega programiranja, različica 16 prinaša konstrukt, ki avtomatizira izdelavo preprostih razredov ... V takšno bohotenje jo sili konkurenca, ki na področju programskih jezikov ni nič manj živahna kot v drugih vejah računalništva.

Knjiga, ki jo držite v rokah, je nastala tudi iz potrebe po zagotovitvi primerne literature za študente računalništva in informatike. Knjiga zajema snov predmeta Programiranje 1 v prvem letniku univerzitetnega študija na ljubljanski Fakulteti za računalništvo in informatiko. To seveda ne pomeni, da je primerna zgolj za omenjene študente; nasprotno, priporočam jo komurkoli, ki bi se želel naučiti programiranja v javi (ali programiranja nasploh), saj ne predpostavlja nikakršnega programerskega predznanja. Programiranja *nasploh*? Da, v večini priljubljenih programskih jezikov (C/C++, python, javascript ...) najdemo enake ali sorodne koncepte kot v javi.

Ker je java preobsežna, da bi lahko vse, kar ponuja, stlačili v razumno število strani, sem se hočeš nočeš moral omejiti. Nekaterim temam sem se povsem odrekel, nekaterih sem se le dotaknil. Kljub temu pa bi vas morala knjiga opremiti z dovolj znanja, da se boste brez pretiranih pretresov lahko poglobili v zahtevnejše programerske tehnike.

Česa se bomo naučili? V poglavju 1 si bomo pripravili potrebščine za programiranje in napisali svoj prvi program. V poglavju 2 bomo spoznali osnovne pojme programiranja, kot so izraz, spremenljivka, tip itd. Poglavje 3 bomo namenili krmilnim konstruktom, torej pogojnim stavkom, zankam in sorodnim rečem, poglavje 4 pa metodam. V poglavju 5 se bomo posvetili tabelam. V poglavju 6 se bomo sezna-

---

<sup>1</sup><https://www.tiobe.com/tiobe-index/>

nili z razredi in objekti. Sledijo poglavja o dedovanju (7), generikih (8) in vmesnikih (9). V poglavju 10 bomo govorili o vsebovalnikih, ki jih ponuja javina knjižnica, v poglavju 11 pa o konstruktih, imenovanih lambde.

Na podlagi tega seznama lahko sklepate, da bo poudarek predvsem na jeziku in programiranju, veliko manj pa na uporabi javine knjižnice. Omejili se bomo skoraj izključno na elemente paketov `java.lang` in `java.util`, le v poglavju 11 bomo pokukali še v paket `java.util.function`. Marsikateri jezikovni konstrukt bomo dokaj podrobno spoznali in ga pospremili z obilico primerov. Menim, da bodočim programerjem<sup>2</sup> na ta način postavim precej boljše temelje, kot bi jim, če bi zavoljo večjega poudarka na javini knjižnici varčeval pri jezikovnih konstruktih ali pripadajočih primerih.

Primeri so ponekod gonilna sila razlage. Na primer, v poglavju 7 bomo vse ključne koncepte dedovanja spoznali skozi dva primera. Toda tudi v poglavjih, ki so organizirana po konceptih, igrajo primeri osrednjo vlogo. Primere dopolnjujejo naloge, namenjene samostojnemu reševanju.

Vrstni red predstavitve konceptov se v nekaterih poglavjih namerno odmika od uveljavljenega. Na primer, notranje razrede obravnavamo v poglavju 9 (Vmesniki in notranji razredi), ne v poglavju 6 (Razredi in objekti), kamor bi jih najbrž umestila večina avtorjev. Vendar pa je po mojem mnenju včasih bolje počakati na trenutek, ko koncept resnično potrebujemo, četudi zato grešimo zoper načela sistematičnega podajanja snovi. Pri iteratorjih, ki jih predstavljamo v poglavju 9, se (nestatični in anonimni) notranji razredi resnično izkažejo za koristne, pred tem pa zanje ni prave motivacije. Podobno se zgodi pri tabeli javinih operatorjev, ki bi bolj sodila v poglavje 2 (Osnovni pojmi) kot v poglavje 3 (Krmilni konstrukti). Vendar pa težko govorimo o, denimo, pogojnem operatorju, dokler ne spoznamo koncepta pogojnega izvajanja.

Če bi radi poleg te knjige vzeli v roke še kakšno, vam priporočam Downey in Mayfield (2019), Burd (2017) ali Sierra in Bates (2005), če pa v javi že programirate in bi radi odkrili drobne trike izkušenih mačkov, se zagrizite v Bloch (2017) ali Evans in sod. (2018). Kdor želi spoznati elemente funkcijskega programiranja, ki jih je prinesla različica 8, naj se vrže v Urma in sod. (2014), kdor bi rad do potankosti dognal vse skrivnosti generikov in vsebovalnikov, ki nas v javi spremljajo že od različice 5, pa naj na svoj bralni seznam doda Naftalin in Wadler (2006). Če vas zanima »le« to, kaj vse java ponuja, pa bo bržčas najboljša izbira Schildt (2018), 1200-stranski referenčni priročnik, ki ga avtor redno posodablja. Na voljo je še ogromno drugih knjig, da o spletnih virih sploh ne govorimo.

Kaj pa zbirke nalog? V večini navedenih knjig boste našli tudi naloge, a vsaka od njih, tudi ta, ki jo držite v rokah, je prvenstveno učbenik. Obstajata, recimo, knjigi Kopec (2021) in Leonard (2019), a nobena od njiju ni primerna za začetnike, ki precej bolj kot vsebovalnike in lambde (v Leonard (2019) jih boste našli že pri prvi

---

<sup>2</sup>In programerkam. Naj izraz »programer« v nadaljevanju predstavlja oboje. Upam, da mi bralke tega ne bodo pretirano zamerile.

nalogi!) potrebujejo pogojne stavke in zanke. V tem trenutku lahko med zbirkami nalog priporočam le Mahnič in sod. (2008). Pa ne zato, ker sem soavtor, ampak zato, ker je, kot je videti, primerno zbirko nalog težko najti. Če me kdo postavi na laž, bom seznam virov z veseljem posodobil.

Vsi programi v tej knjigi so na voljo na sledeči spletni strani:

<http://ltpo2.frii.uni-lj.si/javaodzacetka/>

Če opazite napako ali pa če se vam bo zdelo, da bi lahko bilo v tej knjigi kaj bolje ali drugače, mi pišite na [luka.fuerst@frii.uni-lj.si](mailto:luka.fuerst@frii.uni-lj.si). Na odgovor boste morda morali malo počakati, na uresničitev svoje želje ali pobude pa ... najbrž še nekoliko dlje.

Java je neodvisna od operacijskega sistema, zato bi se morali vsi programi v tej knjigi prevesti in pognati povsod, kamor je mogoče naložiti javansko razvojno ogrodje (JDK — Java Development Kit). V poglavju 1, kjer bomo spoznali osnovne potrebščine za pisanje programov, pa bomo do neke mere kljub temu odvisni od okolja, saj bomo svoj prvi program prevedli in pognali v terminalu — programu, ki nam omogoča besedilno interakcijo z operacijskim sistemom. Omejil se bom na sistem Linux,<sup>3</sup> ki mi je kot dolgoletnemu uporabniku osebno najbližji, in sistem Windows, ki ga sicer slabo poznam, a ga zaradi vsesplošne razširjenosti skorajda moram vključiti. Uporabnikom drugih operacijskih sistemov (npr. Mac OS X) se opravičujem, a obstaja precejšnja verjetnost, da si boste lahko pomagali z navodili za Linux.

Rad bi se zahvalil vsem, ki so kakorkoli pripomogli k nastanku te knjige, še zlasti recenzentoma prof. Viljanu Mahniču in prof. Borutu Robiču ter založniku prof. Francu Solini. Prof. Mahnič me je kot soavtorja povabil k pisanju knjige Java skozi primere (Mahnič in sod., 2008); izkušnje, ki sem jih takrat pridobil, sem sedaj s pridom izkoristil in, upam, še nadgradil. Doc. Boštjanu Slivniku se zahvaljujem za številne debate na temo programiranja in z njim povezanih jezikovnih vprašanj, ki so mi pomagale razrešiti marsikatero dilemo. H kakovosti besedila so prispevali tudi študentje, ki jim predavam. Še posebej se zahvaljujem tistim, ki so odkrili katero od napak, ki so se mi izmuznile kljub večkratnemu natančnemu branju. Za koristne slovnično-slogovne nasvete se zahvaljujem Kristini.

Aha, liki iz profesorjevega kabineta ... Navdih zanje sem črpal deloma po Cormen in sod. (2009), legendarnem učbeniku algoritmike, v katerem ta ali oni profesor vsake toliko časa predlaga kak nepravilen ali neučinkovit, a kljub temu zanimiv algoritem, deloma pa po vrhunskem »neleposlovnem romanu« *Gödel, Escher, Bach* (Hofstadter, 1979), kjer dialogi med osrednjima protagonistoma mojstrsko dopolnjujejo stvarno besedilo.

Hvala še vsem tistim, ki sem jih morda pozabil navesti. Ni bilo namerno!

---

<sup>3</sup>Naziv *Linux* je prvotno predstavljal zgolj jedro operacijskega sistema, danes pa se nekoliko kri-  
vično (pozablja se na ključno vlogo projekta GNU) uporablja za celoten sistem. Na začetku se je  
sistem imenoval GNU/Linux.

Zadnji odstavek namenjam vsem, ki berete te vrstice, ne glede na to, ali bi radi spoznali java, ali bi se radi naučili programirati ali pa ste po knjigi posegli kar tako, brez kakšnega višjega cilja. Upam, da boste pri branju vsaj tako uživali, kot sem sam pri pisanju.

Luka Fürst

februarja 2023

# Kazalo

<b>1</b>	<b>Uvod</b>	<b>1</b>
1.1	Hiter in natančen, toda neumen . . . . .	1
1.2	Program, programski jezik, java . . . . .	1
1.3	Namestitev jave . . . . .	2
1.4	Terminal . . . . .	3
1.4.1	Zagon terminala . . . . .	3
1.4.2	Absolutne in relativne poti . . . . .	3
1.4.3	Najnujnejši ukazi . . . . .	5
1.4.4	Terminal in java . . . . .	7
1.5	Pisanje javanskih programov . . . . .	9
1.6	Prevajanje in poganjanje . . . . .	9
1.7	Prvi program . . . . .	9
1.8	Lepopisna pravila . . . . .	12
1.9	Komentarji . . . . .	13
1.10	Odpravljanje napak . . . . .	14
1.11	Povzetek . . . . .	16
<b>2</b>	<b>Osnovni pojmi</b>	<b>17</b>
2.1	Javanska konzola . . . . .	17
2.2	Samostojen program . . . . .	17
2.3	Števila in aritmetične operacije . . . . .	18
2.4	Izraz in vrednost . . . . .	19
2.5	Spremenljivka in prireditev . . . . .	20
2.6	Tip . . . . .	23
2.6.1	Številski tipi . . . . .	23
2.6.2	Izrecna pretvorba tipa . . . . .	25
2.6.3	Znaki . . . . .	27
2.6.4	Nizi . . . . .	29
2.6.5	Priredljivost tipov . . . . .	30
2.7	Stavek . . . . .	30
2.8	Bločna zgradba programa in doseg spremenljivk . . . . .	31
2.9	Vhod in izhod . . . . .	32

2.9.1	Standardni izhod . . . . .	32
2.9.2	Standardni vhod (in bežen pogled v pakete) . . . . .	33
2.10	Preverjanje programov . . . . .	35
2.10.1	Tipkovnica in zaslon . . . . .	36
2.10.2	Ukaz echo . . . . .	36
2.10.3	Preusmeritev vhoda . . . . .	37
2.10.4	Preusmeritev vhoda in izhoda . . . . .	37
2.10.5	Avtomatizacija preverjanja . . . . .	38
2.10.6	Preverjanje in dokazovanje . . . . .	40
2.11	Povzetek . . . . .	40
<b>3</b>	<b>Krmilni konstrukti</b>	<b>43</b>
3.1	Zaporedje stavkov . . . . .	43
3.2	Pogojni stavek . . . . .	43
3.2.1	if in if-else . . . . .	43
3.2.2	else if . . . . .	47
3.3	Logični izrazi . . . . .	49
3.4	Zanka <i>while</i> . . . . .	52
3.4.1	Sprotno seštevanje . . . . .	56
3.4.2	Vsota vseh vhodnih števil . . . . .	59
3.5	Sestavljeni prireditveni operatorji . . . . .	60
3.6	Zanka <i>do</i> . . . . .	61
3.6.1	Ugibanje naključnega števila . . . . .	64
3.7	Zanka <i>for</i> . . . . .	67
3.7.1	Sprehod po (angleški) abecedi . . . . .	70
3.7.2	Iskanje maksimuma . . . . .	71
3.7.3	Praštevila . . . . .	73
3.7.4	Tabela množkov brez poravnave . . . . .	75
3.8	Stavek <code>System.out.printf(...)</code> . . . . .	76
3.8.1	Tabela množkov s poravnavo . . . . .	78
3.8.2	Izpis parov . . . . .	78
3.9	Stavka <i>break</i> in <i>continue</i> . . . . .	80
3.9.1	Stavek <i>break</i> . . . . .	80
3.9.2	Stavek <i>continue</i> . . . . .	83
3.9.3	Praštevila učinkoviteje . . . . .	84
3.9.4	Pitagorejska števila . . . . .	88
3.10	Stavek <i>switch</i> . . . . .	90
3.11	Pogojni operator . . . . .	95
3.12	Delo z biti . . . . .	96
3.12.1	Bitni zapis celih števil . . . . .	96
3.12.2	Bitni operatorji . . . . .	97
3.13	Pregled operatorjev . . . . .	99



3.14 Povzetek . . . . .	101
<b>4 Metode . . . . .</b>	<b>103</b>
4.1 Metoda in klic . . . . .	103
4.2 Parametri metode . . . . .	105
4.3 Vračanje vrednosti . . . . .	112
4.4 Metode in podproblemi . . . . .	115
4.4.1 (Že spet) praštevila . . . . .	115
4.4.2 Prijateljska števila . . . . .	116
4.4.3 Pitagorejska števila . . . . .	118
4.5 Rekurzija . . . . .	119
4.6 Povzetek . . . . .	122
<b>5 Tabele . . . . .</b>	<b>125</b>
5.1 Motivacijski primer . . . . .	125
5.2 Tabela . . . . .	125
5.3 Dostop do elementov tabele . . . . .	126
5.4 Izdelava tabele . . . . .	127
5.5 Sprehod po tabeli . . . . .	128
5.5.1 Zanki <i>for</i> in <i>for-each</i> . . . . .	128
5.5.2 Vsota in maksimum . . . . .	129
5.6 Uporaba tabel . . . . .	131
5.6.1 Rešitev motivacijskega primera . . . . .	131
5.6.2 Pogostost ocen . . . . .	132
5.6.3 (Še zadnjič) praštevila . . . . .	135
5.7 Iskanje v urejeni tabeli . . . . .	137
5.8 Urejanje tabel . . . . .	140
5.9 Memoizacija . . . . .	142
5.10 Primitivni in referenčni tipi . . . . .	145
5.11 Dvodimenzionalne tabele . . . . .	150
5.11.1 Izdelava in dostop . . . . .	151
5.11.2 Sprehod . . . . .	154
5.11.3 Najcenejše poti v grafu . . . . .	157
5.12 Tridimenzionalne tabele . . . . .	162
5.13 Povzetek . . . . .	165
<b>6 Razredi in objekti . . . . .</b>	<b>167</b>
6.1 Razredi, objekti, atributi . . . . .	167
6.2 Dostopna določila . . . . .	173
6.3 Izdelava objekta in konstruktor . . . . .	175
6.4 Metode . . . . .	177
6.4.1 Statične in nestatične metode . . . . .	177

6.4.2	Metode za vračanje vrednosti atributov (»getterji«)	181
6.4.3	Metode za nastavljanje vrednosti atributov (»setterji«)	182
6.4.4	(Ne)spremenljivost	183
6.4.5	Metode za izpis vsebine in vračanje vsebine v obliki niza	184
6.4.6	Metode za primerjanje objektov	185
6.4.7	Izpuščanje besede <code>this</code>	188
6.5	Primer uporabe razreda	189
6.6	Statični atributi	190
6.7	Več metod z istim imenom	193
6.8	Več konstruktorjev	194
6.9	Razred <code>String</code>	195
6.10	Razred za predstavitev vektorja	198
6.10.1	Razred <code>VektorInt</code>	198
6.10.2	Razred <code>VektorString</code>	203
6.10.3	Primer uporabe	205
6.11	Povzetek	207
<b>7</b>	<b>Dedovanje</b>	<b>211</b>
7.1	Uvod	211
7.2	Hierarhija študentov	212
7.2.1	Nadrazred	212
7.2.2	Podrazred	213
7.2.3	Konstruktor podrazreda	214
7.2.4	Redefinicija metode	216
7.2.5	Tip kazalca in tip objekta	219
7.2.6	Dedovanje in tabele	223
7.3	Hierarhija geometrijskih likov	225
7.3.1	Načrt hierarhije	225
7.3.2	Razred <code>Lik</code>	227
7.3.3	Razred <code>Pravokotnik</code>	229
7.3.4	Razred <code>Kvadrat</code>	230
7.3.5	Razred <code>Krog</code>	232
7.3.6	Primer klica metode v hierarhiji	233
7.3.7	Testni razred	233
7.3.8	Operator <code>instanceof</code> in izrecna pretvorba tipa	235
7.3.9	Kovariantnost	239
7.4	Razred <code>Object</code>	241
7.4.1	Metoda <code>toString</code>	242
7.4.2	Metoda <code>equals</code>	242
7.4.3	Metoda <code>hashCode</code>	244
7.4.4	Razred <code>Object</code> in tabele	246
7.5	Ovojni tipi	246

7.6	Vektor z elementi poljubnih referenčnih tipov . . . . .	251
7.7	Slovar . . . . .	254
7.7.1	Naivna implementacija slovarja . . . . .	255
7.7.2	Implementacija slovarja z zgoščeno tabelo . . . . .	256
7.7.3	Primer uporabe slovarja . . . . .	262
7.8	Povzetek . . . . .	264
<b>8</b>	<b>Generiki</b>	<b>269</b>
8.1	Uvod . . . . .	269
8.2	Generični razredi . . . . .	272
8.3	Generične metode . . . . .	273
8.4	Zgornja meja tipnega parametra . . . . .	274
8.5	Omejitve pri uporabi generikov . . . . .	276
8.6	Generični razred Vektor . . . . .	277
8.7	Generični razred Slovar . . . . .	280
8.8	Nadomestni znak . . . . .	282
8.9	Povzetek . . . . .	285
<b>9</b>	<b>Vmesniki in notranji razredi</b>	<b>289</b>
9.1	Opredelitev in uporaba . . . . .	289
9.2	Vmesnik Comparable in naravna urejenost . . . . .	292
9.3	Vmesnik Comparator . . . . .	296
9.4	Vmesnik Iterator . . . . .	299
9.4.1	Iterator nad vektorjem . . . . .	300
9.4.2	Iterator nad slovarjem . . . . .	303
9.5	Vmesnik Iterable . . . . .	307
9.6	Notranji razredi . . . . .	309
9.6.1	Statični notranji razredi . . . . .	309
9.6.2	Nestatični notranji razredi . . . . .	310
9.6.3	Anonimni notranji razredi . . . . .	312
9.7	Povzetek . . . . .	316
<b>10</b>	<b>Vsebovalniki</b>	<b>319</b>
10.1	Pregled . . . . .	319
10.2	Hierarhija vsebovalnikov . . . . .	320
10.3	Vmesnik Collection . . . . .	321
10.4	Seznami . . . . .	324
10.4.1	Vmesnik List . . . . .	324
10.4.2	Razreda ArrayList in LinkedList . . . . .	325
10.5	Množice . . . . .	329
10.5.1	Vmesnik Set . . . . .	330
10.5.2	Razred HashSet . . . . .	330

10.5.3 Razred TreeSet . . . . .	333
10.5.4 Uporaba množic . . . . .	338
10.6 Slovarji . . . . .	342
10.6.1 Vmesnik Map . . . . .	343
10.6.2 Razreda HashMap in TreeMap . . . . .	345
10.6.3 Uporaba slovarjev . . . . .	348
10.7 Razred Collections . . . . .	357
10.8 Povzetek . . . . .	359
<b>11 Lambde</b>	<b>361</b>
11.1 Od samostojnega do anonimnega razreda . . . . .	361
11.2 Lambda . . . . .	363
11.3 Vrednost in tip lambde . . . . .	364
11.4 Nadaljnje poenostavitve . . . . .	368
11.5 Vgrajeni funkcijski vmesniki . . . . .	369
11.6 Lambde in lokalne spremenljivke . . . . .	372
11.7 Lambde in zbirke . . . . .	373
11.7.1 Štetje elementov, ki izpolnjujejo pogoj . . . . .	373
11.7.2 Izvedba opravlja za vsak element zbirke . . . . .	375
11.7.3 Združevanje elementov z dvojiškim operatorjem . . . . .	376
11.7.4 Grupiranje elementov po rezultatih funkcije . . . . .	377
11.8 Primer: obdelava rezultatov izpita . . . . .	379
11.8.1 Vhod in izhod . . . . .	379
11.8.2 Razred Student . . . . .	382
11.8.3 Branje . . . . .	382
11.8.4 Urejanje . . . . .	387
11.8.5 Izpis . . . . .	388
11.9 Povzetek . . . . .	391
<b>Literatura</b>	<b>395</b>
<b>Stvarno kazalo</b>	<b>397</b>

# 1 Uvod

V tem poglavju se bomo seznanili z vsem, kar potrebujemo, da napišemo svoj prvi javanski program ter ga uspešno prevedemo in poženemo.

## 1.1 Hiter in natančen, toda neumen

Računalnik je neverjetno hiter. V sekundi lahko izvrši več milijard (!) ukazov. No, ne ravno ukazov tipa »predvajaj film« ali »uredi preglednico po priimkih«. Gre za preproste *procesorske ukaze*, kot so, denimo, »seštej dve števili«, »preveri, ali je število pozitivno« in »skoči na nek drug ukaz«. Kljub temu pa se boste gotovo strinjali, da je taka hitrost zavidljiva. Kako si lahko predstavljamo *milijardinko* sekunde, če še stotinkam ne moremo slediti?

Računalnik je neverjetno natančen. Isti ukaz bo vedno izvršil na enak način. Nikoli se ne bo zmotil. Nikoli ne bo utrujen, muhast ali brezvoljen. Napake v delovanju programov so posledica človekovih, ne računalnikovih hib.

Žal — ali pa morda na srečo — pa je računalnik tudi neverjetno neumen. Vsa »inteligenca«, ki jo navidez premore, je zgolj in samo plod človekovih možganov. Programe za igranje šaha, priporočanje »prijateljev« na družabnem omrežju in upravljanje vesoljskih ladij je napisal človek. Računalnik jih zgolj pokorno izvaja.

Ni povsem znano, kdo je avtor sledečega navedka, vsekakor pa drži kot pribit:

Računalnik je hiter in natančen, toda neumen. Človek pa je počasen in površen, toda inteligenčen. Skupaj lahko dosežeta čudeže.

V tej knjigi žal ne bo čudežev, boste pa spoznali osnove pisanja programov oziroma *programiranja*, večšine, ki vas bo morda pripeljala do velikih dosežkov.

## 1.2 Program, programski jezik, java

Program ni nič drugega kot besedilna datoteka z navodili, ki jih računalnik lahko »razume« in izvrši. Programov žal ne moremo pisati v običajni slovenščini ali angleščini. Pišemo jih v *programskih jezikih*, ki so v nasprotju z naravnimi povsem natančno definirani in nedvoumni. Na primer, v slovenščini zlahka tvorimo stavke, kot je »Kosilo je vrgel psu, ta ga ni maral, zato ga je brcnil«, programski jeziki pa takšnih dvoumnosti

(kdo je koga brcnil?) ne smejo omogočati. Vsak element programskega jezika mora imeti enolično določen pomen.

Med stotinami programskih jezikov, ki so programerju danes na voljo, bomo izbrali jezik po imenu *java*. Java (letnik 1996) je podobna jezikoma C (Kernighan in Ritchie, 1988) in C++ (Stroustrup, 2013), ki podobo programerske krajine krojita že od leta 1972 (C) oziroma 1985 (C++), a je varnejša in — kljub nenehnim izboljšavam in dopolnitvam — enostavnejša od njiju. Njena prednost pred čedalje bolj priljubljenimi skriptnimi jeziki, kot sta python (Lutz, 2013) in javascript (Flanagan, 2020), pa se pokaže pri obsežnejših programih in tistih, ki se izvajajo dlje časa. Po avtorjevih izkušnjah so skriptni jeziki nadvse pripravi, ko mora, denimo, obdelati rezultate izpita, pri kompleksnejših programih pa kaj kmalu prične pogrešati javino strogost in strukturiranost. Pri programih z daljšim časom izvajanja pride do izraza javina hitrost, pa tudi prevajalnik, ki nas na zatipkanino opozori, še preden program poženemo. Avtorju se je že zgodilo (in to ne le enkrat!), da je program, napisan v skriptnem jeziku, pustil teči čez noč, zjutraj pa je zaprepaden ugotovil, da je zaradi napake v vrstici, ki naj bi rezultate varno shranila v datoteko, vse izgubil.

V nasprotju s programi v skriptnih jezikih javanskih programov ne moremo neposredno pognati, ampak jih moramo s pomočjo programa, ki mu pravimo *prevajalnik*, najprej prevesti v obliko, prikladno za izvajanje. Za razliko od jezikov C in C++, pri katerih se program prevede neposredno v procesorske ukaze, se javanski program prevede v nekakšno vmesno obliko — jezik *javanskega navideznega stroja* (JVM — Java Virtual Machine). Programi, ki jih tvori javanski prevajalnik, so zato za odtenek počasnejši od tistih, ki jih tvori prevajalnik za jezik C ali C++, vendar pa so bistveno bolj prenosljivi. Program, sestavljen iz procesorskih ukazov, je namreč vezan na določen tip procesorja, program za javanski navidezni stroj pa lahko poganjamo na kateremkoli računalniku, na katerem je naložen sistem za izvajanje takih programov (JRE — Java Runtime Environment).

### 1.3 Namestitev java

Danes lahko javanske programe pišemo in poganjamo kar v brskalniku, kljub temu pa je java udobneje namestiti na lasten računalnik. V nekaterih operacijskih sistemih lahko java namestimo kar s pomočjo vgrajenega upravljalnika paketov. V Linuxovi distribuciji Ubuntu, denimo, s kombinacijo tipk Ctrl-Alt-T odpremo terminal in odtipkamo

```
terminal> sudo apt install openjdk-N-jdk
```

pri čemer je *N* številka različice.<sup>1</sup> Če te možnosti nimamo, pa lahko java namestimo

<sup>1</sup>Besedila `terminal>` ne odtipkamo, saj zgolj označuje, da ukaz vnesemo v terminalu operacijskega sistema.

s spletne strani podjetja Oracle.<sup>2</sup> Izberemo najnovejšo različico jave SE (ne EE ali ME) in sistem JDK (ne JRE). Po nekaj klikih se java namesti na naš računalnik.

## 1.4 Terminal

V tej knjigi bomo programe prevajali in poganjali v *terminalu operacijskega sistema*. Gre za program, ki nam omogoča neposreden vnos ukazov za zagon programov, za delo z datotekami in imeniki, za nadzor procesov itd.<sup>3</sup> Terminal je bil dolgo časa edino sredstvo za komunikacijo z računalnikom, danes pa ga redkokdo pozna, čeprav lahko v njem marsikatero opravilo izvršimo bistveno hitreje in učinkoviteje kot v grafičnem vmesniku. S terminalom se bomo v tej knjigi ukvarjali zgolj toliko, da bomo lahko prevajali in poganjali javanske programe, če bi ga radi podrobneje spoznali, pa se poglobite v katerega od številnih virov (npr. Shotts (2019)).

### 1.4.1 Zagon terminala

V sistemu Linux se terminal (angl. terminal) lahko imenuje tudi *konzola* (angl. console), sistem Windows pa uporablja naziv *ukazni poziv* (angl. command prompt). V Linuxovi distribuciji Ubuntu lahko terminal odpremo s kombinacijo tipk Ctrl-Alt-T, v novjših različicah sistema Windows pa lahko na začetnem zaslonu odtipkamo cmd in pritisnemo tipko Enter. V obeh primerih se bo odprlo prav nič ugledno črno okno. V distribuciji Ubuntu bo po privzetih nastavitvah prikazovalo takšno vrstico:<sup>4</sup>

```
uporabnik@računalnik:~$
```

Tej vrstici pravimo *pozivnik*. *Lupina* — poseben program, ki v ozadju bere in izvršuje ukaze, ki jih vnašamo v terminal — sedaj od nas pričakuje, da bomo vnesli ukaz. *Lupina* bo ukaz izvršila in spet prikazala pozivnik. Kako pa je pozivnik sestavljen? Beseda *uporabnik* predstavlja uporabniško ime, beseda *računalnik* pa ime računalnika. Znak `:` je zgolj ločilo, znak `$` pa pove, da gre za običajni (neprivilegirani) način vnosa ukazov. Med dvopičjem in dolarskim znakom je podana absolutna pot do trenutnega imenika. *Absolutna pot? Trenutni imenik? Čas je za nov razdelek ...*

### 1.4.2 Absolutne in relativne poti

Kot najbrž veste, je datotečni sistem sestavljen iz *imenikov* in *datotek*. Imeniki lahko vsebujejo datoteke in druge imenike, vsaka datoteka pa vsebuje nek dokument, sliko, program ali še marsikaj drugega. Imeniki in datoteke so hierarhično organizirani. Na vrhu hierarhije je *korenski imenik* (`/` v sistemu Linux oziroma `C:\` na razdelku

<sup>2</sup><https://www.oracle.com/java/technologies/javase-downloads.html>

<sup>3</sup>*Imenik* (angl. directory) se v nekaterih operacijskih sistemih imenuje tudi *mapa* (angl. folder).

<sup>4</sup>V drugih Linuxovih distribucijah lahko ta vrstica izgleda drugače.

C v sistemu Windows), nato pa sledijo podimeniki korenskega imenika (npr. /home oziroma C:\Users), podimeniki teh podimenikov (npr. /home/uporabnik oziroma C:\Users\uporabnik) itd. Vsak imenik in vsaka datoteka ima svojo *absolutno pot*. Ta pove, kako pridemo do imenika oziroma datoteke, če pričnemo v korenskem imeniku. Na primer, v sistemu Linux nam absolutna pot

```
/home/uporabnik/programi/java/MojPrvi.java
```

pove, da se iz korenskega imenika premaknemo v njegov podimenik home, nato vstopimo v podimenik uporabnik imenika /home, sledi premik v podimenik programi in zatem še v njegov podimenik java, v tem imeniku pa izberemo datoteko MojPrvi.java.

Poleg absolutnih poznamo tudi *relativne* poti. Te so odvisne od *trenutnega imenika* — imenika, v katerem se »nahajamo«, torej imenika, ki je naveden v pozivniku. Če se absolutna pot do trenutnega imenika (v sistemu Linux) glasi

```
/home/uporabnik/programi
```

potem je relativna pot do datoteke z absolutno potjo

```
/home/uporabnik/programi/java/MojPrvi.java
```

enaka

```
java/MojPrvi.java
```

(»v trenutnem imeniku poišči podimenik java, tam pa datoteko MojPrvi.java«), relativna pot do imenika z absolutno potjo

```
/home/uporabnik/Glasba/ljudska
```

pa se glasi

```
../Glasba/ljudska
```

in jo lahko preberemo kot »pojdi v nadimenik trenutnega imenika (to namreč pomeni zapis ..), od tam pa v podimenik Glasba in nato še v njegov podimenik ljudska«. Absolutna pot se v sistemu Linux vedno prične z znakom / ali ~ (njegov pomen bomo spoznali v kratkem), relativna pa nikoli.

Kot smo povedali, je absolutna pot do trenutnega imenika v Linuxovem pozivniku stlačena med znaka : in \$. V primeru pozivnika

```
uporabnik@računalnik:~$
```

je ta pot torej enaka ~. Tilda, kot se ta znak imenuje, je zgolj okrajšava za absolutno pot do *domačega imenika* uporabnika, ki je v sistem trenutno prijavljen. Domači imenik je izhodišče hierarhije imenikov in datotek, ki pripadajo uporabniku. V



sistemu Linux se absolutna pot do domačega imenika uporabnika *uporabnik* glasi */home/uporabnik*. Ko poženemo terminal, je, kot smo videli, trenutni imenik enak domačemu.

Znak *~* se torej nadomesti s potjo */home/uporabnik*. Namesto

```
/home/uporabnik/programi/java/MojPrvi.java
```

lahko potemtakem pišemo

```
~/programi/java/MojPrvi.java
```

Na trenutni imenik se ne nanašajo samo relativne poti, ampak tudi številni ukazi, če jih izvršimo brez argumentov. Na primer, če ima trenutni imenik absolutno pot *~/programi*, potem je ukaz

```
ls
```

enakovreden ukazu

```
ls ~/programi
```

in tudi ukazu

```
ls .
```

Znak *.* predstavlja relativno pot do trenutnega imenika, ukaz *ls* pa izpiše vsebino podanega (ali trenutnega) imenika.

V sistemu Windows so razmere podobne, a vseeno drugačne. Ko odpremo terminal, dobimo tak pozivnik:

```
C:\Users\uporabnik>
```

Znak *>* nima posebne vloge, preostanek pa je absolutna pot do trenutnega imenika, ki je na začetku prav tako enak domačemu. Absolutna pot do domačega imenika je v sistemu Windows potemtakem enaka *C:\Users\uporabnik*, v nasprotju s sistemom Linux pa je ne moremo okrajšati z znakom *~*. Kot ločilo med elementi poti v sistemu Windows služi znak *\*, ne znak */*, absolutna pot do korenskega imenika pa se glasi *C:\*, ne */*. Zapis *C:* opredeljuje trenutni razdelek na disku (C). V Linuxu oznaka razdelka ni sestavni del absolutne poti, vendar pa to ne pomeni, da razdelkov tam ne moremo imeti, le uporabljamo jih nekoliko drugače (datotečni sistemi posameznih razdelkov so priklopljeni na določene imenike v krovnem datotečnem sistemu).

### 1.4.3 Najnujnejši ukazi

V terminalu imamo na voljo celo vrsto ukazov za delo z datotekami in imeniki, pa tudi za nekatere druge reči. Ogledali si bomo ukaza za spreminjanje trenutnega

imenika in izpis vsebine imenika, še nekaj pa jih bomo le našteli.

Trenutni imenik spremenimo z ukazom `cd` (v obeh sistemih). Ukaz poženemo z vrstico

```
terminal> cd pot
```

pri čemer je *pot* absolutna ali relativna pot do ciljnega imenika. Na primer, če se trenutno nahajamo v domačem imeniku, potem nas vrstica

```
terminal> cd programi/java
```

v Linuxu oziroma

```
terminal> cd programi\java
```

v sistemu Windows prestavi v imenik `/home/uporabnik/programi/java` oziroma `C:\Users\uporabnik\programi\java`. Ustrezno se spremeni tudi pozivnik:

```
uporabnik@računalnik:~/programi/java$
```

oziroma

```
C:\Users\uporabnik\programi\java>
```

Če sedaj želimo trenutni imenik spremeniti v `/home/uporabnik/Glasba/domaca` oziroma `C:\Users\uporabnik\Glasba\domaca`, potem v Linuxu poženemo

```
terminal> cd ../../Glasba/domaca
```

ali

```
terminal> cd ~/Glasba/domaca
```

V sistemu Windows moramo znak `/` zamenjati z znakom `\`, tildo pa z besedilom `C:\Users\uporabnik` ali kar `\Users\uporabnik`, če se oznaka našega trenutnega razdelka glasi `C`.

Če absolutna pot do trenutnega imenika ni razvidna iz pozivnika, jo lahko v Linuxu izpišemo z ukazom `pwd`, v sistemu Windows pa uporabimo ukaz `cd` brez argumenta.

Z ukazom `ls` (Linux) oziroma `dir` (Windows) izpišemo vsebino imenika. Ukaz uporabimo na enak način kot ukaz `cd` (kot argument podamo absolutno ali relativno pot do imenika), če ukaz poženemo brez argumenta, pa se izpiše vsebina trenutnega imenika. Ukaza imata številna stikala; v Linuxu, na primer, ukaz pogosto poženemo kot

```
terminal> ls -l
```

in dobimo izpis, ki ne podaja zgolj imen vsebovanih imenikov in datotek, ampak tudi številne druge podatke (lastnika, velikost, pravice dostopa, datum in čas zadnje spremembe, pa še kaj).

V tabeli 1.1 so nanizani še nekateri drugi ukazi, ki jih lahko uporabljamo v terminalu.

**Tabela 1.1** Nekaj ukazov, ki jih lahko uporabljamo v terminalu.

Pomen ukaza	Linux	Windows
Izpiši vsebino imenika	ls	dir
Zamenjaj trenutni imenik	cd	cd
Prikaži trenutni imenik	pwd	cd (brez argumenta)
Ustvari prazen imenik	mkdir	mkdir ali md
Kopiraj datoteke oz. imenike	cp	copy
Premakni datoteke oz. imenike	mv	move
Izbriši datoteke oz. imenike	rm	del
Izbriši prazen imenik	rmdir	rmdir ali rd
Izpiši vsebino datoteke	cat	type
Primerjaj datoteki	diff	fc
Počisti okno terminala	clear	cls
Izpiši navodila za uporabo ukaza	man in help	help

#### 1.4.4 Terminal in java

Javanske programe bomo prevajali in poganjali v terminalu. To sicer ni edina možnost, je pa enostavna in obenem fleksibilna. Kot bomo videli v razdelku 1.6, bomo uporabljali programa `javac` in `java`; s prvim bomo naš program prevedli, z drugim pa izvedli. Da ju bomo lahko poganjali v poljubnem trenutnem imeniku, bomo vsaj v sistemu Windows morali lupini, ki teče v ozadju, povedati, kje naj ju najde.

Tako v sistemu Linux kot v sistemu Windows obstaja systemska spremenljivka (*spremenljivka okolja*, angl. *environment variable*) po imenu `PATH`, ki vsebuje z dvo-pičjem (Linux) oziroma podpičjem (Windows) ločen seznam absolutnih poti do imenikov. Ko lupina ugotovi, da želi uporabnik pognati program (npr. `javac`), preveri, ali se program nahaja v katerem od teh imenikov. Če se, požene prvega, ki ga najde, v nasprotnem primeru pa sporoči, da programa ni našla. No, lupina v sistemu Windows pred imeniki v spremenljivki `PATH` pregleda še trenutni imenik.

Če javo namestimo v sistemu Linux, se bosta datoteki `javac` in `java` ali bližnjici do njiju po vsej verjetnosti namestili v enega od imenikov, ki je po privzetih nastavitvah že naveden v spremenljivki `PATH` (npr. `/usr/bin` ali `/usr/local/bin`). Zato

nam v Linuxu ponavadi ni treba storiti ničesar. V to se prepričamo tako, da v terminal vtipkamo

```
terminal> javac
```

in nato še

```
terminal> java
```

Če dobimo izpis pomoči programa `javac` oziroma `java`, je vse v redu. Če pa nam lupina sporoči, da programov ne najde, poiščemo imenik, ki vsebuje programa `javac` in `java`, in absolutno pot do njega (označimo jo kot *pot*) dodamo v spremenljivko `PATH`. To storimo tako, da na konec datoteke `~/ .bashrc` dodamo vrstico<sup>5</sup>

```
PATH=pot:$PATH
```

ki povzroči, da se na začetek seznama poti v spremenljivki `PATH` doda pot do imenika s programoma `javac` in `java` (zapis `$PATH` pomeni trenutno vrednost spremenljivke `PATH`).

Če ne določimo drugače, se v okolju Windows programa `javac` in `java` (datoteki `javac.exe` in `java.exe`) namestita v imenik

```
C:\Program Files\Java\jdkN\bin
```

Ker tega imenika skoraj gotovo ne bomo našli v spremenljivki `PATH`, ga moramo tja dodati. Odpremo okno za urejanje spremenljivk okolja (v novejših različicah sistema Windows ga priključimo enostavno tako, da v iskalno polje na začetnem zaslonu vnesemo `env`, v vsakem primeru pa bomo lahko do njega prišli prek okna `System (System)` oziroma `Control Panel (Nadzorna plošča)` in gumba `Advanced system settings (Napredne možnosti)`) in gornjo pot dodamo na začetek seznama poti v spremenljivki `PATH`. Nato zapremo in ponovno odpremo terminal in vanj vtipkamo

```
terminal> javac
```

in

```
terminal> java
```

Če se izpišejo navodila za uporabo teh programov, smo spremenljivko `PATH` uspešno nastavili.

---

<sup>5</sup>Znakov = in : ne smemo obdati s presledki!

## 1.5 Pisanje javanskih programov

Javanski program ni nič drugega kot skupek besedilnih datotek. Dolgo časa bodo naši programi sestavljeni celo iz ene same datoteke. Za pisanje javanskih programov bi zato načeloma lahko uporabili poljuben urejevalnik besedil, ki dokumente shranjuje v čisti besedilni obliki. Vizualni urejevalniki (Microsoft Word, LibreOffice Writer itd.) tako ne pridejo v poštev, lahko pa bi programirali v navadni »beležnici« (Notepad). Čeprav ni s tem nič narobe, pa kljub temu raje uporabljamo programerjem prijaznejše urejevalnike, kot so gedit, Vim ali Emacs v sistemu Linux (začetnikom priporočamo samo prvega), Notepad++ v sistemu Windows ter Visual Studio Code ali Sublime Text na poljubnem operacijskem sistemu.

Nekateri urejevalniki, pravimo jim (integrirana) razvojna orodja (IDE — Integrated Development Environment), omogočajo poleg pisanja programov vsaj še njihovo prevajanje in poganjanje. Čeprav nam razvojna orodja pri obsežnejših programih pridejo zelo prav, jih začetnikom ne priporočamo, saj pogosto zameglijo proces izdelave, prevajanja in poganjanja programov. V številnih razvojnih orodjih moramo namreč tudi za najenostavnejše programe ustvariti t. i. projekt (koncept, ki je povezan z orodjem, ne z javo), orodje pa nato izdelava razvejano hierarhijo imenikov in datotek, tudi če naš program obsega eno samo datoteko.

## 1.6 Prevajanje in poganjanje

Programu v izvornem programskem jeziku (npr. v javi) pravimo tudi *izvorna koda* ali preprosto *koda*. Omenili smo že, da javanske izvorne kode ne moremo neposredno pognati, ampak jo moramo najprej s pomočjo prevajalnika prevesti v jezik javanskega navideznega stroja. Prevajalnik preveri, ali je naš program skladen z javinimi sintaktičnimi in semantičnimi pravili. Če je, izdelava datoteko s prevedenim programom, v nasprotnem primeru pa izpiše seznam napak in ne tvori prevoda. Prevedeni program lahko nato poženemo z *izvajalnikom*. Izvajalnik »razume« ukaze javanskega navideznega stroja in jih lahko neposredno izvrši.

## 1.7 Prvi program

Čas je, da napišemo svoj prvi program. Tradicija velevala, da ta program na zaslon izpiše besedilo »Pozdravljen, svet!«<sup>6</sup> Pojdimo po vrsti:

- Odprimo programerski urejevalnik in odtipkajmo sledečo kodo (brez skrbi, vse bo še jasno):<sup>7</sup>

<sup>6</sup>V izvorniku »Hello, world!«.

<sup>7</sup>V arhivu ZIP, ki se nahaja na spletni strani <http://ltpo2.frii.uni-lj.si/javaodzacetka/>, je ta koda zapisana v datoteki MojPrvi.java znotraj imenika uvod.

```
public class MojPrvi {
    public static void main(String[] args) {
        System.out.println("Pozdravljen, svet!");
    }
}
```

Bodimo pozorni na velikost črk in vrste oklepajev! Barve niso sestavni del programa (ta je lahko samo čista besedilna datoteka); uporabljamo jih zgolj zavoljo boljše preglednosti. Na primer, z modro so označene besede, ki imajo v javi poseben pomen in jih ne moremo uporabljati kot imena spremenljivk, metod ali razredov. Takim besedam pravimo *rezervirane besede*.

- Odtipkano kodo shranimo kot datoteko z imenom `MojPrvi.java`. Tudi tokrat bodimo pozorni na velikost črk! Svetujemo vam, da javanske programe shranjujete v poseben imenik. (Ko bodo programi sestavljeni iz več datotek, boste še raje vsak program postavili v svoj lasten imenik.) Recimo, da program shranimo v imenik z absolutno potjo `~/programi/java` (Linux) oziroma `C:\Users\uporabnik\programi\java` (Windows).
- Odpremo terminal operacijskega sistema in se s pomočjo ukaza `cd` prestavimo v imenik, v katerega smo shranili program:

```
terminal> cd programi
terminal> cd java
```

Seveda se lahko v imenik prestavimo tudi v enem zamahu: v sistemu Linux odtipkamo `cd programi/java`, v sistemu Windows pa `cd programi\java`.

- Z ukazom `ls` (Linux) oziroma `dir` (Windows) se prepričamo, da imenik vsebuje datoteko `MojPrvi.java`.
- Program prevedemo:

```
terminal> javac MojPrvi.java
```

Če smo program pravilno napisali, potem prevajalnik (`javac`) ne izpiše ničesar in tvori datoteko `MojPrvi.class`, ki vsebuje prevedeno kodo programa (torej kodo v jeziku javanskega navideznega stroja). V nasprotnem primeru pa prevajalnik zgolj izpiše seznam napak. Smo pravilno upoštevali velikost črk? Smo uporabili pravilne vrste oklepajev?

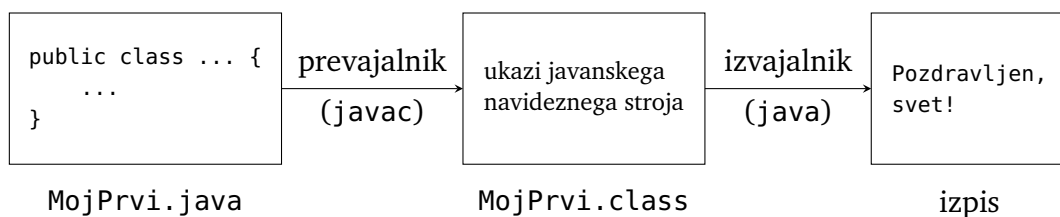
- Prevedeni program poženemo:

```
terminal> java MojPrvi
```

Argument javanskega izvajalnika (programa java) je ime datoteke *brez* končnice `.class`. Izvajalnik sedaj požene program in izpiše

```
Pozdravljen, svet!
```

Kot vidimo, datoteko `MojPrvi.java`, ki vsebuje javanski program, najprej s pomočjo prevajalnika pretvorimo v datoteko `MojPrvi.class`, ki vsebuje ukaze javanskega navideznega stroja, to pa lahko potem poženemo. Opisani dvostopenjski postopek je prikazan na sliki 1.1.



Slika 1.1 Prevajanje in poganjanje.

Oglejmo si naš prvi program nekoliko pobliže. Besedama `public class` v prvi vrstici sledi ime programa oziroma *razreda*, če smo natančni. Program je lahko sestavljen iz poljubnega števila razredov, a do vključno poglavja 5 bomo pisali le programe z enim samim razredom.

Ime datoteke, ki vsebuje program oziroma razred, mora biti sestavljeno iz imena razreda in končnice `.java`. Datoteka z razredom po imenu `MojPrvi` se torej *mora* imenovati `MojPrvi.java`.

Druga vrstica (`public static void main(String[] args)`) definira *metodo* po imenu `main`. O metodah bomo govorili v poglavju 4. Zaenkrat povejmo le to, da gre za poimenovan kos kode in da mora vsak program vsebovati metodo `main`.<sup>8</sup>

Vidimo, da se obe začetni vrstici zaključita z zavitim predklepajem, v zadnjih dveh vrsticah pa se nahajata ujemajoča zavita zaklepaja. Zaviti oklepaji podajajo *bločno zgradbo* programa. Razredu `MojPrvi` pripada vse, kar se nahaja med predklepajem v prvi in zaklepajem v zadnji vrstici, pod metodo `main` pa sodi vsa koda med predklepajem v drugi in zaklepajem v predzadnji vrstici. Za vsak predklepaj mora obstajati ujemajoči zaklepaj, sicer prevajalnik javi napako. Da se izognemo morebitnim pozabljenim zaklepajem, je smiselno najprej napisati oba oklepaja in šele potem tisto, kar gre vmes.

Ostane nam le še vrstica

```
System.out.println("Pozdravljen, svet!");
```

<sup>8</sup>Pojem metode je soroden pojmu *funkcije*, ki obstaja v številnih drugih jezikih.

Ta vrstica je javanski *stavek*, ki izpiše besedilo `Pozdravljen, svet!`. Kasneje bomo spoznali, da je `System.out.println` *klic metode*, "Pozdravljen, svet!" pa *argument* klica. Argumente navedemo znotraj okroglih oklepajev.<sup>9</sup> Dvojna narekovaja zamejujeta *niz*, kakor se v programerskem žargonu reče kosu besedila. Stavek zaključimo s podpičjem.

## 1.8 Lepopisna pravila

**Lepopisna pravila** ne vplivajo na prevajanje in izvajanje (tudi »grd« program se lahko uspešno prevede in izvede), močno pa vplivajo na berljivost programa. V javi ima vsak programerski element svoja pravila poimenovanja: `RazredePišemoTakole`, `metodePišemoTakole`, `KONSTANTE_PISAMO_TAKOLE` itd. Ta pravila bomo podajali, ko bomo spoznavali posamezne elemente programa.

Da je bločna zgradba programa bolj poudarjena, se zavitim oklepajem vedno pridružijo še *zamiki* — tabulatorski premiki oziroma skupki (tipično) po štirih presledkov na začetku posameznih vrstic. Kodo zamikamo v skladu s sledečimi pravili:

- Prva vrstica programa ni zamaknjena. Njen zamik je enak 0.
- Če se prejšnja vrstica zaključi z zavitim predklepajem, potem je zamik trenutne vrstice za eno enoto (en tabulatorski premik oziroma štiri presledke) večji od zamika prejšnje vrstice.
- Če se trenutna vrstica prične z zavitim zaklepajem, potem je njen zamik za eno enoto manjši od zamika prejšnje vrstice.
- Če ne velja nič od tega, je zamik trenutne vrstice enak zamiku prejšnje vrstice.

Navedena pravila smo upoštevali tudi pri programu `MojPrvi`. Prva in druga vrstica se končata z zavitim zaklepajem, zato sta druga in tretja vrstica glede na svoji predhodnici zamaknjeni za eno enoto v desno, predzadnja in zadnja vrstica pa se začneta z zavitim predklepajem, zato sta glede na svoji predhodnici zamaknjeni za eno enoto v levo.

Berljivost programa lahko povečamo tudi z vstavljanjem presledkov in praznih vrstic na primerna mesta. Na primer, vsak dvojiški operator z obeh strani obdamo s presledki. Namesto

```
int a=27*b-(int)(10*Math.sin(Math.PI/4));
```

torej raje pišemo

```
int a = 27 * b - (int) (10 * Math.sin(Math.PI / 4));
```

<sup>9</sup>V javi uporabljamo kar štiri vrste oklepajev: (okrogle), [oglate], {zavite} in <lomljene>. Vsaka vrsta ima svoj natanko določen namen uporabe.



Po tradicionalnih lepopisnih pravilih naj vrstica ne bi bila daljša od 80 znakov. Daljše vrstice zato smiselno delimo. Tega pravila se morda ni treba brezpogojno držati, kljub temu pa lahko služi kot dobra smernica.

## 1.9 Komentarji

Komentar je opomba h kodi, ki služi zgolj lažjemu razumevanju programa; prevajalnik ga ignorira. V javi lahko komentarje pišemo na dva načina:

- *Bločni komentarji* se pričnejo z zaporedjem `/*` in končajo z zaporedjem `*/`.
- *Vrstični komentarji* se pričnejo z zaporedjem `//` in segajo do konca vrstice.

V sledečem programu, ki prebere polmer in višino valja, izpiše pa njegovo prostornino, uporabljamo tako bločne kot vrstične komentarje:

```
// koda 1.1 (uvod/ProstorninaValja.java)
import java.util.Scanner;

public class ProstorninaValja {
    public static void main(String[] args) {
        // preberi podatke o valju
        Scanner sc = new Scanner(System.in);
        double r = sc.nextDouble(); // polmer
        double h = sc.nextDouble(); // višina

        // izpiši prostornino valja
        System.out.println(prostornina(r, h));
    }

    /*
    Vrne prostornino valja s podanim polmerom osnovne ploskve in
    višino.
    polmer -- polmer osnovne ploskve valja
    visina -- višina valja
    */
    public static double prostornina(double polmer, double visina) {
        return Math.PI * polmer * polmer * visina;
    }
}
```

S komentarji nikar ne varčujmo! Pišimo jih povsod, kjer delovanje kode ni takoj očitno. Tudi če nam je v trenutku, ko kodo pišemo, povsem jasno, kakšen je njen

pomen, ga bomo morda že v nekaj dneh pozabili in bomo potrebovali nesorazmerno veliko časa, da ga bomo spet izluščili. V tej knjigi bo komentarjev razmeroma malo, a zgolj zato, ker bomo programe podrobno pojasnjevali v spremnem besedilu. V praksi pa takšno spremno besedilo tvorijo komentarji.

Ker prevajalniku za komentarje ni mar, jih lahko uporabimo tudi za to, da določen del kode (začasno) onemogočimo. To nam pride prav še zlasti pri odpravljanju napak.

Omenimo še posebno vrsto bločnih komentarjev, imenovano *komentarji za javadoc*. Tovrstne komentarje pričnemo z zaporedjem `/**` in zaključimo z zaporedjem `*/`, pri njihovem pisanju pa moramo upoštevati še nekaj dodatnih pravil. Uporabljamo jih za dokumentiranje razredov in metod. Komentarje lahko s pomočjo programa `javadoc` pretvorimo v datoteko v formatu HTML in si jih v pregledni in strukturirani obliki ogledamo v brskalniku. Tovrstnih komentarjev v tej knjigi ne bomo uporabljali, kljub temu pa si zaslužijo vsaj droben primer:

```
// koda 1.2 (uvod/ProstorninaValja2.java)
import java.util.Scanner;

public class ProstorninaValja2 {
    ...
    /**
     Vrne prostornino valja s podanim polmerom osnovne ploskve in
     višino.
     @param polmer  polmer osnovne ploskve valja
     @param visina  višina valja
     @return prostornina valja
     */
    public static double prostornina(double polmer, double visina) {
        return Math.PI * polmer * polmer * visina;
    }
}
```

## 1.10 Odpravljanje napak

Programiranje je intelektualno zahtevna dejavnost, zato so napake njegova neobhodna sestavina. Nekatere se pokažejo že v fazi prevajanja. Lahko se nam, denimo, zgodi, da stavek pozabimo zaključiti s podpičjem. Lahko, recimo, pokličemo neobstoječo metodo ali pa spremenljivki priredimo vrednost neustreznega tipa. Tovrstne napake zazna že prevajalnik.

Nekatere napake pa se pokažejo šele v fazi izvajanja. Napakam, ki jih zazna izvajalnik, pravimo *izjeme*. Preprost primer izjeme je celoštevilsko deljenje z ničlo.

Prevajalnik izjem ne more zaznati, saj je njihov nastop praviloma odvisen od podatkov, ki v času prevajanja še niso znani. Na primer, nastop deljenja z ničlo je lahko pogojen z vrednostmi več različnih spremenljivk ali pa celo s podatki, ki jih program pridobi iz zunanjega sveta (npr. z uporabnikovim vnosom).

Če nas doletijo napake pri prevajanju ali izjeme, jih hočeš nočeš moramo odpraviti. Odpravljanje napak pri prevajanju je v splošnem lažje, čeprav tudi ni vedno trivialno. Odpravljanje izjem (in popravljanje napačnega delovanja, ki ga seveda ne zazna niti izvajalnik) pa je svojevrstna umetnost, ki ji pravimo *razhroščevanje*. *Hrošč* je namreč uveljavljen izraz za programerske napake, izvira pa iz časov, ko so bili računalniki veliki kot omare. Nekoč je v enega od njih zašla žuželka. Povzročila je pregrevanje vezja in s tem napačno delovanje programa, ki ga je omenjeni računalnik takrat izvajal.

Napak pri prevajanju se lotimo tako, da se osredotočimo na *prvo* napako, ki jo izpiše prevajalnik, jo odpravimo in program ponovno prevedemo. Nadaljnje napake so namreč pogosto neposredna posledica prve; ko prvo odpravimo, jih lahko kar precej »čudežno«<sup>1</sup> izgine.

Oglejmo si primer napake pri prevajanju. Če v programu `MojPrvi` pozabimo podpičje na koncu tretje vrstice, nam prevajalnik sporoči tole:

```
MojPrvi.java:3: error: ';' expected
    System.out.println("Pozdravljen, svet!")
                                ^
1 error
```

Najpomembnejši del obvestila o napaki najdemo takoj na začetku: zapis `MojPrvi:3` pove, da se napaka nahaja v datoteki `MojPrvi.java`, in sicer v vrstici 3. Brez tega podatka bi bilo iskanje napake bistveno težje. V nadaljevanju obvestila prevajalnik jasno izpiše, da pričakuje podpičje, poleg tega pa natančno označi mesto, kjer bi moralo biti.

Prevajalnikova obvestila niso vedno tako nedvoumna. Na primer, če izbrišemo zadnjo vrstico, dobimo obvestilo

```
MojPrvi.java:4: error: reached end of file while parsing
    }
    ^
1 error
```

Če kodo dosledno zamikamo, se takim napakam praviloma izognemo.

Včasih lahko ena sama napaka prevajalnik tako »zmede«, da izpiše še množico drugih (neobstoječih) napak. Na primer, če besedo `class` spremenimo v `clas`, dobimo pravo poplavo:

```
MojPrvi.java:1: error: class, interface, or enum expected
```

```

public clas MojPrvi {
    ^
MojPrvi.java:2: error: class, interface, or enum expected
    public static void main(String[] args) {
        ^
MojPrvi.java:4: error: class, interface, or enum expected
    }
    ^
3 errors

```

Kot smo že povedali, se osredotočimo samo na *prvo* napako. Ko jo odpravimo (ta je povsem jasno označena!), izginejo tudi druge.

Pri razhroščevanju programov si lahko pomagamo z orodji, ki jim pravimo *razhroščevalniki*, veliko napak pri izvajanju programov pa je mogoče odpraviti z dodajanjem preprostih izpisov. V prihodnjih poglavjih bomo na to temo še kakšno rekli.

**Naloga 1.1** Program *MojPrvi* »kvarite« na različne načine (npr. odstranite narekovaje v tretji vrstici, spremenite začetnico katere od besed ...) in opazujte prevajalnikove izpise. Ne pozabite, da morate program po vsaki spremembi shraniti in ponovno prevesti.

## 1.11 Povzetek

- Javanski program je sestavljen iz ene ali več čistih besedilnih datotek s končnico `.java`. V vsaki taki datoteki definiramo po en razred.
- Program s pomočjo prevajalnika (`javac`) najprej prevedemo v jezik javinega navideznega stroja, nato pa dobljeni prevod izvršimo s pomočjo izvajalnika (`java`).
- Pri pisanju programu moramo upoštevati sintaktična pravila, sicer se program ne bo prevedel. Zelo priporočljivo je, da se držimo tudi lepopisnih pravil.
- Napake so neobhodni del programiranja. Nekatere napake se pokažejo v času prevajanja, nekatere pa šele v času izvajanja.

## 2 Osnovni pojmi

V tem poglavju se bomo srečali z najosnovnejšimi programerskimi pojmi, kot so spremenljivka, izraz, tip itd. Na začetku se bomo »igrali« z javino konzolo, nato pa se bomo vrnili k samostojnim programom in spoznali, da ti postanejo precej zanimivejši, če vanje vpletemo še uporabnika.

### 2.1 Javanska konzola

Java od različice 9 naprej ponuja *konzolo*, ki omogoča podoben način dela kot terminal operacijskega sistema: vnesemo javanski izraz ali stavek, konzola pa ga izvrši in izpiše rezultat izraza oziroma morebitne podatke o učinku stavka. Konzolo požemo tako, da v terminalu operacijskega sistema izvršimo ukaz

```
terminal> jshell
```

Prikaže se pozivnik (da bo jasno, da smo v konzoli, ga bomo dosledno pisali):

```
jshell>
```

Sedaj lahko po mili volji vnašamo javanske izraze in stavke, konzola pa nam bo sproti »odgovarjala«. Ko se je naveličamo, jo zapustimo z ukazom `/exit`:

```
jshell> /exit
```

V javanski konzoli veljajo nekoliko drugačna pravila kot pri samostojnih programih. Podpičja na koncu stavkov niso obvezna, vrednost spremenljivke lahko izpišemo kar tako, da navedemo njeno ime (v samostojnih programih moramo uporabiti stavek `System.out.println(...)` ali kaj podobnega), isto spremenljivko lahko večkrat deklariramo (npr. zaporedje stavkov `int a = 3` in `int a = 4` je v konzoli povsem možno), pa še kaj bi se našlo.

### 2.2 Samostojen program

V poglavju 1 smo spoznali, kako napišemo, prevedemo in poženemo javanski program. Do vključno poglavja 4 bodo vsi naši programi sestavljeni iz enega samega

razreda. Do vključno poglavja 3 bo ta razred vseboval zgolj metodo `main`. To pomeni, da bodo vsi programi imeli takšno zgradbo:

```
public class ImeRazreda {
    public static void main(String[] args) {
        stavek1
        stavek2
        ...
    }
}
```

Ime razreda (*ImeRazreda*) je lahko sestavljeno iz črk, števk in podčrtajev. Po pravilih lepega vedenja se prične z veliko črko. Ime razreda naj bi odražalo cilj ali delovanje programa. Na primer, razred za program, ki prebere stranici pravokotnika in izpiše njegovo ploščino, je smiselno poimenovati `PloščinaPravokotnika`.

Datoteka, v katero zapišemo naš program, mora imeti enako ime kot razred (in končnico `.java`), prevedemo in poženemo pa jo z ukazoma `javac` in `java`:

```
terminal> javac ImeRazreda.java
terminal> java ImeRazreda
```

Ker bo ta postopek do nadaljnjega vedno enak (spreminjalo se bo le ime razreda), bomo pogosto pisali le kodo znotraj metode `main`:

```
stavek1
stavek2
...
```

## 2.3 Števila in aritmetične operacije

Ker je glavna naloga računalnikov računanje, lahko v vsakem spodobnem programskem jeziku zapisujemo števila in nad njimi izvajamo osnovne računske operacije. Java, kot bomo spoznali nekoliko kasneje, pozna več *tipov* števil, v osnovi pa jih lahko delimo na *celoštevilske* in *realnoštevilske*. Cela števila zapisujemo tako, kot smo navajeni, denimo 42, 0, -999, 1234567890 itd. Pri zapisu realnih števil uporabljamo decimalno piko (npr. 3.14, -23.0, 0.0025 ...), po želji pa tudi znak E ali e, ki predstavlja potenco števila 10. Na primer, zapis 5E+3 predstavlja število 5000.0 ( $= 5 \cdot 10^3$ ), zapis 7.91e-6 pa število 0.00000791 ( $= 7,91 \cdot 10^{-6}$ ). Število, ki vsebuje decimalno piko ali znak E oz. e, se obravnava kot realno, tudi če je matematično enako celemu številu. Na primer, število 42 je v javi celo, število 42.0 pa realno, čeprav se matematično ne razlikujeta.

Java ponuja operatorje za pet aritmetičnih operacij: seštevanje (+), odštevanje (-), množenje (\*), deljenje (/) in ostanek pri deljenju (%). Rezultat operacije je

odvisen od tipov operandov. Če sta oba operanda celoštevilska, je tak tudi rezultat, če je vsaj en operand realno število, pa je rezultat operacije realno število.

Zaženimo konzolo in računajmo:

```
jshell> 17 + 3
20
jshell> 17.0 - 3
14.0
jshell> 17 * 3.0
51.0
jshell> 17 / 3
5
jshell> 17.0 / 3
5.666666666666667
jshell> 17 % 3
2
```

Vidimo, da operator `/` pri celoštevilskih operandih decimalke preprosto odreže. Rezultat izraza  $a / b$  je torej odgovor na vprašanje, kolikokrat »gre«  $b$  v  $a$ , rezultat izraza  $a \% b$  pa nam pove, koliko pri tem ostane. Ostanek pri deljenju lahko uporabljamo tudi z realnoštevilskimi operandi, a to le redko pride prav. Tudi nasploh bomo cela števila uporabljali bistveno pogostejše kot realna. Še več: dogovorimo se, da pojem *število* brez dodanega pridevnika označuje celo število.

Potenciranje v javi ni osnovna operacija, obstaja pa metoda `Math.pow`, ki sprejme realni števili  $a$  in  $b$  in vrne rezultat izraza  $a^b$ , ki je seveda prav tako realno število. Metodi lahko podamo tudi celi števili, vendar pa se bosta samodejno pretvorili v realni. Na primer, klic `Math.pow(3, 4)` vrne vrednost `81.0`.

Operatorji `*`, `/` in `%` imajo prednost pred operatorjema `+` in `-`, zaporedni operatorji iz iste prednostne skupine pa se obravnavajo od leve proti desni. Na primer, izraz  $3 + 4 * 5$  se izračuna kot  $3 + (4 * 5)$ , izraz  $10 / 2 * 3$  pa kot  $(10 / 2) * 3$ , saj operatorji `*`, `/` in `%` pripadajo isti prednostni skupini. Enako kot v matematiki lahko vrstni red izvajanja operacij spremenimo z oklepaji (a le z okroglimi). Na primer, vrednost izraza  $(3 + 4) * (2 - 5)$  znaša `-21`.

## 2.4 Izraz in vrednost

*Izraz* je pravilno strukturirano zaporedje operatorjev in operandov. Primeri izrazov so `5`, `-2.1 * (14.3 - 7.6)` in `8 - a + b`. Vsi trije so *aritmetični izrazi*, kasneje pa bomo spoznali še nekatere druge vrste. Vsak izraz ima svojo *vrednost* (rezultat izračuna). Prvi navedeni izraz ima vrednost `5`, drugi `-14.07`, vrednost tretjega izraza pa je odvisna od vrednosti spremenljivk  $a$  in  $b$ . S spremenljivkami se bomo seznanili v naslednjem razdelku.

**Naloga 2.1** Kakšne so vrednosti sledečih izrazov?

- (a)  $7 / 4 * 4$
- (b)  $7 / 4 * 4.0$
- (c)  $(7 / 4) * 4.0$
- (d)  $7.0 / 4 * 4$
- (e)  $7 / (4 * 4)$
- (f)  $7 / (4 * 4.0)$
- (g)  $7 \% 4 * 4$
- (h)  $7 \% (4 * 4)$

**2.5 Spremenljivka in prireditve**

Računalnik nam ne omogoča le računanja, ampak tudi shranjevanje že izračunanih vrednosti in njihov ponovni priklic. To možnost nam ponuja tudi večina kalkulatorjev, a v računalniku lahko shranimo bistveno več števil. V letu 2023 ima povprečni računalnik 16 gigabajtov pomnilnika, kar zadošča za reci in piši štiri milijarde celih števil ali dve milijardi realnih! V praksi si sicer precejšen del pomnilnika prisvoji operacijski sistem, a še vedno ga je ogromno na voljo.

V javi lahko do te velikanske shrambe dostopamo prek *spremenljivk*. Spremenljivka je poimenovan delček pomnilnika, ki lahko hrani neko vrednost — celo število, realno število ali kaj tretjega. Vsaka spremenljivka ima svoje *ime* in *tip*. Tip nam pove, kakšne vrednosti lahko hrani spremenljivka.

S sledečim stavkom spremenljivko *deklariramo*:

```
tip ime;
```

Na primer, stavek

```
int ocena;
```

deklarira spremenljivko tipa `int` z imenom `ocena`, stavek

```
double ploscina;
```

pa deklarira spremenljivko tipa `double` z imenom `ploscina`. Mimogrede smo spoznali dva podatkovna tipa: tip `int` je eden od štirih celoštevilskih, tip `double` pa eden od dveh realnoštevilskih tipov. Kot bomo videli kasneje, lahko spremenljivka tipa `int` hrani poljubno celo število z intervala  $[-2^{31}, 2^{31} - 1]$ , spremenljivka tipa `double` pa realno število s (približnega) intervala  $[-10^{308}, 10^{308}]$ , pri čemer je natančnost omejena na približno 16 desetiških mest.<sup>1</sup>

<sup>1</sup>Število  $n$  pripada intervalu  $[a, b]$ , kadar je  $a \leq n \leq b$ .



Ime spremenljivke je lahko poljubno zaporedje velikih in malih črk, števk in podčrtajev (znakov `_`), vendar pa se ne sme pričeti s števko. Kadar je le mogoče, uporabimo ime, ki odraža pomen vrednosti, ki jo bo spremenljivka hranila. Na primer, če bomo v spremenljivko zapisali ploščino, je najboljša ime kar `ploscina`, slabše (a morda dopustno v kratkih ali enostavnih programih ali pa takrat, ko je pomen očiten iz konteksta) je `p`, ime `abc123` pa deluje kot slaba šala.<sup>2</sup> Javanski bonton zahteva, da se ime spremenljivke prične z malo črko, če pa je sestavljeno iz več besed, ga sestavimo takole, ne pa morda takole.

Ko spremenljivko deklariramo, izvajalnik zanjo rezervira primerno velik kos pomnilnika, vrednosti pa ji še ne dodeli. Spremenljivki lahko vrednost *priredimo* (tj. nastavimo) s *prireditvenim stavkom*:

```
spremenljivka = izraz;
```

Prireditveni stavek se izvrši v dveh korakih:

- Najprej se izračuna izraz na desni strani.
- Dobljena vrednost izraza se nato shrani v spremenljivko na levi strani.

Isti spremenljivki lahko vrednost večkrat priredimo. V sledečem primeru bo spremenljivka `ocena` najprej imela vrednost 7, nato pa 10:

```
int ocena;    // spremenljivka ocena še nima vrednosti
ocena = 7;    // v spremenljivki ocena je sedaj vpisano število 7
ocena = 10;   // spremenljivka ocena sedaj hrani število 10
```

Tip spremenljivke podamo samo ob deklaraciji, kasneje pa navajamo le še njeno ime, saj je tip že znan (in ga ne moremo spremeniti). Spremenljivka, ki ima vrednost, je *definirana*, če nima vrednosti, pa je *nedefinirana*.

Začetni določitvi vrednosti spremenljivke pravimo tudi *inicializacija*. Deklaracijo in inicializacijo ponavadi združimo v en sam stavek:

```
tip ime = izraz;
```

V sledečem primeru deklariramo in obenem inicializiramo pet spremenljivk:

```
int ocena = 10;
double a = 3.6;
double b = 5.0;
double ploscina = a * b;
double dvakratPloscina = 2 * ploscina;
```

<sup>2</sup>Namesto `ploscina` lahko pišemo tudi `ploščina`, žal pa (še vedno) tvegamo, da bo naš program manj prenosljiv ali pa da se bomo morali ukvarjati z rečmi, ki s programiranjem nimajo nikakršne povezave, denimo s kodiranjem znakov.

Kot vidimo, lahko v izrazih na desni strani nastopajo tudi že definirane spremenljivke. V četrti vrstici se spremenljivka `a` nadomesti z vrednostjo `3.6`, spremenljivka `b` pa z vrednostjo `5.0`. V peti vrstici se spremenljivka `ploscina` nadomesti z vrednostjo `18.0`; vrednost spremenljivke dvakrat `Ploscina` torej postane `36.0`.

Ker se prireditveni stavek izvede strogo od desne proti levi, lahko v javi pišemo stavke, ki nimajo nobenega smisla, če nanje gledamo skozi matematična očala:

```
int a = 10;
a = a + 1;
```

Kaj se zgodi v drugem stavku? Izračuna se vrednost izraza `a + 1` (tj. `11`), ta pa se nato shrani v spremenljivko `a`. Spremenljivka `a` se je tako povečala za `1`. Kot vidimo, operator `=` predstavlja določitev vrednosti spremenljivke, ne pa enakost v matematičnem smislu. Zato ga ne preberemo kot »je enako kot«, ampak kot »dobi vrednost« ali »postane«. Spremenljivka `a` je torej *dobila vrednost* izraza `a + 1`.

Spremenljivka ohrani svojo vrednost, dokler je aktivno ne spremenimo. Na primer:

```
jshell> int a = 5;
jshell> int b = a + 1;
jshell> a = a + 4;
jshell> a
9
jshell> b
6
jshell> b = a + 1;
jshell> b
10
```

Vidimo, da sprememba vrednosti spremenljivke `a` ne vpliva na spremenljivko `b`. Spremenljivka `b` se spremeni šele, ko njeno vrednost ponovno nastavimo na vrednost izraza `a + 1`.

Omenimo še, da lahko prireditveni stavek nastopa tudi kot izraz, saj ima namreč poleg svojega učinka (spremenljivka na levi strani dobi novo vrednost) tudi svojo lastno vrednost: to je nova vrednost spremenljivke na levi strani. Zato lahko v javi pišemo take reči:

```
jshell> int a = 3;
jshell> int b = 4;
jshell> int c = (a = b);
jshell> a
4
jshell> b
```

```
4
jshell> c
4
```

Koda `a = b` povzroči, da spremenljivka `a` dobi vrednost 4, hkrati pa ima tudi sama svojo vrednost — to je nova vrednost spremenljivke `a`, torej 4. Zato se vrednost 4 vpiše tudi v spremenljivko `c`. Mimogrede, v stavku `int c = (a = b)` lahko oklepaje izpustimo, saj je prireditveni operator, kot bomo videli v razdelku 3.13, desnoasociativen.

**Naloga 2.2** Kakšni sta vrednosti spremenljivk `u` in `v` po izvedbi sledečih stavkov?

```
int u = 10;
u = u * 2;
int v = u * 2;
u = u - v;
v = v + u;
u = u * v / (-(u - v))
```

**Naloga 2.3** Kakšni sta vrednosti spremenljivk `a` in `b` po izvedbi sledečih stavkov?

```
int a = 0;
int b = 1;
a = (a = b) + (a = a + b) + (b = a + b);
```

## 2.6 Tip

Vsaka spremenljivka v javi ima svoj tip. To velja tudi za vsako vrednost in vsak izraz. Na primer, tip vrednosti `-15` je `int`, vrednost `6.0` pa je tipa `double`. Tip izraza je enak tipu njegove vrednosti. Na primer, tip izraza `1 - 2.0 * 3` je `double`, saj je njegova vrednost enaka `-5.0`.

### 2.6.1 Številski tipi

Tipa `int` in `double` sta privzeta tipa za cela oziroma realna števila. Golo celo število bo vedno tipa `int`, golo realno število pa tipa `double`. Poleg njiju pa obstajajo še drugi številski tipi. Kot prikazujeta tabeli 2.1 in 2.2, se razlikujejo po razponu števil, ki jih lahko hranijo spremenljivke, pri realnoštevilskih tipih pa je pomembna tudi natančnost.

Najmanjšo oz. največjo vrednost, ki jo lahko hrani spremenljivka določenega celoštevilskega tipa, lahko pridobimo z izrazom `T.MIN_VALUE` oz. `T.MAX_VALUE`, kjer

**Tabela 2.1** Celoštevilski tipi v javi.

Tip	Najmanjša vrednost	Največja vrednost
byte	$-2^7 = -128$	$2^7 - 1 = 127$
short	$-2^{15} = -32\,768$	$2^{15} - 1 = 32\,767$
int	$-2^{31} \approx -2,1 \cdot 10^9$	$2^{31} - 1 \approx 2,1 \cdot 10^9$
long	$-2^{63} \approx -9,2 \cdot 10^{18}$	$2^{63} - 1 \approx 9,2 \cdot 10^{18}$

**Tabela 2.2** Realnoštevilski tipa v javi.

Tip	Razpon	Natančnost
float	$\pm 3,4 \cdot 10^{38}$	7 mest
double	$\pm 1,7 \cdot 10^{308}$	15–16 mest

je *T* enak Byte (za tip byte), Short (za tip short), Integer (za tip int) oziroma Long (za tip long). Na primer:

```
jshell> Integer.MAX_VALUE
2147483647
jshell> Integer.MIN_VALUE
-2147483648
```

Kaj se zgodi, če največjemu možnemu številu tipa int prištejemo 1?

```
jshell> Integer.MAX_VALUE + 1
-2147483648
```

Dobimo Integer.MIN\_VALUE! Če presežemo zgornjo mejo številskega tipa, se torej preselimo na drugo stran številske osi. Podobno se zgodi, če od vrednosti Integer.MIN\_VALUE odštejemo 1. To obnašanje — ki je v popolnem nasprotju z matematično intuicijo! — si lažje predstavljamo, če si namesto številske osi zamislimo krog, v katerem sta točki Integer.MAX\_VALUE in Integer.MIN\_VALUE soseda, ravno tako kot sta soseda točki 41 in 42 ali pa −13 in −12. Na enak način se obnašajo tudi drugi celoštevilski tipi.

Tipa byte in short se razmeroma redko uporabljata, tip long pa nam pride prav, ko imamo opravka z velikimi števili (a ne večjimi od približno  $9 \cdot 10^{18}$ ). Število tipa long definiramo tako, da dodamo pripono L. Na primer, število 100 je tipa int, število 100L pa tipa long. Če je vsaj en operand v izrazu tipa long, se izračun izvede v razponu tipa long, sicer pa se izraz izračuna v razponu tipa int:

```
jshell> 1000000 * 1000000    // to se izračuna v obsegu tipa int
-727379968
jshell> 1000000L * 1000000    // to se izračuna v obsegu tipa long
1000000000000
```

Pri pisanju števil lahko med številke vstavimo poljubno mnogo podčrtajev. Zapis `1_000_000_000_000L` je bistveno preglednejši od `1000000000000L`, pomeni pa isto.

**Naloga 2.4** Naj bosta `a` in `b` spremenljivki tipa `int` in naj velja  $a < b$ . V katerih primerih velja  $a + 1 < b + 1$ ? Kaj pa  $a / 2 < b / 2$ ?

**Naloga 2.5** Naj bosta `a` in `b` spremenljivki tipa `int`. Zapišite matematično formulo za izračun javanskega izraza  $a + b$ . (Namig: morda si boste morali pomagati z operatorjem mod, kakor v matematiki označujemo ostanek pri deljenju.)

### 2.6.2 Izrecna pretvorba tipa

Kaj izpiše sledeči izsek kode?

```
int a = 1;
int b = 1;
int c = 0;
int d = 1;
int n = 4;
System.out.println((a + b + c + d) / n);
```

Najbrž bi marsikdo pričakoval izpis `0.75` (povprečje števil 1, 1, 0 in 1), a ga bo javanski izvajalnik presenetil z ničlo. Seveda: števila `a`, `b`, `c` in `d` so cela, zato je njihova vsota prav tako cela, takšno pa je tudi število `n`. Operacija deljenja se zato izvede v domeni celih števil.

Če želimo deljenje celoštevilskih operandov izvesti v domeni realnih števil, moramo vsaj en operand pretvoriti v tip `double` ali `float`. To storimo z *izrecno pretvorbo tipa* (angl. *typecast*):

(tip) *izraz*

V gornjem primeru bi lahko torej rezultat `0.75` dobili takole ...

```
System.out.println( ((double) (a + b + c + d)) / n);
```

... ali takole ...

```
System.out.println( (a + b + c + d) / (double) n);
```

... seveda pa bi lahko v tip `double` pretvorili tudi oba operanda.

Če vrednost ožjega celoštevilskega tipa priredimo spremenljivki širšega celoštevilskega tipa ali pa če vrednost celoštevilskega tipa priredimo spremenljivki realnoštevilskega tipa, se ustrezna pretvorba tipa izvede samodejno. Na primer, namesto

```
int a = 3;
long b = (long) a;
double c = (double) b;
```

lahko pišemo kar

```
int a = 3;
long b = a;
double c = b;
```

Pri prirejanju vrednosti širšega celoštevilskega tipa spremenljivki ožjega celoštevilskega tipa in pri prirejanju vrednosti realnoštevilskega tipa spremenljivki celoštevilskega tipa pa moramo uporabiti izrecno pretvorbo tipa. V nasprotnem primeru prevajalnik sporoči napako, v kateri nas opozori na morebitno izgubo informacije. Na primer:

```
jshell> int a = 3L;
|   Error:
|   incompatible types: possible lossy conversion from long to int
|   int a = 3L;
|           ^^

jshell> int a = (int) 3L;
jshell> a
3
jshell> int b = (int) 10.7;
jshell> b
10
```

Vidimo, da pretvorba realnega števila v celo zgolj odreže decimalke. Če želimo, da se število zaokroži na najbližje celo število, si pomagamo z metodo `Math.round`:

```
jshell> int b = (int) Math.round(10.7);
jshell> b
11
```

Metoda `Math.round` vrne število tipa `long`, zato za pretvorbo v tip `int` še vedno potrebujemo operator `(int)`.

Ožji tipi se po potrebi samodejno pretvorijo v širše tudi pri aritmetičnih operacijah: če je en operand celoštevilskega, drugi pa realnoštevilskega tipa, se celoštevilski

operand samodejno pretvori v tip realnoštevilskega operanda. Operacija se izvede v domeni realnoštevilskega tipa, takšnega tipa pa je tudi rezultat. Podobno velja za tipa `int` in `long`: če v aritmetični operaciji nastopata oba, se operand tipa `int` pretvori v tip `long`. Operacija se izvede v domeni tipa `long`, tega tipa pa je tudi rezultat.

### Naloga 2.6 Kakšne so vrednosti sledečih izrazov?

- (a) `(int) 3.9 + 3.9`
- (b) `(int) 3.9 + (int) 3.9`
- (c) `(int) (3.9 + 3.9)`
- (d) `3.9 + (int) 3.9`
- (e) `(double) (3 / 4)`
- (f) `(double) 3 / 4`
- (g) `((double) 3) / 4`
- (h) `1000000 * 1000000 / 1000000`
- (i) `1000000 * 1000000 / 1000000L`
- (j) `1000000 * 1000000L / 1000000`
- (k) `1000000L * 1000000L / 1000000L`

### 2.6.3 Znaki

V javi se posamezni znaki obravnavajo preprosto kot števila. Vsak znak ima namreč svojo celoštevilsko *kodo*. Na primer, znak `'A'` (znake pišemo v enojnih narekovajih) ima kodo 65, zato je predstavljen kar s številom 65. Izraz `'A' + 1`, denimo, ima vrednost 66, to pa je hkrati koda znaka `'B'`.

Že leta 1963 so Američani definirali t. i. abecedo ASCII (izg. áski), ki določa kode za 128 znakov, med katerimi najdemo vse velike in male črke angleške abecede, števke, običajne matematične operatorje, oklepaje, ločila in še kaj. Danes, ko je računalništvo že dolgo časa mednarodna disciplina, je v veljavi abeceda Unicode, a ta je zgolj razširitev abecede ASCII (vsak znak iz abecede ASCII pripada tudi abecedi Unicode in ima v obeh abecedah isto kodo), zato se z njo ne bomo ukvarjali.

Znaki v javi pripadajo tipu `char`. Ta tip je dejansko celoštevilski tip z razponom vrednosti  $[0, 2^{16} - 1]$ , zato ga lahko brez težav pretvarjamo v tip `int`. Tip `char` se od ostalih celoštevilskih tipov razlikuje zgolj pri izpisu: če izpišemo vrednost tipa `char`, se izpiše znak, če to vrednost pretvorimo v kak drug celoštevilski tip, pa se izpiše koda znaka:

```
jshell> char c = 'A';
jshell> c
'A'
jshell> int d = c;
```

```
jshell> d
65
jshell> char e = (char) (d + 1)
jshell> e
'B'
jshell> e + 1
67
```

Vidimo, da se lahko znak samodejno pretvori v tip `int` (ali `long`), v obratni smeri pa potrebujemo operator `(char)`, saj je tip `char` ožji od tipov `int` in `long`.

Tako velike kot male črke angleške abecede imajo zaporedne kode: `'A' ↦ 65`, `'B' ↦ 66`, ..., `'Z' ↦ 90`, `'a' ↦ 97`, `'b' ↦ 98`, ..., `'z' ↦ 122`. Če vemo, da, denimo, spremenljivka `crka` hrani veliko (oz. malo) črko angleške abecede, dobimo zaporedno številko te črke z izrazom `crka - 'A' + 1` (oz. `crka - 'a' + 1`).

Zaporedne kode imajo tudi števke: `'0' ↦ 48`, `'1' ↦ 49`, ..., `'9' ↦ 57`. Števke tipa `char` torej niso istovetne s števki tipa `int`! Če želimo števko tipa `char` pretvoriti v pripadajočo števko tipa `int`, moramo od nje odšteti vrednost 48 (oziroma kar znak `'0'`):

```
jshell> char stevka = '3';
jshell> stevka
'3'
jshell> (int) stevka
51
jshell> stevka - '0'
3
```

Kode od 0 do 31 in koda 127 v abecedi ASCII pripadajo t. i. krmilnim znakom. Večina od njih ima danes le še zgodovinski pomen, nekateri pa se še vedno uporabljajo. Na primer, znak s kodo 10 (zapišemo ga kot `'\n'`) predstavlja v Linuxu in drugih unixovskih sistemih *prelom vrstice* (tj. skok v naslednjo vrstico); v sistemu Windows je prelom vrstice predstavljen z zaporedjem znakov `'\r'` (koda 13) in `'\n'`. Znak s kodo 9 (`'\t'`) je tabulatorski premik. Znak s kodo 7 (`'\7'`) v nekaterih terminalih odda zvočni signal.

**Naloga 2.7** Recimo, da spremenljivka `crka` hrani veliko črko angleške abecede. Napišite izraz, katerega vrednost je pripadajoča mala črka, pri čemer se delajte, da ne poznate kod velikih in malih črk (vse, kar veste, je, da so tako kode velikih kot kode malih črk zaporedne).



### 2.6.4 Nizi

Pri programiranju imamo ponavadi največ opravka s števili, kljub temu pa se pogosto ukvarjamo tudi z *nizi* (kosi besedila). V javi pišemo nize znotraj dvojnih narekovajev, "na primer takole". Nizi pripadajo tipu `String`:

```
String pozdrav = "Dober dan";
String ime = "Maja";
```

Nize lahko sestavljamo s pomočjo operatorja `+`:

```
String skupaj = pozdrav + ", " + ime + "!";
System.out.println(skupaj); // Dober dan, Maja!
```

Operator `+` ima torej dva pomena: če ga vstavimo med dve števili, ju sešteje, niza pa zlepi. Ali lahko z operatorjem `+` povežemo tudi niz in število (ali pa število in niz)? Da. V tem primeru se število pretvori v niz, nato pa se niza zlepi:

```
int a = 5;
int b = 6;
int c = a + b;
System.out.println("Vsota števil " + a + " in " + b + " znaša " + c);
```

Če niz vsebuje zgolj pravilen zapis števila (npr. "25" ali "-17.6"), ga lahko pretvorimo v pripadajoče celo oziroma realno število s pomočjo metode `Integer.parseInt` oziroma `Double.parseDouble`. Število pa v niz pretvorimo s pomočjo metode `Integer.toString` oziroma `Double.toString`:

```
String strPolmer = "10";
String strPi = "3.14159";
int polmer = Integer.parseInt(strPolmer); // 10
double pi = Double.parseDouble(strPi); // 3.141519
String strPloscina = Double.toString(pi * polmer * polmer);
// "314.159"
```

Zaenkrat bodi dovolj, se bomo pa k nizom vrnili v razdelku 6.9.

**Naloga 2.8** Kaj izpiše sledeči izsek kode in kako bi ga popravili, ne da bi števili sešteli v posebno spremenljivko?

```
int a = 5;
int b = 6;
System.out.println("Vsota števil " + a + " in " + b +
    " znaša " + a + b);
```

### 2.6.5 Priredljivost tipov

Če želimo spremenljivki prirediti vrednost, se morata tipa vrednosti in spremenljivke ujemati. Sledeči stavek, ki poskuša vrednost tipa `String` prirediti spremenljivki tipa `int`, se seveda ne bo prevedel:

```
int n = "35";
```

Kljub temu pa ni nujno, da sta tipa vrednosti in spremenljivke povsem enaka. Na primer, videli smo, da lahko vrednost tipa `int` priredimo spremenljivki tipa `long`:

```
int a = 42;
long b = a;
```

Obratno ni mogoče: če želimo vrednost tipa `long` prirediti spremenljivki tipa `int`, si moramo pomagati z izrecno pretvorbo tipa:

```
int c = (int) b;
```

Če lahko vrednost tipa  $T_1$  priredimo spremenljivki tipa  $T_2$ , bomo rekli, da je tip  $T_1$  *priredljiv* tipu  $T_2$ . Na primer, tip `int` je priredljiv tipu `long`, ne pa tudi obratno. Če sta tipa enaka, sta seveda vzajemno priredljiva, pri številiških tipih pa so poleg tega ožji tipi priredljivi širšim, celoštevilski pa realnoštevilskim.

## 2.7 Stavek

Stavek je osnovna enota programa, ki »nekaj naredi«. Pravimo, da ima vsak stavek svoj *učinek*.

Nekaj stavkov že poznamo. Recimo prireditveni stavek:

```
spremenljivka = izraz;
```

Kaj naredi prireditveni stavek? Izračuna vrednost izraza na desni strani in rezultat vpiše v spremenljivko na levi strani. Njegov učinek je torej ta, da spremenljivka na levi strani dobi novo vrednost.

Poznamo tudi stavek oblike

```
tip spremenljivka;
```

ki deklarira spremenljivko podanega tipa, in stavek oblike

```
tip spremenljivka = izraz;
```

ki spremenljivko inicializira z vrednostjo podanega izraza.

Ta stavek smo prav tako že večkrat srečali:

```
System.out.println(izraz);
```

Kakšen je njegov učinek? Izračuna izraz in izpiše njegovo vrednost. Kot bomo videli v poglavju 4, je gornji stavek primer *klica metode*.

## 2.8 Bločna zgradba programa in doseg spremenljivk

Zaporedje stavkov znotraj para zavitih oklepajev se imenuje *blok*. Pojem bloka je tesno povezan z *dosegom spremenljivk*. Spremenljivke, deklarirane znotraj bloka, namreč obstajajo le do konca tega bloka; ko se blok zaključi, izginejo.

V izseku kode na sliki 2.1 so v vsaki vrstici navedene spremenljivke, ki takrat obstajajo. Spremenljivki *b* in *d* obstajata od svoje deklaracije do konca bloka 1, spremenljivka *c* pa od svoje deklaracije do konca bloka 2. Če bi, denimo, do spremenljivke *c* poskušali dostopati po koncu bloka 2, bi prevajalnik javil napako.

```
int a = 1;
// a
{
    blok 1
    {
        // a
        int b = 2;
        // a, b
        {
            blok 2
            {
                // a, b
                int c = 3;
                // a, b, c
            }
        }
        // a, b
        int d = 4;
        // a, b, d
    }
}
// a
```

Slika 2.1 Koda z vgnezenima blokoma.

Vsaka spremenljivka je v okviru istega bloka lahko deklarirana le po enkrat. Na primer, sledeča koda se ne prevede, saj je spremenljivka *a* z vrednostjo 3 vidna tudi v notranjem bloku:

```
{
    int a = 3;
    {
        int a = 4;
    }
}
```

```
    }
}
```

Naslednja koda pa se prevede brez napak, saj spremenljivka *a* z vrednostjo 3 obstaja samo do konca prvega bloka. Spremenljivka *a* v drugem bloku je *povsem nova* spremenljivka in nima nič skupnega s prvo spremenljivko *a* (ki takrat sploh ne obstaja več). Še več: druga spremenljivka *a* bi lahko pripadala celo drugemu tipu.

```
{
    int a = 3;
}
{
    int a = 4;
}
```

Definicije metod se kljub zavitim oklepajem formalno ne obravnavajo kot bloki, glede dosega spremenljivk pa se obnašajo povsem enako. Kot bomo videli v poglavju 4, so spremenljivke, deklarirane znotraj metode, vidne samo v tisti metodi in v nobeni drugi.

## 2.9 Vhod in izhod

Skoraj vsak program tako ali drugače komunicira s svojim okoljem. Podatki, ki jih program od okolja pridobi, sestavljajo njegov *vhod*, podatki, ki jih posreduje okolju, pa tvorijo njegov *izhod*. Programi v tej knjigi bodo podatke brali s *standardnega vhoda*, izpisovali pa jih bodo na *standardni izhod*. Standardni vhod je privzeto tipkovnica, standardni izhod pa zaslon. Kot bomo videli, pa lahko tako standardni vhod kot standardni izhod preusmerimo (denimo na datoteko).

### 2.9.1 Standardni izhod

Podatke bomo izpisovali s pomočjo stavkov `System.out.print(...)` in `System.out.println(...)`, v razdelku 3.8 pa bomo spoznali še stavek `System.out.printf(...)`. Stavek `System.out.print(...)` preprosto izpiše svoj argument, stavek `System.out.println(...)` pa poleg tega izpiše še prelom vrstice — znak `'\n'` v Linuxu oziroma zaporedje znakov `'\r'` in `'\n'` v sistemu Windows, ki povzroči skok na začetek naslednje vrstice. To pomeni, da stavka

```
System.out.print("a");
System.out.println("b");
```

izpišeta

```
ab
```

stavka

```
System.out.println("a");
System.out.println("b");
```

pa

```
a
b
```

S stavkoma `System.out.print(...)` in `System.out.println(...)` lahko izpišemo podatke poljubnih tipov. Na primer, koda

```
System.out.print("Dober dan");
char klicaj = '!';
System.out.println(klicaj);
int a = 10;
double b = 2.4;
float c = (float) (a / b);
System.out.print(a);
System.out.print(" / ");
System.out.print(b);
System.out.print(" = ");
System.out.println(c);
```

izpiše

```
Dober dan!
10 / 2.4 = 4.1666665
```

### 2.9.2 Standardni vhod (in bežen pogled v pakete)

Programi postanejo zanimivejši, ko vanje vključimo uporabnika. Uporabnik lahko podatke posreduje na različne načine, v tej knjigi pa se bomo omejili na standardni vhod. To pomeni, da bo uporabnik podatke v osnovi vnašal prek tipkovnice, s preprostim trikom pa bomo lahko dosegli, da bo program svoje vhodne podatke namesto tega bral iz datoteke.

Javina standardna knjižnica ponuja več razredov, ki omogočajo branje podatkov, za naše potrebe pa bo najpripravnejši razred `Scanner`. Njegovo uporabo najlažje prikažemo na primeru. Sledeči program prosi uporabnika, naj vnese dve števili, nato pa izpiše njuno vsoto:<sup>3</sup>

<sup>3</sup>Odslej bomo kodo pogosto pričeli s komentarjem, ki vsebuje številko, s pomočjo katere se bomo lahko kasneje nanjo sklicevali, in pot do datoteke (v okviru arhiva ZIP na spletni strani knjige), ki kodo vsebuje.

```
// koda 2.1 (osnovniPojmi/Vsota1.java)
import java.util.Scanner; // (1)

public class Vsota1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in); // (2)

        System.out.print("Vnesite prvo število: ");
        int prvo = sc.nextInt();
        System.out.print("Vnesite drugo število: ");
        int drugo = sc.nextInt();

        int vsota = prvo + drugo;
        System.out.println("Vsota znaša " + vsota);
    }
}
```

Ker se razred `Scanner` nahaja v paketu `java.util` namesto v »privzetem« paketu `java.lang`, ga v vrstici (1)<sup>4</sup> uvozimo. Če tega ne storimo, prevajalnik ne bo vedel, kje naj ga najde. *Paket?* To ni nič drugega kot množica sorodnih razredov. Paket `java.lang`, denimo, vsebuje nekaj najnujnejših razredov, paket `java.util` med drugim ponuja razrede za delo z vsebovalniki (poglavje 10), v paketu `java.time` so zbrani razredi za delo z datumi in časi itd. Razrede iz paketa `java.lang` lahko uporabljamo brez stavka `import`, razrede iz ostalih paketov pa moramo bodisi uvoziti ali pa pred vsako pojavitev njihovih imen dodati ime paketa (npr. `java.util.Scanner` namesto `Scanner`). Običajno se odločimo za prvo možnost.

Namesto

```
import java.util.Scanner;
```

lahko pišemo tudi

```
import java.util.*;
```

in s tem uvozimo vse razrede iz paketa `java.util`. Ta možnost je priročna, vendar pa se nam lahko nehote zgodi, da uvozimo razred, ki se imenuje enako kot nek drug razred v našem programu. Java ta pojav sicer dopušča, vendar pa moramo potem vseskozi nedvoumno razlikovati med takima razredoma; imen razredov ne smemo več pisati samostojno, temveč le v obliki *paket.razred*. V tej knjigi bomo na začetku iz paketa `java.util` uvažali posamične razrede, od poglavja 9 naprej pa bomo paket uvažali v celoti.

---

<sup>4</sup>Tudi v nadaljevanju bomo izbrane vrstice v kodi označevali na ta način in se potem v besedilu nanje sklicevali.

Toliko o paketih; z njimi se ne bomo več ukvarjali. Vrnimo se na naš program. V vrstici (2) inicializiramo spremenljivko tipa `Scanner`; izraz `System.in` predstavlja standardni vhod. (Vrstici (1) in (2) lahko do nadaljnjega obravnavamo kot del neobhodne navlake.) Razreda `Scanner` ni težko uporabljati, saj moramo zgolj s klicem `sc.nextInt()` brati posamezna cela števila. Če bi želeli prebrati realno število, bi napisali `sc.nextDouble()`. Podobne metode obstajajo tudi za nekatere druge tipe. Spoznali jih bomo, ko jih bomo potrebovali.

Sledi primer izvedbe programa:

```
Vnesite prvo število: 5
Vnesite drugo število: -9
Vsota znaša -4
```

Uporabnikov vnos smo podčrtali. Tega dogovora se bomo držali tudi v prihodnje. Program izpiše poziv `Vnesite prvo število:` in prijazno počaka, da uporabnik vnese število in pritisne tipko `Enter`. Nato se zgodba ponovi še za drugo število. Na koncu izpiše vsoto vnesenih števil. Če bi uporabnik namesto celega števila vnesel nekaj drugega (npr. 3.6 ali abc), bi se sprožila izjema, saj klic `sc.nextInt()` prebranega vnosa ne bi mogel pretvoriti v celo število.

**Naloga 2.9** Napišite program, ki prebere stranici pravokotnika in izpiše njegovo ploščino in obseg.

**Naloga 2.10** Napišite program, ki prebere začetni in končni čas (oboje v urah in minutah) in izpiše razliko med časoma v urah in minutah. Lahko predpostavite, da je začetni čas pred končnim. Na primer:

```
Vnesite začetno uro: 15
Vnesite začetno minuto: 50
Vnesite končno uro: 18
Vnesite končno minuto: 30
Časovna razlika znaša 2 h 40 min.
```

## 2.10 Preverjanje programov

V prejšnjem razdelku smo napisali program, ki prebere dve števili in izpiše njuno vsoto. Če želimo program nekoliko bolj sistematično (in avtomatizirano) preveriti, je smiselno odstraniti vse pomožne izpise:

```
// koda 2.2 (osnovniPojmi/Vsota.java)
import java.util.Scanner;
```

```

public class Vsota {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int prvo = sc.nextInt();
        int drugo = sc.nextInt();
        int vsota = prvo + drugo;
        System.out.println(vsota);
    }
}

```

Program je sedaj bistveno manj prijazen. Ko ga poženemo, vidimo samo utripajoč kurzor in če programa ne bi poznali, ne bi niti vedeli, da moramo vnesti neko število. Ko odtipkamo obe števili, nam program izpiše rezultat, in to spet brez kakršnegakoli spremnega izpisa. Po drugi strani pa je program sedaj lažje preverjati, saj smo ves njegov izhod (tj. vse, kar med svojim izvajanjem izpiše) skrčili na tisto najbolj bistveno: izpis vsote prebranih števil. Take programe pogosto preverjamo tako, da si pripravimo nekaj datotek, ki vsebujejo primere veljavnih vhodov, in enako število datotek, ki vsebujejo pripadajoče pričakovane izhode, nato pa program poženemo na vsaki vhodni datoteki posebej in dobljene rezultate primerjamo s pripadajočimi izhodnimi datotekami. Kot bomo videli, lahko z nekaj truda postopek tudi avtomatiziramo.

### 2.10.1 Tipkovnica in zaslon

Najosnovnejši način preverjanja programov že poznamo: program prevedemo in poženemo, nato pa vnesemo vhodne podatke in si ogledamo dobljeni izpis. Na primer:

```

5
6
11

```

### 2.10.2 Ukaz echo

Program lahko namesto s preprostim `java Vsota` v terminalu poženemo tudi tako, da mu spotoma podamo še celoten vhod:

```

terminal> echo 5 6 | java Vsota
11

```

Ukaz `echo` poznata tako Linux kot Windows. Deluje enostavno tako, da svoje argumente (v našem primeru sta to števili 5 in 6) izpiše na standardni izhod. Operator `|` preusmeri izhod, ki ga proizvede program na njegovi levi (torej `echo`), na vhod programa na njegovi desni. To pomeni, da se števili 5 in 6 dejansko ne bosta izpisali



na zaslon, ampak se bosta našemu programu posredovali kot vhod. Programu je »vseeno«, ali svoj vhod pridobi prek tipkovnice ali s preusmeritvijo izhoda nekega drugega programa, saj za vse poskrbi lupina operacijskega sistema.

Prednost tega načina pred osnovnim se pokaže, če želimo program večkrat zaporedoma pognati z istim vhodom. To je kar pogost scenarij: program poženemo na izbranem vhodu, ugotovimo, da ne deluje, zato ga popravimo, prevedemo in ponovno poženemo z istim vhodom itd. Tipkanje vedno istega vhoda prej ali slej postane naporno, pri zagonu programa s pomočjo ukaza echo pa si lahko pomagamo s funkcijo ponovnega priklica že izvršenega ukaza, ki jo ponuja vsak spodoben terminal (po zgodovini vnesenih ukazov se ponavadi lahko sprehajamo s tipkama ↑ in ↓).

### 2.10.3 Preusmeritev vhoda

Pri obsežnejših vhidih tudi ukaz echo postane okoren. Kot smo že povedali, lahko tako standardni vhod kot standardni izhod preusmerimo na datoteko. Oglejmo si, kako dosežemo, da program svoj vhod namesto s tipkovnice prebere iz datoteke vhod.txt. Pripravimo si datoteko vhod.txt (shranimo jo v imenik, v katerem se nahaja program Vsota) in vanjo zapišimo želeni vhod. Na primer:

5 6

Ni pomembno, ali vsako število zapišemo v svojo vrstico ali pa ju zgolj ločimo s presledkom. Vsak klic `sc.nextInt()` preskakuje presledke, tabulatorje in prelome vrstice, dokler ne prispe do prvega znaka, ki ni nič od naštetega.

V terminalu lahko sedaj poženemo sledeči ukaz:

```
terminal> java Vsota < vhod.txt
11
```

Program se tudi tokrat ne zaveda, da bere iz datoteke namesto s tipkovnice. Vse se zgodi za kulisami.

### 2.10.4 Preusmeritev vhoda in izhoda

Preusmerimo lahko tudi izhod:

```
terminal> java Vsota < vhod.txt > rezultat.txt
```

Program je sedaj svoj izhod namesto na zaslon izpisal v datoteko rezultat.txt. Če jo odpremo v poljubnem besedilnem urejevalniku, vidimo število 11. Z ukazom `cat` (Linux) oziroma `type` (Windows) lahko njeno vsebino izpišemo v terminal:

```
terminal> cat rezultat.txt
11
```

Če si poleg vhodne datoteke (npr. `vhod.txt` z vsebino 5 6) vnaprej pripravimo tudi pripadajočo datoteko s pričakovanim izhodom (npr. `izhod.txt` z vsebino 11), lahko dejanski izhod programa s pomočjo ukaza `diff` (Linux) oziroma `fc` (Windows) primerjamo s pričakovanim:

```
terminal> diff rezultat.txt izhod.txt
```

Ukaz preprosto primerja podani datoteki. Če imata isto vsebino, ne izpiše ničesar, sicer pa izpiše razlike med njima.

### 2.10.5 Avtomatizacija preverjanja

Preusmeritev vhoda in izhoda ter ukaz `diff` oz. `fc` nam omogočata avtomatizirano preverjanje programov s pomočjo *testnih primerov* — primerov vhodov in pripadajočih izhodov. Pripravimo si, denimo, deset testnih primerov, torej deset parov vhodnih in izhodnih datotek. Poimenujmo jih `vhod01.txt`, `vhod02.txt`, ..., `vhod10.txt` in `izhod01.txt`, `izhod02.txt`, ..., `izhod10.txt`. Na primer, vsebina prvega para datotek je lahko 5 6 (`vhod01.txt`) in 11 (`izhod01.txt`), drugi par lahko tvorita vhod -10 -20 in izhod -30 itd. Naš program lahko sedaj preverimo z *lupinsko skripto*, programom, sestavljenim iz ukazov `lupine`, ki ga je mogoče pognati neposredno v terminalu.

Oglejmo si, kako se lupinske skripte za samodejno preverjanje programov lotimo v sistemu Linux. Odprimo programerski ali navadni besedilni urejevalnik in odtipkajmo sledeče vrstice:

```
#!/bin/bash
# koda 2.3 (osnovniPojmi/preveri.sh)

for i in {01..10}; do
    echo -n "$i: "
    java $1 < vhod$i.txt > rezultat$i.txt
    if diff rezultat$i.txt izhod$i.txt > /dev/null; then
        echo "pravilno"
    else
        echo "napačno"
    fi
done
```

Datoteki damo ime `preveri.sh` in jo shranimo v imenik, kjer se nahaja program, ki ga želimo preveriti. Da jo bomo lahko neposredno pognali, jo moramo označiti za izvršljivo:

```
terminal> chmod +x preveri.sh
```

Sedaj prevedemo naš javanski program, če ga še nismo, in poženemo skripto:

```
terminal> ./preveri.sh Program
```

Na primer:

```
terminal> ./preveri.sh Vsota
```

Ne bomo se spuščali v podrobnosti, saj je tema te knjige java, ne pa lupina operacijskega sistema. Povejmo samo to, da skripta požene podani program na vsaki vhodni datoteki in dobljeni izhod primerja s pričakovanim. Če se izhoda ujemata, izpiše besedilo pravilno, sicer pa napačno.

V sistemu Windows lahko enakovredno lupinsko skripto napišemo takole:

*REM koda 2.4 (osnovniPojmi/preveri.bat)*

```
@echo off
setlocal enabledelayedexpansion

for /l %%i in (1, 1, 10) do (
    set j=0%%i
    set k=!j:~-2!
    set vhod=vhod!k!.txt
    set izhod=izhod!k!.txt
    set rezultat=rezultat!k!.txt
    java %1 < !vhod! > !rezultat!
    fc !izhod! !rezultat! > NUL
    if !errorlevel! equ 1 (echo !k!: napačno) else (echo !k!: OK)
)
```

Skripto shranimo kot datoteko z imenom `preveri.bat` v imenik, kjer domuje program, ki ga želimo preveriti. V sistemu Windows se datoteke s končnico `.bat` samodejno obravnavajo kot izvršljive, zato jo lahko takoj poženemo:

```
terminal> preveri.bat Program
```

Na primer:

```
terminal> preveri.bat Vsota
```

**Naloga 2.11** Na različne načine poženite še programa iz nalog 2.9 in 2.10. Oba temeljito preverite z množico vhodnih in izhodnih datotek.

### 2.10.6 Preverjanje in dokazovanje

Recimo, da smo si pripravili veliko množico testnih primerov in program uspešno preverili na vseh. Ali lahko trdimo, da je program pravilen? Samo v primeru, če smo izčrpali vse *možne* vhode. Žal pa je to v praksi skoraj vedno nemogoče. Že pri programu, ki sešteje dve števili tipa `int`, imamo  $(2^{32})^2 = 2^{64}$  možnih vhodov! Če bi za vsak testni primer potrebovali milijardinko sekunde (v letu 2023 bi v resnici precej več), bi preverjanje trajalo kakšno milijardo let! Kako se lahko potem sploh kdaj prepričamo v pravilnost delovanja programa?

Edini način je ta, da pravilnost programa *matematično dokažemo*. Pri programu, ki sešteje dve celi števili, bi morali javansko vsoto zapisati z matematično formulo (naloge 2.5). Ko bi to storili, bi hitro ugotovili, da program proizvede matematično pravilen rezultat le v primeru, ko vsota števil ne presega meja tipa `int`. Če predpostavimo, da bosta vhodni števili vedno v intervalu (denimo)  $[-10^9, 10^9]$ , pa lahko z matematično gotovostjo trdimo, da program deluje pravilno.

Zanimivo je, da lahko v nekaterih primerih dokažemo le pogojno pravilnost: program deluje pravilno, *če se ustavi*. Obstajajo namreč programi, za katere *ne moremo* (ne zgolj, da *ne znamo*) dokazati ustavljenosti *pri vseh mogočih* vseh vhodih.

Matematično dokazovanje pravilnosti in ustavljenosti programov presega okvir te knjige. Tudi v praksi se večina programov le preverja. Moramo pa se zavedati, da to ponavadi ni dovolj. Saj tudi sami dobro vemo: vsak kolikor toliko obsežen program vsebuje vsaj kakšno napako.<sup>5</sup>

## 2.11 Povzetek

- Java nam omogoča prevajanje in poganjanje samostojnih programov ter izvajanje posamičnih stavkov v konzoli.
- Java ponuja operatorje za seštevanje, odštevanje, množenje, deljenje in ostanek pri deljenju.
- Izraz je pravilno strukturirano zaporedje operatorjev in operandov. Vsak izraz ima svojo vrednost, npr. celo število, realno število, znak itd.
- Spremenljivka je poimenovan prostorček v pomnilniku, ki hrani neko vrednost. Vsako spremenljivko moramo deklarirati, torej podati njen tip in njeno ime.
- S pomočjo prireditvenega stavka lahko v spremenljivko vpišemo poljubno vrednost, ki pripada tipu spremenljivke. Spremenljivki lahko vrednost določimo že ob deklaraciji. Spremenljivka, ki nima vrednosti, je nedefinirana, tista, ki jo ima, pa je definirana.

---

<sup>5</sup>Ena od zelo redkih izjem je  $\text{\TeX}$  (Knuth, 1986), podstat sistema  $\text{\LaTeX}$  (Mittelbach in sod., 2004), v katerem je napisana ta knjiga.

- Vsaka vrednost (pa tudi vsaka spremenljivka in vsak izraz) ima svoj tip. Tip podaja vrsto podatka in razpon njegovih možnih vrednosti. Java ponuja štiri celoštevilске in dva realnoštevilška tipa, spoznali pa smo še tipa za predstavitev znakov (char) in nizov (String). Znaki se v javi obravnavajo kot cela števila.
- V prireditvenem stavku mora biti tip izraza na desni strani priredljiv tipu spremenljivke na levi strani. V nekaterih primerih (npr. pri številskih tipih) si lahko pomagamo z izrecno pretvorbo tipa.
- Stavek je osnovna izvršljiva enota programa. Vsak stavek ima svoj učinek.
- Spremenljivke, deklarirane v nekem bloku, so vidne zgolj v tistem bloku. To vključuje morebitne vgnezdene bloke.
- Podatki, ki jih program pridobi iz okolja, tvorijo njegov vhod, tisti, ki jih okolju posreduje, pa njegov izhod. V tej knjigi bomo delali le s standardnim vhodom in standardnim izhodom. Standardni vhod je v osnovi tipkovnica, standardni izhod pa zaslon, vendar lahko oba preusmerimo (denimo na datoteko).
- Programe lahko preverjamo na različne načine, najpogosteje pa si pomagamo s testnimi primeri — primeri vhodov in pripadajočih izhodov. Na ta način lahko postopek preverjanja avtomatiziramo, še vedno pa se moramo zavedati, da preveriti ni enako kot dokazati.

### ***Iz profesorjevega kabineta***

Leopold Doberšek, čislani profesor na ugledni računalniški fakulteti, trdi, da je vrednosti celoštevilskih spremenljivk *a* in *b* med seboj mogoče zamenjati enostavno takole:

```
a = b;
b = a;
```

Jože Slapšak, njegov zvesti asistent, mu plaho odvrne, da mu brez pomožne spremenljivke žal ne bo uspelo. V podkrepitev svoje trditve pokaže lastno rešitev problema.

Pristopi še Genovefa Javornik, docentka na isti fakulteti, navrže, da se moti tudi asistent, in postreže z rešitvijo, ki med seboj zamenja vrednosti dveh celoštevilskih spremenljivk brez uvedbe pomožne spremenljivke.

Kakšno rešitev je predlagal asistent in kakšno docentka?



## 3 Krmilni konstrukti

S prireditvenimi stavki in vhodno-izhodnimi operacijami lahko rešimo vrsto zanimivih problemov, a še zdaleč ne vseh. Nabor rešljivih problemov pa močno povečamo, če svojo programersko orodjarno obogatimo s *pogojnimi stavki* in *zankami*. S prvimi lahko dosežemo, da se določen kos programa izvede samo v primeru, če je izpolnjen določen pogoj, z zankami pa lahko izbrani odsek programa poljubno mnogokrat izvedemo. Na primer, lahko ga ponavljamo tako dolgo, dokler je izpolnjen določen pogoj.

### 3.1 Zaporedje stavkov

Tukaj ni nobenih skrivnosti. Zaporedje stavkov se izvede — zaporedno. V kodi

```
stavek1  
stavek2  
...  
stavekn
```

se torej najprej izvede *stavek<sub>1</sub>*, nato *stavek<sub>2</sub>*, ..., nazadnje pa *stavek<sub>n</sub>*.

### 3.2 Pogojni stavek

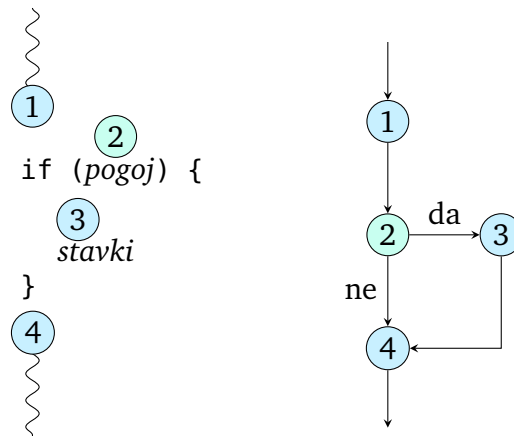
Pogojni stavek nam omogoča, da določen del programa izvedemo samo v primeru, če je izpolnjen določen pogoj. Pogojne stavke pravzaprav srečujemo vsak dan. Če je današnji dan sobota, lahko spim malo dlje. Če nimajo masla, kupi margarino. Če dežuje, ostanem doma, sicer pa grem na sprehod.

#### 3.2.1 if in if-else

Pogojni stavek ima dve obliki. Prva je taka:

```
if (pogoj) {  
    stavki  
}
```

Zapis *pogoj* predstavlja nek logičen pogoj, zapis *stavki* pa zaporedje stavkov. Ta oblika pogojnega stavka se izvede takole: če je *pogoj* izpolnjen, se *stavki* izvršijo, sicer pa se enostavno preskočijo (in izvajalnik takoj preide na stavek, ki sledi pogojnemu). Slika 3.1 prikazuje diagram poteka za prvo obliko pogojnega stavka.



**Slika 3.1** Izvajanje prve oblike pogojnega stavka (brez odseka `else`).

Druga oblika pogojnega stavka izgleda tako:

```
if (pogoj) {
    stavki1
} else {
    stavki2
}
```

Če je *pogoj* izpolnjen, se izvršijo *stavki*<sub>1</sub>, sicer pa se izvedejo *stavki*<sub>2</sub>. Ko se eno ali drugo zaporedje stavkov zaključi, se izvajanje nadaljuje s stavkom, ki sledi pogojnemu. Diagram poteka za to obliko je prikazan na sliki 3.2.

Če je zaporedje stavkov znotraj odseka `if` ali `else` sestavljeno iz enega samega stavka, lahko par zavitih oklepajev izpustimo. Na primer, namesto<sup>1</sup>

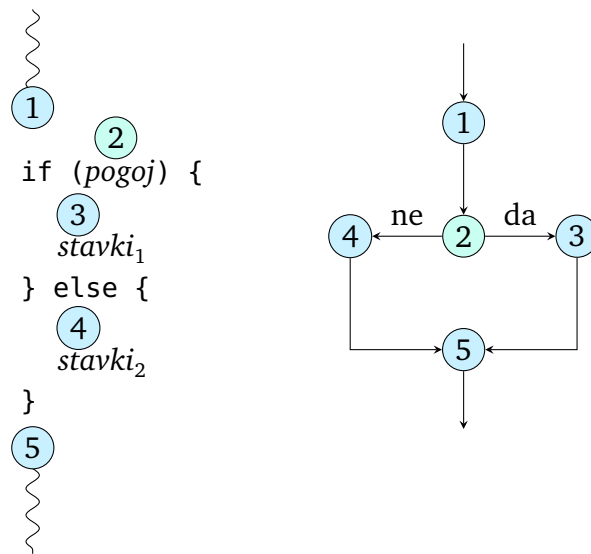
```
if (starost >= 18) {
    System.out.println("Vstopite, prosim!");
} else {
    System.out.println("Žal mi je, počakajte še malo ...");
}
```

lahko pišemo

```
if (starost >= 18)
```

<sup>1</sup>Operator `>=` pomeni »je večje ali enako« ( $\geq$  v matematičnem zapisu).





**Slika 3.2** Izvajanje druge oblike pogojnega stavka (z odsekom else).

```

System.out.println("Vstopite, prosim!");
else
    System.out.println("Žal mi je, počakajte še malo ...");

```

V tej knjigi bomo zavite oklepaje uporabljali v vseh primerih, saj menimo, da na ta način zmanjšamo možnost napak, povezanih s strukturo programa.

**Primer 3.1.** Napišimo program, ki prebere dve števili in izpiše, katero od njiju (prvo ali drugo) je večje.

*Rešitev.* Poskusimo takole. Preberemo števili in ju shranimo v spremenljivki (npr. prvo in drugo), nato pa preverimo, ali je prvo število večje od drugega. Če je, to dejstvo tudi obelodanimo, sicer pa izpišemo, da je drugo število večje od prvega:

```

// koda 3.1 (krmilniKonstrukti/Vecje1.java)
import java.util.Scanner;

public class Vecje1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Vnesite prvo število: ");
        int prvo = sc.nextInt();
        System.out.print("Vnesite drugo število: ");
        int drugo = sc.nextInt();
    }
}

```

```

    if (prvo > drugo) {           // (1)
        System.out.println("Prvo število je večje.");
    } else {
        System.out.println("Drugo število je večje.");
    }
}

```

Program prevedimo in ga nekajkrat poženimo. Vidimo, da pravilno deluje, če sta števili različni. Kaj pa se zgodi, če sta števili enaki?

```

Vnesite prvo število: 5
Vnesite drugo število: 5
Drugo število je večje.

```

Seveda: ker pogoj v vrstici (1) ni izpolnjen, se izvrši odsek `else`, zato program izpiše, da je drugo število večje. □

Pomanjkljiv je v resnici opis problema, saj predvideva samo dva scenarija namesto treh. Boljši opis bi bil tak:

**Primer 3.2.** Napišimo program, ki prebere dve števili in izpiše, katero od njiju je večje (če sta števili različni), če sta števili enaki, pa naj izpiše, da sta enaki.

*Rešitev.* Popravljenе naloge se lahko lotimo takole. Najprej preverimo, ali je prvo število večje od drugega. Če je, izpišemo ustrezno besedilo, sicer pa imamo spet dve možnosti. Preverimo, ali je prvo število manjše od drugega; če je, to tudi sporočimo, sicer pa nam ostane le še možnost, da sta števili enaki. Velik del programa ostane enak, spremeni se le pogojni stavek:

```

// koda 3.2 (krmilniKonstrukti/Vecje2.java)
if (prvo > drugo) { // (1)
    System.out.println("Prvo število je večje.");
} else { // (2)
    if (prvo < drugo) { // (3)
        System.out.println("Drugo število je večje.");
    } else { // (4)
        System.out.println("Števili sta enaki.");
    }
}

```

Če uporabnik sedaj vnese enaki števili, pogoj v vrstici (1) ni izpolnjen, zato se odsek `if` preskoči in se prične izvajati odsek `else` (vrstica (2)). Pogoj v vrstici (3) prav tako ni izpolnjen, zato skočimo na notranji odsek `else` (vrstica (4)) in izpišemo, da sta števili enaki. □

### 3.2.2 else if

Napovedali smo, da bomo pri odsekih `if` in `else` dosledno pisali zavite oklepaje, tudi ko bo njihovo telo sestavljeno iz enega samega stavka. No, v enem primeru bomo naredili izjemo. Če odsek `else` ne vsebuje nič drugega kot pogojni stavek, tega stavka ne bomo obdali z oklepaji, pa tudi zamaknili ga ne bomo. Na primer, kodo 3.2 bomo zapisali takole:

```
// koda 3.3 (krmilniKonstrukti/Vecje3.java)
if (prvo > drugo) {
    System.out.println("Prvo število je večje.");
} else if (prvo < drugo) {
    System.out.println("Drugo število je večje.");
} else {
    System.out.println("Števili sta enaki.");
}
```

Program lahko sedaj preberemo na sledeči način: če je prvo število večje od drugega, to izpiši, sicer pa preveri, ali je prvo število manjše od drugega; če je, to izpiši, sicer pa izpiši, da sta števili enaki.

Koda 3.3 je poseben primer *verige pogojnih stavkov*:

```
if (pogoj1) {
    stavki1
} else if (pogoj2) {
    stavki2
} else if (pogoj3) {
    stavki3
...
} else {
    stavkin
}
```

V tej verigi se izvrši *natanko eno* od zaporedij stavkov  $stavki_1, \dots, stavki_n$ . Če je  $pogoj_1$  izpolnjen, se izvršijo  $stavki_1$ , vse ostalo pa se preskoči. Če  $pogoj_1$  ni izpolnjen,  $pogoj_2$  pa je, se izvršijo  $stavki_2$ , vse ostalo pa se preskoči. Če ni izpolnjen niti prvi niti drugi pogoj, tretji pa je, se izvršijo samo  $stavki_3$ . Zaporedje  $stavki_n$  se izvrši samo v primeru, če ni izpolnjen nobeden od pogojev.

**Primer 3.3.** Napišimo program, ki prebere število točk (0–100) in ga po sledečem kriteriju pretvori v oceno od 5 do 10: 0–49 → 5, 50–59 → 6, 60–69 → 7, 70–79 → 8, 80–89 → 9, 90–100 → 10.

*Rešitev.* Ko preberemo število točk, preverimo, ali je enako vsaj 90. Če je, imamo

oceno 10, sicer pa preverimo, ali je število točk enako najmanj 80. Če to drži, je ocena enaka 9, sicer pa preverimo, ali je število točk enako najmanj 70 itd.

```
// koda 3.4 (krmilniKonstrukti/Ocena.java)
import java.util.Scanner;

public class Ocena {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Vnesite število točk: ");
        int tocke = sc.nextInt();

        int ocena = 0;           // (1)
        if (tocke >= 90) {
            ocena = 10;
        } else if (tocke >= 80) {
            ocena = 9;
        } else if (tocke >= 70) {
            ocena = 8;
        } else if (tocke >= 60) {
            ocena = 7;
        } else if (tocke >= 50) {
            ocena = 6;
        } else {
            ocena = 5;
        }
        System.out.println("Ocena: " + ocena);
    }
}
```

Ker spremenljivka *ocena* v verigi pogojnih stavkov v vsakem primeru dobi neko vrednost, nam ji v vrstici (1) ne bi bilo treba prirediti začetne vrednosti; lahko bi jo zgolj deklarirali. Po načelih dobre prakse pa se izogibamo deklaracijam brez inicializacije.<sup>2</sup> □

**Naloga 3.1** Kaj izpiše koda 3.4, če uporabnik vnese število točk izven intervala [0, 100]? Program dopolnite tako, da bo v takem primeru javil napako.

**Naloga 3.2** Če je število točk v intervalu [50, 99], ga lahko v oceno pretvorimo brez uporabe pogojnega stavka. Kako?

<sup>2</sup>V javi to sicer ni tako ključno, povsem drugače pa je v jezikih C in C++, kjer po izvedbi stavka `int a` spremenljivka `a` ni nedefinirana, ampak ima neko določeno (a vnaprej neznano) vrednost.

### 3.3 Logični izrazi

Pogoj v glavi pogojnega stavka mora biti *logični izraz*. To je izraz, katerega vrednost je bodisi *logična resnica* (`true`) bodisi *logična neresnica* (`false`). Primer logičnega izraza je izraz `5 < 7`. Ker je število 5 zares manjše od števila 7, je vrednost tega izraza enaka `true`. Vrednost izraza `10 > 10` pa je enaka `false`, saj število 10 ni večje od samega sebe.

Vrednosti `true` in `false` pripadata tipu `boolean`, poimenovanemu po angleškem matematiku in logiku Georgeu Boolu. Tip `boolean` nima nobene druge vrednosti. Podobno kot pri številskih tipih lahko deklariramo spremenljivke tipa `boolean` in jim prirejamo vrednosti tega tipa:

```
boolean a = 5 < 7;
boolean b = 10 > 10;
boolean c = true;
System.out.println(a); // true
System.out.println(b); // false
System.out.println(c); // true
```

Pogosta podvrsta logičnih izrazov so *primerjalni izrazi*. Splošna oblika tovrstnih izrazov je

*izraz<sub>1</sub> op izraz<sub>2</sub>*

pri čemer *op* označuje *primerjalni operator*. Java pozna šest primerjalnih operatorjev; zbrani so v tabeli 3.1.

**Tabela 3.1** Primerjalni operatorji.

Operator	Pomen	Primer izraza	Vrednost izraza
<code>==</code>	je enako kot	<code>5 == 5</code>	<code>true</code>
<code>!=</code>	je različno od	<code>5 != 5</code>	<code>false</code>
<code>&gt;</code>	je večje od	<code>5 &gt; 5</code>	<code>false</code>
<code>&gt;=</code>	je večje ali enako	<code>5 &gt;= 5</code>	<code>true</code>
<code>&lt;</code>	je manjše od	<code>5 &lt; 5</code>	<code>false</code>
<code>&lt;=</code>	je manjše ali enako	<code>5 &lt;= 5</code>	<code>true</code>

Bodimo pozorni na razliko med operatorjema `=` in `==`. Operator `=` je *prireditveni operator*, ki izračuna izraz na desni strani in njegovo vrednost shrani v spremenljivko na levi strani. Koda `a = b` praviloma nastopa kot samostojen stavek, čeprav ima, kot smo videli v razdelku 2.5, tudi svojo vrednost. Operator `==` v izrazu `a == b` pa zgolj preveri, ali imata njegova operanda enako vrednost. Če jo imata, je rezultat

celotnega izraza `true`, sicer pa `false`. Izraz `a == b` nima nobenega učinka, zato ne more nastopati kot samostojen stavek.

Logične izraze lahko med seboj povezujemo z *logičnimi operatorji*. Java pozna tri logične operatorje: *logični in* (`&&`), *logični ali* (`||`) in *logični ne* (`!`). Izraz `p && q` je resničen natanko tedaj, ko sta izraza `p` in `q` oba resnična, izraz `p || q` pa je resničen natanko tedaj, ko je resničen vsaj eden od izrazov `p` in `q`. Izraz `!p` je resničen, ko je izraz `p` neresničen, in obratno. Ta pravila so povzeta v tabeli 3.2.<sup>3</sup>

**Tabela 3.2** Logični operatorji.

p	q	p && q	p    q	!p
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Operator `!` ima prednost pred operatorjem `&&`, ta pa pred operatorjem `||`. Na primer, izraz

```
a < b || !(c == d) && f >= g
```

ima vrednost `true` natanko tedaj, ko velja vsaj eno od sledečega: (1) `a < b`; (2) `c != d` in hkrati `f >= g`.

**Primer 3.4.** Napišimo program, ki ugotovi, ali je podano leto prestopno.

*Rešitev.* Leto je prestopno, če je izpolnjen eden od sledečih pogojev:

- Leto je deljivo s 400.
- Leto je deljivo s 4, vendar pa ni deljivo s 100.

Kako preverimo deljivost? Število `a` je deljivo s številom `b` natanko tedaj, ko se deljenje števila `a` s številom `b` izide, torej ko je ostanek pri deljenju enak 0. Naš program lahko potemtakem napišemo takole:

```
// koda 3.5 (krmilniKonstrukti/PrestopnoLeto.java)
import java.util.Scanner;

public class PrestopnoLeto {
    public static void main(String[] args) {
```

<sup>3</sup>Poleg pojmov *in*, *ali* in *ne* se uporabljajo tudi pojmi *konjunkcija* (`&&`), *disjunkcija* (`||`) in *negacija* (`!`).

```

Scanner sc = new Scanner(System.in);
System.out.print("Vnesite leto: ");
int leto = sc.nextInt();

if (leto % 400 == 0 || leto % 4 == 0 && leto % 100 != 0) {
    System.out.println("Leto je prestopno.");
} else {
    System.out.println("Leto ni prestopno.");
}
}
}

```

Operator `||` sicer pokriva tudi možnost, da sta izpolnjena oba pogoja, vendar pa se to tako ali tako ne more zgoditi. □

Operatorja `&&` in `||` se izvajata *kratkostično*. Če ima izraz  $p$  v izrazu  $p \ \&\& \ q$  vrednost `false`, potem se izraz  $q$  sploh ne bo izračunal, saj je jasno, da bo vrednost celotnega izraza `false`. Podobno velja za izraz  $p \ || \ q$ : če ima izraz  $p$  vrednost `true`, je celoten izraz lahko samo resničen, zato se izraz  $q$  ne bo izračunal.

Zaradi kratkostičnosti se lahko program v nekaterih primerih obnaša drugače, kot bi se, če bi se vsi izrazi izračunali. Na primer, če ne bi bilo kratkostičnosti, bi sledeči izraz zaradi deljenja z ničlo sprožil izjemo:

```
3 == 2 && 1 / 0 == 0
```

**Naloga 3.3** Naj bosta  $a$  in  $b$  logična izraza. Izrazite logično implikacijo (če velja  $a$ , potem velja  $b$ ) s pomočjo javinih logičnih operatorjev.

**Naloga 3.4** Pri katerih parih vrednostih spremenljivk  $a$  in  $b$  bo imel izraz  $((a == b) == (a == b)) != (a == b)$  vrednost `true`?

**Naloga 3.5** Kaj izpiše sledeči izsek kode?

```

boolean a = true;
boolean b = false;
if (a = b) {    // pozor!
    System.out.println("DA");
} else {
    System.out.println("NE");
}

```

### 3.4 Zanka *while*

Zdolgočaseni Janezek je med poukom znova klepetal s sosedom. Učiteljica mu je za kazen naložila, naj stokrat napiše stavek *Ne smem klepetati med poukom*. Vrli Janezek, ne bodi len, prižge računalnik, odpre svoj omiljeni programerski urejevalnik in prične tipkati:

```
// koda 3.6 (krmilniKonstrukti/NeSmemKlepetati.java)
public class NeSmemKlepetati {
    public static void main(String[] args) {
        int stevec = 1;
        while (stevec <= 100) {
            System.out.println("Ne smem klepetati med poukom.");
            stevec = stevec + 1;
        }
    }
}
```

Program prevede in požene, dobljeni izpis pa še natisne. Učiteljica vidi sto ponovitev stavka *Ne smem klepetati med poukom* in je z Janezkom vsaj do naslednjega incidenta zadovoljna.

Pravkar smo napisali svojo prvo *zanko*, programski konstrukt, s pomočjo katerega lahko dosežemo, da se določen odsek kode večkrat izvede. Java nam ponuja tri tipe zank, v tem razdelku pa se bomo osredotočili na zanko *while*. Ta zanka ima v splošnem tako obliko:

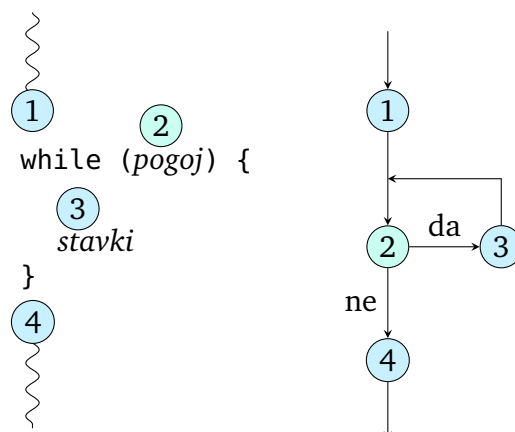
```
while (pogoj) {
    stavki
}
```

Če *pogoj* v glavi zanke ni izpolnjen, se telo zanke preskoči in se izvede stavek, ki sledi zanki. Če je pogoj izpolnjen, pa se najprej izvršijo *stavki*, nato pa se izvajalnik vrne na preverjanje pogoja in ponovi postopek (če je pogoj še vedno izpolnjen, vnovič izvrši stavke v telesu zanke, preveri pogoj itd., v nasprotnem primeru pa stavke v telesu zanke preskoči in s tem zapusti zanko). Zanka *while* torej izvaja podano zaporedje stavkov, *dokler* je izpolnjen podani pogoj. Diagram poteka zanjo je prikazan na sliki 3.3.

Janezkov program za podrobnejšo analizo ni najbolj pripraven, zato ga bomo nekoliko spremenili:

```
// koda 3.7 (krmilniKonstrukti/PrimerWhile.java)
int i = 1;                                // (1)
while (i <= 3) {                          // (2)
```





Slika 3.3 Izvajanje zanke while.

```

System.out.println("a"); // (3)
i = i + 1;                // (4)
}
System.out.println("b");  // (5)

```

Tabela 3.3 prikazuje natančen potek izvajanja programa. Vidimo, da se telo zanke trikrat izvede (ko ima spremenljivka  $i$  vrednosti 1, 2 in 3, pri vrednosti 4 pa pogoj v glavi ni več izpolnjen), kar pomeni, da se bo najprej trikrat izpisala črka a, ko se zanka zaključi, pa še črka b.

**Primer 3.5.** Napišimo program, ki prebere števili  $a$  in  $b$  in izpiše zaporedje števil od  $a$  do  $b$ . Na primer, pri  $a = 5$  in  $b = 8$  pričakujemo takšen izpis:

```

5
6
7
8

```

**Rešitev.** Brez zanke ne bo šlo, saj ne vemo vnaprej, kakšna bosta  $a$  in  $b$ , in zato ne vemo, koliko vrstic bo program izpisal. Lotimo se torej zanke. Pogosto je lažje, če najprej napišemo njeno telo in šele potem glavo. Kaj torej sodi v telo? Tisto, kar se bo ponavljalo (izvršilo v vsakem obhodu zanke). Pri našem problemu se ponavlja izpisovanje števil. Smiselno je, da v vsakem obhodu izpišemo po eno število. (To sicer ne bi bilo nujno, je pa najenostavneje.) Število, ki ga izpisujemo, se spreminja, zato ga bomo hranili v spremenljivki; recimo ji *stevilo*. Spremenljivko *stevilo* na začetku — že pred vstopom v zanko — nastavimo na vrednost  $a$  (to število se bo najprej izpisalo), nato pa ga v vsakem obhodu zanke povečamo za 1:

**Tabela 3.3** Izvajanje kode 3.7.

Vrstica	Dogodek	Vrednost <i>i</i>
(1)	Inicializiraj <i>i</i>	1
(2)	Pogoj je izpolnjen, zato pojdi v telo zanke	
(3)	Izpiši <i>a</i>	
(4)	Povečaj <i>i</i>	2
	Vrni se v glavo zanke	
(2)	Pogoj je izpolnjen, zato pojdi v telo zanke	
(3)	Izpiši <i>a</i>	
(4)	Povečaj <i>i</i>	3
	Vrni se v glavo zanke	
(2)	Pogoj je izpolnjen, zato pojdi v telo zanke	
(3)	Izpiši <i>a</i>	
(4)	Povečaj <i>i</i>	4
	Vrni se v glavo zanke	
(2)	Pogoj ni izpolnjen, zato preskoči telo zanke	
(5)	Izpiši <i>b</i>	

```
int stevilo = a;
while (pogoj) {
    System.out.println(stevilo);
    stevilo = stevilo + 1;
}
```

Pogoj v glavi zanke moramo nastaviti tako, da se bo spremenljivka izpisovala, dokler njena vrednost ne bo presegla vrednosti *b*. Napišimo celoten program:

```
// koda 3.8 (krmilniKonstrukti/Zaporedje.java)
import java.util.Scanner;

public class Zaporedje {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Vnesite začetno število: ");
        int a = sc.nextInt();
        System.out.print("Vnesite končno število: ");
        int b = sc.nextInt();

        int stevilo = a;
```

```

        while (stevilo <= b) {
            System.out.println(stevilo);
            stevilo = stevilo + 1;    // (1)
        }
    }
}

```

Ko bo spremenljivka *stevilo* v vrstici (1) dobila vrednost  $b + 1$ , pogoj v glavi zanke ne bo več izpolnjen, zato bo program izstopil iz zanke.  $\square$

Kaj se zgodi, če v kodi 3.8 izpustimo vrstico (1)? Če bo  $a \leq b$ , bo zankni pogoj *vedno* izpolnjen, zato se zanka *nikoli* ne bo prekinila. Program bo izpisoval število *a* in se ne bo ustavil (v žargonu rečemo, da se *zacikla*). Takšne reči se dogajajo tudi izkušenim programerjem. Ker se bodo tudi vam, naj takoj povemo, da lahko program, ki se izvaja v terminalu, v večini primerov prekinemo s kombinacijo tipk Ctrl-C.

Kako pa neskončno zanko zaznamo in odpravimo? Prvi pripomoček so pomožni izpisi. V zanko, za katero sumimo, da se ne ustavi, vstavimo klic `System.out.println(...)` s poljubnim argumentom; če je zanka res neskončna, bomo to takoj videli. Če ne vemo, katera zanka bi lahko bila neskončna, pač poskusimo z vsemi. Ko problematično zanko najdemo, poskušamo ugotoviti, zakaj se ne ustavi. Zopet si lahko pomagamo z vmesnimi izpisi, le da tokrat izpisujemo vrednosti spremenljivk, ki nastopajo v zanki. S podobnimi prijemi se lahko lotimo tudi drugih vrst napak.

**Naloga 3.6** Brez poganjanja kode 3.8 odgovorite na sledeči vprašanji: (1) Kaj se izpiše v primeru  $a = b$ ? (2) Kaj se izpiše v primeru  $a > b$ ?

**Naloga 3.7** Koliko vrstic izpiše sledeča zanka?

```

int i = 30;
while (i >= 0) {
    System.out.println(i);
    i = i - 3;
}

```

**Naloga 3.8** Profesor Doberšek trdi, da je zanka

```

int i = 1;
while (i >= 0) {
    i = i + 1;
}

```

neskončna. Ima prav?

**Naloga 3.9** Izpis sledeče kode izrazite s številom  $n$ .

```
int a = 0;
int b = 0;
int i = 0;
while (a < n) {
    b = b + 1;
    a = a + b;
    i = i + 1;
}
System.out.println(i);
```

### 3.4.1 Sprotno seštevanje

Napišimo tri programe, ki bi nam lahko prišli prav v vsakdanjem življenju, denimo v trgovini.

**Primer 3.6.** Naš prvi program v tem razdelku bo bral števila in jih sproti sešteval, dokler ne bo vsota preseгла vrednosti 42. Na primer:

```
Vnesite število: 6
6
Vnesite število: 7
13
Vnesite število: 4
17
Vnesite število: 10
27
Vnesite število: 9
36
Vnesite število: 10
46
```

*Rešitev.* Tudi tokrat ne moremo brez zanke. Najbolj naravno je zanko nastaviti tako, da bo program v vsakem obhodu prebral število, ga prištel dosedanji vsoti in izpisal novo vsoto. Vsoto dosedanjih števil bomo hranili v spremenljivki *vsota*. Začetna vsota je enaka 0, po vsakem branju števila pa se poveča za to število (nova vsota = dosedanja vsota + prebrano število). Zanka se izvaja, dokler vsota ne preseže 42.

```
// koda 3.9 (krmilniKonstrukti/SprotnoSeštevanje1.java)
import java.util.Scanner;
```

```

public class SprotnoSestevanje1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int vsota = 0;    // vsota doslej prebranih števil
        while (vsota <= 42) {
            System.out.print("Vnesite število: ");
            int vnos = sc.nextInt();
            vsota = vsota + vnos;
            System.out.println(vsota);
        }
    }
}

```



**Primer 3.7.** Napišimo program, ki prebere pet števil in sproti izpisuje njihovo vsoto. Na primer:

```

Vnesite število: 10
10
Vnesite število: 15
25
Vnesite število: 7
32
Vnesite število: 20
52
Vnesite število: 5
57

```

*Rešitev.* Pri tej nalogi bi načeloma lahko shajali brez zanke, saj je število korakov fiksno in vnaprej znano. Vendar pa bo program precej kompaktnejši, predvsem pa lažje nadgradljiv, če si bomo pomagali z zanko. V posameznih obhodih zanke počnemo natanko iste reči kot v prejšnjem primeru (preberemo število in ga prištejemo k tekoči vsoti, nato pa tekočo vsoto izpišemo), le izstopni pogoj bo drugačen. Pogoj tokrat ni več določen z vrednostjo vsote, ampak s številom uporabnikovih vnosov (oziroma obhodov zanke). To pomeni, da potrebujemo posebno spremenljivko, ki bo štela vnose. Taki spremenljivki rečemo *števlec*. Števlec inicializiramo na 0, po vsakem vnosu pa ga povečamo za 1. Ko doseže vrednost 5, je čas za prekinitev zanke.

Navajamo samo osrednji del programa:

```

// koda 3.10 (krmilniKonstrukti/SprotnoSestevanje2.java)
int stVnosov = 0;    // števlec vnosov

```

```

int vsota = 0;           // vsota doslej prebranih števil
while (stVnosov < 5) {
    System.out.print("Vnesite število: ");
    int vnos = sc.nextInt();
    stVnosov = stVnosov + 1;
    vsota = vsota + vnos;
    System.out.println(vsota);
}

```

□

**Primer 3.8.** Napišimo program, ki bere števila in sproti izpisuje njihovo vsoto, dokler ne prebere pet števil oziroma dokler vsota ne preseže 42.

*Rešitev.* Ta naloga je očitno kombinacija prejšnjih dveh. V zanki beremo števila in jih seštevamo v spremenljivko *vsota*, obenem pa štejemo uporabnikove vnose. Zanka se zaključi, ko vsota preseže 42 *ali* pa ko števec vnosov doseže 5. Z drugimi besedami: zanka se nadaljuje, dokler je vsota manjša ali enaka 42 *in* je števec vnosov manjši od 5.

```

// koda 3.11 (krmilniKonstrukti/SprotnoSestevanje3.java)
int stVnosov = 0;
int vsota = 0;
while (stVnosov < 5 && vsota <= 42) {
    System.out.print("Vnesite število: ");
    int vnos = sc.nextInt();
    stVnosov = stVnosov + 1;
    vsota = vsota + vnos;
    System.out.println(vsota);
}

```

□

**Naloga 3.10** Programe, ki smo jih napisalni v tem razdelku, dopolnite tako, da se bo pri vsakem pozivniku izpisala tudi zaporedna številka vnesenega števila (Vnesite 1. število:, Vnesite 2. število: itd.).

**Naloga 3.11** Napišite program, ki prebere števili  $a$  in  $m$  in izpiše vse večkratnike števila  $a$  (torej  $a$ ,  $2a$ ,  $3a$  ...), ki niso večji od  $m$ .

**Naloga 3.12** Napišite program, ki prebere števili  $a$  in  $m$  in izpiše vse potence števila  $a$  (začenši z  $a^0$ ), ki niso večje od  $m$ .

**Naloga 3.13** Napišite program, ki prebere število  $n$  in izpiše, kolikokrat moramo število zaporedoma celoštevilsko deliti z 2, da dobimo 0. (Dobljeni rezultat je tesno povezan z vrednostjo  $\log_2 n$ .)

### 3.4.2 Vsota vseh vhodnih števil

**Primer 3.9.** Napišimo program, ki izpiše vsoto vseh vhodnih števil. Predpostavimo, da so na vhodu zapisana zgolj cela števila v obsegu tipa `int`.

*Rešitev.* Tokrat ne vemo vnaprej, koliko števil imamo na vhodu, pa tudi na pogoje, kot je npr. zgornja meja tekoče vsote, se ne moremo zanašati. Kako lahko potem zaznamo, kdaj naj prenehamo z branjem vhodnih števil? K sreči si lahko pomagamo s klicem `sc.hasNextInt()` (`sc` je spremenljivka tipa `Scanner`), ki vrne `true`, če na vhodu še obstaja kakšno število tipa `int`, in `false`, če to ne drži. Pogoju zapišemo v glavo zanke *while*:

```
// koda 3.12 (krmilniKonstrukti/VsotaVhoda.java)
import java.util.Scanner;

public class VsotaVhoda {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int vsota = 0;
        while (sc.hasNextInt()) {
            int stevilo = sc.nextInt();
            vsota = vsota + stevilo;
        }
        System.out.println(vsota);
    }
}
```

Torej: dokler na vhodu še obstaja kakšno število tipa `int`, ga preberi in prištej tekoči vsoti, ko pa to ne velja več, prekini zanko in izpiši dosedanjo vsoto. □

Program lahko preizkusimo tako, da si pripravimo vhodno datoteko (na primer `stevila.txt`) in vanjo zapišemo števila, nato pa program poženemo s preusmeritvijo standardnega vhoda na to datoteko (razdelek 2.10):

```
terminal> java VsotaVhoda < stevila.txt
```

Števila v vhodni datoteki so lahko med seboj ločena s poljubnim nepraznim zaporedjem presledkov, tabulatorjev in prelomov vrstice, saj klic `sc.nextInt()` preskakuje vsa takšna ločila.

Še manj dela imamo z lupinskim ukazom `echo`:

```
terminal> echo 10 7 5 14 | java VsotaVhoda
36
```

**Naloga 3.14** Napišite program, ki izpiše število vhodnih števil. Predpostavite, da so na vhodu zapisana zgolj cela števila v obsegu tipa `int`.

**Naloga 3.15** Napišite program, ki izpiše predzadnje vhodno število. Tudi to pot predpostavite, da so na vhodu zapisana zgolj cela števila v obsegu tipa `int`.

**Naloga 3.16** Napišite program, ki izpiše število, sestavljeno iz števk, zapisanih na vhodu. Predpostavite, da so na vhodu zapisane zgolj števke (števila iz množice  $\{0, 1, \dots, 9\}$ ), da prva števka ni enaka 0 in da števki ni več kot 18. Na primer, če so na vhodu zapisane števke 5, 0, 2 in 7, naj program izpiše število 5027. Program naj izhodno število sestavi z uporabo celoštevilskih računskih operacij, ne z lepljenjem nizov ali veriženjem stavkov `System.out.print`.

### 3.5 Sestavljeni prireditveni operatorji

Ena od pogostejših operacij pri programiranju je prištevanje vrednosti spremenljivki. Ker je zapis

```
spremenljivka = spremenljivka + izraz
```

nekoliko dolgovezen, še zlasti če ima *spremenljivka* dolgo ime, ga lahko okrajšamo v

```
spremenljivka += izraz
```

Konstrukt *spremenljivka* += *izraz* lahko deluje kot samostojen stavek, saj ima učinek (spremenljivka na levi se poveča za vrednost izraza na desni), ali pa kot izraz znotraj nekega stavka ali večjega izraza; njegova vrednost je nova vrednost spremenljivke. Na primer:

```
int a = 3;
a += 2;
int b = (a += 4);
```

Spremenljivka *a* ima po prvem stavku vrednost 3, po drugem 5, po tretjem pa 9. Spremenljivka *b* dobi vrednost 9. V tretjem stavku lahko oklepaje izpustimo.

Povečevanje spremenljivke za 1 je še posebej pogosto, saj se uporablja pri večini zank. To operacijo lahko zato zapisujemo še nekoliko krajše, in sicer na dva načina:

- s *prefiksnim* operatorjem ++ (++*spremenljivka*);
- s *postfiksni*m operatorjem ++ (*spremenljivka*++).



Izraza `++spremenljivka` in `spremenljivka++` imata enak učinek: vrednost spremenljivke povečata za 1. Razlikujeta pa se po vrednosti: vrednost izraza `++spremenljivka` je nova (povečana) vrednost spremenljivke, vrednost izraza `spremenljivka++` pa je njena prvotna vrednost. Izraza se zato med seboj razlikujeta le v primeru, ko nastopata kot del večjega izraza. Oglejmo si primer:

```
int a = 1;
int b = 1;
++a;    // a == 2
b++;    // b == 2
int c = ++a;    // a == 3, c == 3
int d = b++;    // b == 3, d == 2
```

Vidimo, da je spremenljivka `c` dobila novo, spremenljivka `d` pa staro vrednost spremenljivke na desni strani (`a` oziroma `b`). V obeh primerih pa se je spremenljivka na desni strani povečala za 1.

Po analogiji z operatorjem `+=` obstajajo tudi operatorji `-=`, `*=`, `/=` in `%=`. Na primer, zapis `a -= b` je okrajšava za `a = a - b`, izraz `a *= b` je enakovreden izrazu `a = a * b` itd. Poleg prefiksne in postfiksne operatorja `++` obstajata tudi prefiksni in postfiksni operator `--`. Izraza `--a` in `a--` zmanjšata vrednost spremenljivke `a` za 1, vendar pa je rezultat prvega izraza nova, rezultat drugega pa stara vrednost spremenljivke.

**Naloga 3.17** Kakšni sta vrednosti spremenljivk `a` in `b` po izvedbi sledečih stavkov?

```
int a = 10;
int b = a-- - --a;
```

## 3.6 Zanka do

Zanka *do* ima takšno obliko:

```
do {
    stavki
} while (pogoj);
```

Zanka je podobna zanki *while*, le da se najprej izvedejo *stavki*, šele nato se preveri *pogoj*. Če je pogoj izpolnjen, se ponovno izvedejo *stavki* in preveri *pogoj*, v nasprotnem primeru pa se zanka zaključi. Telo zanke se torej v vsakem primeru izvede vsaj enkrat. Diagram poteka zanke *do* je prikazan na sliki 3.4, tabela 3.4 pa podaja natančen potek sledeče kode:

```
// koda 3.13 (krmilniKonstrukti/PrimerDo.java)
```

```

int i = 1;                // (1)
do {
    System.out.println("a"); // (2)
    i++;                    // (3)
} while (i <= 3);          // (4)
System.out.println("b");   // (5)

```

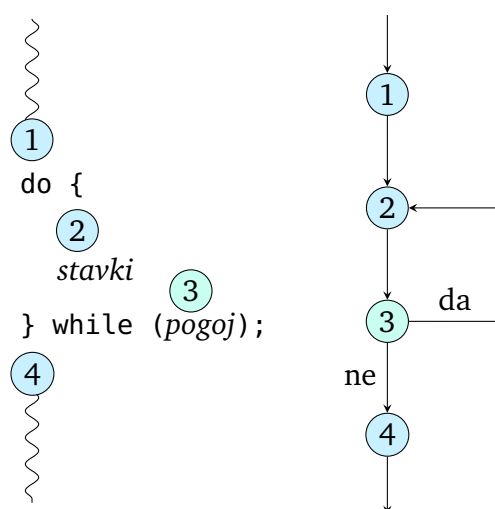
Slika 3.4 Izvajanje zanke *do*.

Tabela 3.4 Izvajanje kode 3.13.

Vrstica	Dogodek	Vrednost i
(1)	Inicializiraj i	1
(2)	Izpiši a	
(3)	Povečaj i	2
(4)	Pogoj je izpolnjen, zato ponovno izvedi telo zanke	
(2)	Izpiši a	
(3)	Povečaj i	3
(4)	Pogoj je izpolnjen, zato ponovno izvedi telo zanke	
(2)	Izpiši a	
(3)	Povečaj i	4
(4)	Pogoj ni izpolnjen, zato zapusti zanko	
(5)	Izpiši b	

Vsako zanko *do* lahko prepišemo v enakovredno zanko *while*, kljub temu pa je v nekaterih primerih rešitev z zanko *do* elegantnejša.

**Primer 3.10.** Napišimo program, ki uporabnika nadleguje tako dolgo, dokler ne vnese odgovora na vprašanje o življenju, veselju in sploh vsem (Adams, 1996).

*Rešitev.* Naloge se lahko lotimo z zanko *while*:

```
// koda 3.14 (krmilniKonstrukti/OdgovorWhile.java)
import java.util.Scanner;

public class OdgovorWhile {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Vnesite odgovor: ");
        int odgovor = sc.nextInt();
        while (odgovor != 42) {
            System.out.print("Vnesite odgovor: ");
            odgovor = sc.nextInt();
        }
    }
}
```

S tem programom ni nič narobe, vendar pa je rešitev z zanko *do* krajša:

```
// koda 3.15 (krmilniKonstrukti/OdgovorDo.java)
import java.util.Scanner;

public class OdgovorDo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        do { // (1)
            System.out.print("Vnesite odgovor: ");
            int odgovor = sc.nextInt();
        } while (odgovor != 42); // (2)
    }
}
```

Žal nam jo zagode prevajalnik, saj izpiše tole napako:

```
OdgovorDo.java:11: error: cannot find symbol
    } while (odgovor != 42);
```

```

      ^
symbol:  variable odgovor
location: class OdgovorDo

```

Težava nastane, ker je spremenljivka `odgovor` deklarirana znotraj para zavutih oklepajev v vrsticah (1) in (2), kar pomeni, da obstaja samo do zavitega zaklepaja. Če jo želimo uporabiti tudi v zančnem pogoju, jo moramo deklarirati že pred zanko:

```

int odgovor = 0;
do {
    System.out.print("Vnesite odgovor: ");
    odgovor = sc.nextInt();
} while (odgovor != 42);

```

□

**Naloga 3.18** Dopolnite program tako, da se bo zaključil po desetih vnosih, če uporabnik ne bo že prej vnesel pričakovanega števila.

**Naloga 3.19** Prepišite zanko *while* iz kode 3.11 v enakovredno zanko *do*.

### 3.6.1 Ugibanje naključnega števila

**Primer 3.11.** Napišimo program, ki prebere število  $m$ , izbere naključno število  $z$  intervala  $[1, m]$  in omogoči uporabniku, da ga ugiba, dokler ga ne ugame. Program naj po vsakem neuspešnem uporabnikovem poskusu izpiše, ali je njegovo izbrano število večje ali manjše od pravkar vnesenega. Na primer:

```

Vnesite zgornjo mejo: 50
Vaš poskus: 25
Izbrano število je večje.
Vaš poskus: 37
Izbrano število je manjše.
Vaš poskus: 31
Izbrano število je manjše.
Vaš poskus: 28
Izbrano število je večje.
Vaš poskus: 30
Izbrano število je manjše.
Vaš poskus: 29
Čestitke!

```

*Rešitev.* V tem programu bomo tvorili *naključna števila*. Naključna števila so pomemben element številnih iger. Težko si predstavljamo, recimo, legendarni Tetris,

pri katerem bi bila izbira naslednjega lika povsem predvidljiva. Naključna števila, kot jih običajno razumemo v računalništvu, sploh niso naključna, saj se računajo po (razmeroma enostavnem) računskem postopku, vendar pa opazovalec samo na podlagi spremljanja zaporedja tovrstnih števil ne more napovedati, katero bo naslednje. Zaradi tega jim pogosto pravimo *psevdonaključna* števila.

V javi lahko naključna števila tvorimo tako, da ustvarimo objekt razreda `Random` iz paketa `java.util`, nato pa nad tem objektom kličemo metodo `nextInt`. Objekt ustvarimo samo enkrat, metodo pa pokličemo vsakokrat, ko želimo ustvariti naključno število:

```
import java.util.Random;

public class ... {
    public static void main(String[] args) {
        ...
        // samo enkrat
        Random random = new Random( seme );
        ...
        // za vsako naključno število
        int stevilo = random.nextInt(n);
        ...
    }
}
```

Rezultat klica `random.nextInt(n)` je naključno število z intervala  $[0, n - 1]$ .

Pri izdelavi objekta razreda `Random` lahko po želji podamo argument, imenovan *seme*. Seme je število, ki enolično določa zaporedje naključnih števil na določenem intervalu. Na primer, sledeča koda bo vedno izpisala zaporedje števil 30, 63, 48, 84, 70, 25, 5, 18, 19 in 93, ne glede na to, kolikokrat in kje jo poženemo:

```
// koda 3.16 (krmilniKonstrukti/PrimerRandom.java)
Random random = new Random(42);
int i = 1;
while (i <= 10) {
    System.out.println(random.nextInt(100));
    i++;
}
```

Če bi kot seme podali število 43, bi dobili povsem drugačno zaporedje, a še vedno bi bilo pri vseh zagonih programa enako. Če semena ne podamo, ga izbere računalnik, in to tako, da bosta semeni v dveh zaporednih zagonih programa zelo verjetno različni, zato bodo tudi zaporedja naključnih števil med seboj zelo verjetno različna.

V programu za ugibanje števila semena ne bomo podali, saj ne želimo, da pro-

gram pri isti zgornji meji vedno izbere isto število. Po izbiri števila bo program vstopil v zanko, v kateri bo v vsakem obhodu prebral uporabnikov vnos, ga primerjal z izbranim številom in izpisal odnos med izbranim in vnesenim številom. Zanka se bo iztekla, ko bo uporabnik uganil izbrano število. Ker bo uporabnik v vsakem primeru vnesel vsaj eno število, bo zanka *do* priročnejša od zanke *while*:

```
// koda 3.17 (krmilniKonstrukti/Ugani.java)
import java.util.*;    // potrebujemo Scanner in Random

public class Ugani {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Vnesite zgornjo mejo: ");
        int meja = sc.nextInt();

        Random random = new Random();
        int izbrano = random.nextInt(meja) + 1;
        int poskus = 0;

        do {
            System.out.print("Vaš poskus: ");
            poskus = sc.nextInt();
            if (izbrano > poskus) {
                System.out.println("Izbrano število je večje.");
            } else if (izbrano < poskus) {
                System.out.println("Izbrano število je manjše.");
            }
        } while (poskus != izbrano);

        System.out.println("Čestitke!");
    }
}
```

□

**Naloga 3.20** Napišite izraz, katerega vrednost je naključno celo število z intervala  $[a, b]$ .

**Naloga 3.21** Program Ugani dopolnite tako, da bo po čestitki izpisal še število poskusov (vnosov).

**Naloga 3.22** Program Ugani dopolnite tako, da bo omogočal predčasen zaključek z vnosom negativnega števila (v tem primeru naj ničesar več ne izpiše).

**Naloga 3.23** V največ koliko poskusih bomo z optimalnim postopkom ugibanja zanesljivo uganili število z intervala  $[1, n]$ ?

**Naloga 3.24** Napišite program, ki rešuje obraten problem: uporabnik si izmisli število, program pa ga ugiba po optimalnem postopku. Uporabnik naj po vsakem poskusu vnese (recimo) 1, 0 oz.  $-1$  (če je računalnikov poskus premajhen, pravilen oz. prevelik).

**Naloga 3.25** Verjetnost, da ima izraz `random.nextInt(2)` vrednost 1, je enaka  $\frac{1}{2}$ . Kolikšna je verjetnost, da sledeči program izpiše število  $r$ ?

```
import java.util.Random;

public class Nakljucja {
    public static void main(String[] args) {
        Random random = new Random();
        int k = 0;
        int n = 0;
        while (k < 10) {
            k += random.nextInt(2);
            n++;
        }
        System.out.println(n);
    }
}
```

### 3.7 Zanka *for*

Številne zanke imajo sledečo obliko:

```
inicijalizacija
while (pogoj) {
    stavki
    posodobitev
}
```

Z zapisom *inicijalizacija* označujemo stavke, namenjene pripravi na vstop v zanko, z zapisom *posodobitev* pa stavke, s katerimi se pripravimo na vstop v naslednji obhod zanke. Zanko te oblike smo že srečali:

```
// koda 3.18
int i = 1;
```

```
while (i <= 3) {
    System.out.println("a");
    i = i + 1;
}
System.out.println("b");
```

V fazi inicializacije nastavimo števec *i* na vrednost 1, v fazi posodobitve pa ga povečamo za 1.

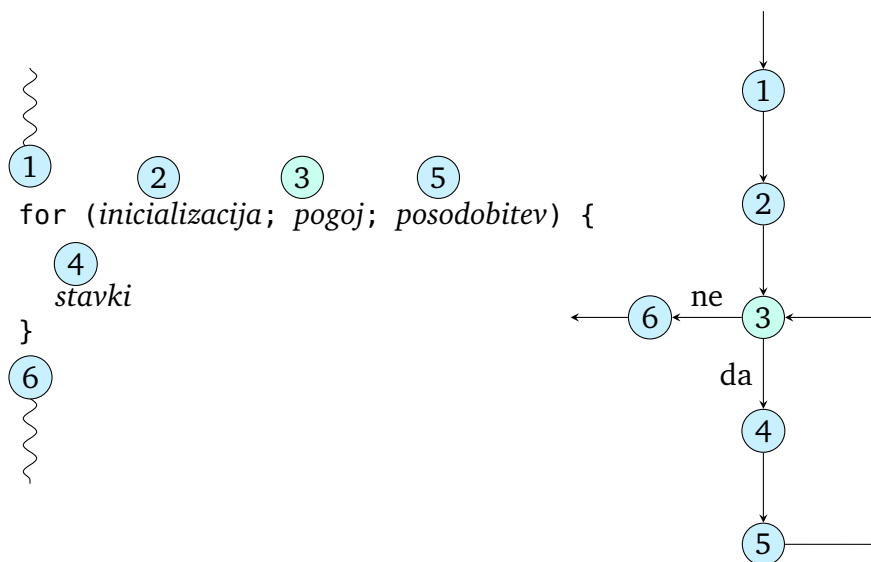
Ker so tovrstne zanke tako pogoste, jih lahko zapisujemo jedrnateje — z zanko *for*:

```
for (inicializacija; pogoj; posodobitev) {
    stavki
}
```

Na primer, kodo 3.18 bi lahko prepisali takole:

```
// koda 3.19 (krmilniKonstrukti/PrimerFor.java)
for (int i = 1; i <= 3; i++) {
    System.out.println("a");
}
System.out.println("b");
```

Slika 3.5 prikazuje diagram poteka za zanko *for*, v tabeli 3.5 pa je predstavljeno izvajanje kode 3.19.



**Slika 3.5** Izvajanje zanke *for*.



Tabela 3.5 Izvajanje kode 3.19.

Stavek/pogoj	Vrednost i
int i = 1	1
i <= 3 (true)	
System.out.println("a")	
i++	2
i <= 3 (true)	
System.out.println("a")	
i++	3
i <= 3 (true)	
System.out.println("a")	
i++	4
i <= 3 (false)	
System.out.println("b")	

Kot smo videli, je zanka *for* samo posebna oblika zanke *while*. Zanko *for* zlahka prepíšemo v zanko *while*, velja pa tudi obratno, saj so lahko komponente v glavi zanke *for* prazne (prazna komponenta *pogoj* je enakovredna vedno resničnemu pogoju *true*). Kljub enakovrednosti pa je zanko *for* smiselno uporabiti takrat, ko je izstopni pogoj vezan na vrednost števca (ki se v zanki povečuje ali zmanjšuje), zanko *while* (ali *do*) pa za druge tipe pogojev. Pri zanki *for* je število obhodov praviloma znano vnaprej, čeprav je lahko odvisno od vrednosti spremenljivk. Zanko *for* imamo raje tudi v primerih, ko moramo določeno opravilo izvesti za vsak element iz neke skupine (npr. za vsako število na določenem intervalu). Z upoštevanjem teh napotkov bi kodo 3.6, 3.8 in 3.10 torej raje zapisali z zanko *for*, pri kodi 3.9, 3.11 in 3.15 pa bi obdržali zanko *while* oz. *do*.

V zvezi z zanko *for* moramo omeniti še naslednjo anomalijo. Če v prvi komponenti glave zanke (*inicializacija*) deklariramo novo spremenljivko, potem ta spremenljivka obstaja samo do konca telesa zanke *for*:

```
// koda 3.20 (krmilniKonstrukti/AnomalijaFor.java)
public class AnomalijaFor {
    public static void main(String[] args) {
        for (int i = 10; i <= 20; i++) {
            System.out.println(i);
        } // (1)
        System.out.println(i);
    }
}
```

Ta program se ne prevede, saj spremenljivka `i` obstaja samo do zaklepaja v vrstici (1). Spremenljivka `i` dejansko pripada istemu bloku kot telo zanke, čeprav je navidez deklarirana v oklepajočem bloku.

**Naloga 3.26** V kodi 3.6, 3.8 in 3.10 zamenjajte zanko *while* z zanko *for*.

**Naloga 3.27** Napišite program, ki prebere število  $n$  in izpiše vrednost  $n!$  (zmnožek števil od 1 do  $n$ ). Lahko predpostavite, da je  $n \leq 20$ .

### 3.7.1 Sprehod po (angleški) abecedi

**Primer 3.12.** Napišimo program, ki po abecednem vrstnem redu izpiše vse velike črke angleške abecede.

*Rešitev.* V razdelku 2.6.3 smo spoznali, da so znaki v javi predstavljeni s števili (kodami). Zaporedje kod velikih (in tudi malih) črk angleške abecede sledi abecednemu vrstnemu redu: znak 'A' ima kodo 65, znak 'B' ima kodo 66 itd. Nalogo lahko potemtakem rešimo preprosto tako, da se z zanko sprehodimo od kode znaka 'A' do kode znaka 'Z' in spotoma vsak znak izpišemo. Ker gre za sprehod po intervalu, je zanka *for* bolj smiselna od zanke *while*:

```
// koda 3.21 (krmilniKonstrukti/Abeceda.java)
public class Abeceda {
    public static void main(String[] args) {
        for (int koda = 'A'; koda <= 'Z'; koda++) {
            System.out.println((char) koda);
        }
    }
}
```

Izrecna pretvorba tipa pri izpisu je nujna, saj bi se sicer namesto znakov izpisale njihove kode. Če pa spremenljivko v zanki deklariramo kot `char` (ker gre za celoštevilski tip, lahko v njem računamo), se izognemo tudi izrecni pretvorbi:

```
for (char znak = 'A'; znak <= 'Z'; znak++) {
    System.out.println(znak);
}
```

□

**Naloga 3.28** Napišite program, ki v obratnem vrstnem redu izpiše male črke angleške abecede.

### 3.7.2 Iskanje maksimuma

**Primer 3.13.** Napišimo program, ki prebere število  $n$  in  $n$  rezultatov skakalcev v daljino in izpiše zaporedno številko najboljšega skakalca in njegov rezultat. Na primer:

```
Koliko skakalcev tekmuje? 5
Vnesite dolžino skoka za 1. skakalca: 635
Vnesite dolžino skoka za 2. skakalca: 680
Vnesite dolžino skoka za 3. skakalca: 671
Vnesite dolžino skoka za 4. skakalca: 695
Vnesite dolžino skoka za 5. skakalca: 667
Najboljši je 4. skakalec (695).
```

*Rešitev.* Naloge se bomo lotili postopoma. Najprej napišimo program, ki zgolj prebere število  $n$  (v programu bomo uporabili bolj povedno ime `stSkakalcev`) in posamezne dolžine skokov. V vsakem obhodu zanke bomo prebrali po en rezultat. Ker je število obhodov zanke znano vnaprej ( $n$ ), bo zanka *for* najprikladnejša:

```
// koda 3.22 (krmilniKonstrukti/Skakalci.java)
import java.util.Scanner;

public class Skakalci {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Koliko skakalcev tekmuje? ");
        int stSkakalcev = sc.nextInt();

        for (int st = 1; st <= stSkakalcev; st++) {
            System.out.print("Vnesite dolžino skoka za " + st +
                             ". skakalca: ");
            int dolzina = sc.nextInt();
        }
    }
}
```

Kako lahko poiščemo dolžino najdaljšega skoka? Vzdržujemo spremenljivko `najDolzina`, ki hrani *dosedanji rekord*, torej največjo dolžino izmed doslej prebranih. Spremenljivko inicializiramo na 0 (dolžina ne more biti negativna), nato pa vsako prebrano dolžino primerjamo s trenutno vrednostjo spremenljivke `najDolzina`. Če je pravkar prebrana dolžina večja od doslej največje, smo presegli rekord, zato spremenljivko `najDolzina` nastavimo na prebrano dolžino. V nasprotnem primeru pa spremenljivka `najDolzina` še vedno hrani dosedanji rekord, zato nam ni treba storiti ničesar. Ko se zanka zaključi, je v spremenljivki `najDolzina` zapisana največja

dolžina izmed vseh prebranih.

```
System.out.print("Koliko skakalcev tekmuje? ");
int stSkakalcev = sc.nextInt();
int najDolzina = 0;

for (int st = 1; st <= stSkakalcev; st++) {
    System.out.print("Vnesite dolžino skoka za " + st +
                    ". skakalca: ");
    int dolzina = sc.nextInt();
    if (dolzina > najDolzina) {
        najDolzina = dolzina;
    }
}
System.out.println(najDolzina);
```

Preostane nam še izpis zaporedne številke skakalca z najdaljšim skokom. Ta problem lahko rešimo tako, da poleg doslej največje dolžine hranimo še zaporedno številko dosedanjega rekorderja (npr. v spremenljivki `najSt`). Ko presežemo rekord, ne posodobimo samo spremenljivke `najDolzina`, ampak tudi spremenljivko `najSt`: vanjo vpišemo zaporedno številko trenutnega skakalca, ki jo hrani spremenljivka `st`. Na ta način zagotovimo, da `najDolzina` po vsakem branju hrani dosedANJI rekord, `najSt` pa zaporedno številko dosedanjega rekorderja.

```
// koda 3.23
System.out.print("Koliko skakalcev tekmuje? ");
int stSkakalcev = sc.nextInt();
int najDolzina = 0;
int najSt = 1;

for (int st = 1; st <= stSkakalcev; st++) {
    System.out.print("Vnesite dolžino skoka za " + st +
                    ". skakalca: ");
    int dolzina = sc.nextInt();
    if (dolzina > najDolzina) {
        najDolzina = dolzina;
        najSt = st;
    }
}
System.out.println("Najboljši je " + najSt +
                    ". skakalec (" + najDolzina + ").");
```

Tabela 3.6 prikazuje, kako se spreminjajo vrednosti spremenljivk `st`, `dolzina`,

najDolzina in najSt v primeru  $n = 5$  in zaporedja dolžin 635, 680, 671, 695, 667.

**Tabela 3.6** Spreminjanje vrednosti spremenljivk v kodi 3.23.

st	dolzina	najDolzina	najSt
		0	1
1	635	635	1
2	680	680	2
3	671		
4	695	695	4
5	667		

□

**Naloga 3.29** Kako se program obnaša, če si več skakalcev deli prvo mesto?

**Naloga 3.30** Popravite program tako, da bo deloval tudi za negativne vnose (namesto dolžin skokov si predstavljajte temperature zraka v °C).

**Naloga 3.31** Napišite program, ki izpiše zaporedno številko in rezultat drugouvrščenega skakalca. Lahko predpostavite, da je  $n$  vsaj 2 in da obstaja natanko en prvouvrščeni in natanko en drugouvrščeni tekmovalec.

### 3.7.3 Praštevila

Praštevila nimajo samo teoretičnega, ampak tudi praktičen pomen, saj igrajo pomembno vlogo pri kriptografskih metodah z javnimi ključi (Cormen in sod., 2009). Problem iskanja praštevil pa je zanimiv tudi s čisto programerskega vidika, zato se bomo k njemu še vračali.

**Primer 3.14.** Napišimo program, ki prebere število  $n$  in izpiše vsa praštevila od 2 do vključno  $n$ .

*Rešitev.* Število je praštevilo, če ima natanko dva delitelja. Nalogo lahko zato rešimo preprosto tako, da za vsako število od 2 do  $n$  preštejemo njegove delitelje, in če je rezultat enak 2, smo našli praštevilo.<sup>4</sup>

```
// koda 3.24 (krmilniKonstrukti/Praštevila1.java)
import java.util.Scanner;
```

<sup>4</sup>Da bo program enostavnejše (samodejno) preverjati, smo izpustili pozivni izpis pred branjem zgornje meje. Tako bomo pogosto ravnali tudi v prihodnje.

```

public class Prastevila1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        for (int kandidat = 2; kandidat <= n; kandidat++) {
            izračunaj število deliteljev kandidata

            if (stDeliteljev == 2) {
                System.out.println(kandidat);
            }
        }
    }
}

```

Kako izračunamo število deliteljev podanega števila (recimo mu  $k$ )? Števec deliteljev inicializiramo na 0, nato pa število  $k$  zaporedoma delimo s števili 1, 2, ...,  $k$  in povečamo števec vsakokrat, ko se deljenje izide.

```

for (int kandidat = 2; kandidat <= n; kandidat++) {
    // izračunaj število deliteljev kandidata
    int stDeliteljev = 0;
    for (int d = 1; d <= kandidat; d++) {
        if (kandidat % d == 0) {
            stDeliteljev++;
        }
    }

    if (stDeliteljev == 2) {
        System.out.println(kandidat);
    }
}

```

Prikazana rešitev sicer deluje, s stališča hitrosti pa ni kaj prida. Kot bomo videli v razdelkih 3.9.3 in 5.6.3, imamo še veliko rezerve. □

**Naloga 3.32** Napišite program, ki prebere število  $n$  in izpiše vsa števila od 1 do  $n$ , ki imajo več deliteljev od vseh njihovih predhodnic. Prvih deset členov tega zaporedja je 1, 2, 4, 6, 12, 24, 36, 48, 60 in 120. Na primer, število 24 je na seznamu zato, ker ima več deliteljev kot vsako od števil z intervala  $[1, 23]$ .

### 3.7.4 Tabela zmnožkov brez poravnave

**Primer 3.15.** Napišimo program, ki prebere število  $n$  in izpiše tabelo zmnožkov števil od 1 do  $n$ . Na primer, pri  $n = 6$  pričakujemo tak izpis:

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36

*Rešitev.* Ker se v terminalu ne moremo premikati v levo ali navzgor, ne gre drugače, kot da tabelo zmnožkov izpišemo po vrsticah. V  $i$ -tem obhodu zanke bomo izpisali  $i$ -to vrstico.

```
// koda 3.25 (krmilniKonstrukti/TabelaZmnozkovBrezPoravnave.java)
import java.util.Scanner;

public class TabelaZmnozkovBrezPoravnave {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        for (int i = 1; i <= n; i++) {
            izpiši i-to vrstico zmnožkov
        }
    }
}
```

Kako izpišemo  $i$ -to vrstico zmnožkov? Pri  $i = 1$  izpišemo rezultate zmnožkov  $1 \cdot 1$ ,  $1 \cdot 2$ , ...,  $1 \cdot n$ , pri  $i = 2$  imamo zmnožke  $2 \cdot 1$ ,  $2 \cdot 2$ , ...,  $2 \cdot n$  itd. Pri splošni vrednosti  $i$  potemtakem po vrsti izpišemo vrednosti zmnožkov  $i \cdot 1$ ,  $i \cdot 2$ , ...,  $i \cdot n$ . Vidimo, da je prvi faktor ( $i$ ) fiksni, medtem ko drugi potuje od 1 do  $n$ . To pomeni, da se z ločenim števcem (npr.  $j$ ) sprehodimo od 1 do  $n$  in vsakokrat izračunamo zmnožek  $i \cdot j$ , na koncu pa skočimo v naslednjo vrstico:

```
for (int i = 1; i <= n; i++) {
    // izpiši i-to vrstico zmnožkov
    for (int j = 1; j <= n; j++) {
        int zmnozek = i * j;
        System.out.print(zmnozek + " ");
    }
}
```

```
System.out.println();
}
```

Če kot vhod podamo  $n = 6$ , dobimo tak izpis:

```
1 2 3 4 5 6
2 4 6 8 10 12
3 6 9 12 15 18
4 8 12 16 20 24
5 10 15 20 25 30
6 12 18 24 30 36
```

Izpis je sicer matematično pravilen, zaradi nepravilne oblike pa se ne ujema s pričakovanim izhodom. Števila bi namreč morala biti desno poravnana. Tega v tem trenutku še ne znamo, bomo pa kmalu. □

### 3.8 Stavek `System.out.printf(...)`

Stavka `System.out.print(...)` in `System.out.println(...)` postaneta okorna in nepregledna, kadar mešamo dobesečne izpise in vrednosti spremenljivk. V takih primerih je bolje uporabiti stavek `System.out.printf(...)`. Metoda `printf` ponuja veliko možnosti izpisa, zato se bomo omejili le na najpogostejše.

Stavek `System.out.printf(...)` uporabljamo takole:

```
System.out.printf(niz, izraz1, izraz2, ...);
```

Stavek izpiše *niz*, pri tem pa njegov *i*-ti podniz oblike `%*` (znak za odstotek, ki mu v skladu z določenimi pravili sledi še eden ali več znakov) zamenja z vrednostjo izraza *izraz<sub>i</sub>*. Izjema sta podniza `%n` in `%%`, ki imata poseben pomen: podniz `%n` se nadomesti s prelomom vrstice, podniz `%%` pa z znakom `%`. Ostali tovrstni podnizi (rekli jim bomo *oblikovna določila*) podajajo tip izraza, po želji pa tudi nekatera dodatna navodila za oblikovanje izpisa.

V tabeli 3.7 so navedena oblikovna določila za različne podatkovne tipe. Na primer, sledeči stavek na pregleden način izpiše seštevek števil *a* in *b* (npr.  $3 + 5 = 8$ ):

```
System.out.printf("%d + %d = %d%n", a, b, a + b);
```

V gornjem klicu metode `printf` se prvo določilo `%d` nadomesti z vrednostjo izraza *a*, drugo z vrednostjo izraza *b*, tretje pa z vrednostjo izraza *a + b*. Vsi trije izrazi imajo celoštevilsko vrednost, zato smo vsakokrat podali določilo `%d`. Določilo `%n` povzroči skok v naslednjo vrstico.

Sledeča koda ilustrira uporabo vseh navedenih določil:

```
// koda 3.26 (krmilniKonstrukti/PrimerPrintf1.java)
```



Tabela 3.7 Oblikovna določila pri metodi printf.

Določilo	Pomen
%d	celo število (byte, short, int, long)
%f	realno število (float, double)
%c	znak (char)
%s	niz (String)
%b	logična vrednost (boolean)

```
int letnik = 4;
double povprecje = 3.9;
char crka = 'C';
String beseda = "java";
System.out.printf("%d. letnik sem končal s povprečjem %f. " +
    "Programiram v jezikih %s in %c. " +
    "Vrednost izraza 3 < 4 je %b.%n",
    letnik, povprecje, beseda, crka, 3 < 4);
```

Oblikovna določila lahko poleg tipa podajajo tudi navodila za oblikovanje izpisa. Določilo `%wx`, kjer je  $w$  pozitivno število,  $x$  pa oznaka tipa (torej d, f, c, s ali b), pomeni, da bo celoten izpis zasedel (najmanj)  $w$  mest. Če je izpis vrednosti pripadajočega izraza sestavljen iz  $k$  znakov, se bo pred vrednostjo izraza izpisalo  $\max(0, w - k)$  presledkov. Na ta način lahko ustvarimo desno poravnan stolpec. Določilo `%-wx` izpiše  $\max(0, w - k)$  presledkov za vrednostjo izraza, zato nam omogoča izpis levo poravnane stolpca. Pri izpisu celih števil lahko uporabimo tudi določilo `%0wd`, ki pred pripadajočim številom izpiše  $\max(0, w - k)$  ničel, pri izpisu realnih pa določili `%.mf` in `%w.mf`; obe izpišeta število na  $m$  decimal, drugo določilo pa pred številom izpiše še  $\max(0, w - k)$  presledkov.

Oglejmo si še en primer (znak `_` predstavlja presledek):

```
// koda 3.27 (krmilniKonstrukti/PrimerPrintf2.java)
int a = 42;
double b = 3.1415;
String niz = "java";
System.out.printf("[%5d]%n", a); // [__42]
System.out.printf("[%10s]%n", niz); // [____java]
System.out.printf("[%5d]%n", a); // [42__]
System.out.printf("[%05d]%n", a); // [00042]
System.out.printf("[%3f]%n", b); // [3.142]
System.out.printf("[%10.3f]%n", b); // [____3.142]
```

### 3.8.1 Tabela zmnožkov s poravnavo

S pomočjo stavka `System.out.printf(...)` lahko tabelo zmnožkov (razdelek 3.7.4) oblikujemo, kot se šika. Vsako število v tabeli bo zasedlo štiri mesta, poravnano pa bo v desno.

```
// koda 3.28 (krmilniKonstrukti/TabelaZmnozkovPrintf.java)
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        int zmnozek = i * j;
        System.out.printf("%4d", zmnozek);
    }
    System.out.println();
}
```

**Naloga 3.33** Popravite program za izpis tabele zmnožkov tako, da bo število mest poravnave pri poljubnem  $n \in [1, 10^4]$  za 1 večje od največjega števila v tabeli.

### 3.8.2 Izpis parov

**Primer 3.16.** Na teniškem turnirju sodeluje  $n$  tekmovalcev, označenih s številkami od 1 do  $n$ . Vsak bo z vsakim odigral natanko eno igro. Napišimo program, ki prebere število  $n$  in po zgledu sledečega primera (pri katerem velja  $n = 6$ ) izpiše vse igralne pare:

```
1:2 1:3 1:4 1:5 1:6
2:3 2:4 2:5 2:6
3:4 3:5 3:6
4:5 4:6
5:6
```

*Rešitev.* Podobno kot tabelo zmnožkov bomo tudi tabelo parov izpisali po vrsticah. Vidimo, da moramo v prvi vrstici izpisati vse pare oblike  $1 : \_$ , v drugi vse pare oblike  $2 : \_$  itd. V  $i$ -ti vrstici (za  $i \in \{1, \dots, n-1\}$ ) torej izpišemo vse pare oblike  $i : \_$ .

```
// koda 3.29 (krmilniKonstrukti/Pari.java)
import java.util.Scanner;

public class Pari {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
    }
}
```

```

        for (int i = 1; i < n; i++) {
            izpiši vrstico parov oblike i:_
        }
    }
}

```

Sedaj se osredotočimo na izpis  $i$ -te vrstice. Ko je  $i = 1$ , izpišemo pare  $1 : 2, 1 : 3, \dots, 1 : n$ . Pri  $i = 2$  izpišemo pare  $2 : 3, 2 : 4, \dots, 2 : n$ . Pri splošnem  $i$  potemtakem izpišemo pare  $i : i + 1, i : i + 2, \dots, i : n$ . To pomeni, da se bomo za vsak  $i$  z ločeno zanko (njen števec naj bo, denimo,  $j$ ) sprehodili od  $i + 1$  do  $n$  in za vsako vrednost  $j$  izpisali par  $i : j$ .

```

for (int i = 1; i < n; i++) {
    // izpiši vrstico parov oblike i:_
    for (int j = i + 1; j <= n; j++) {
        System.out.printf("%d:%d ", i, j);
    }
    System.out.println();
}

```

□

**Naloga 3.34** Napišite program, ki prebere število  $n$  in izpiše vse trojice  $(a, b, c)$  z lastnostjo  $1 \leq a < b < c \leq n$ .

**Naloga 3.35** Napišite program, ki prebere števila  $n, p \in [0, n - 1]$  in  $q \in [0, n - 1]$  in nariše šahovnico velikosti  $n \times n$ , na kateri so prikazana vsa polja, ki jih napada dama na polju s koordinatama  $(p, q)$ . Napadena polja naj bodo označena z znakom N, nenapadena pa z znakom o (bela) oziroma x (črna). Polje, na katerem stoji dama, naj bo označeno s črko D. Polje  $(0, 0)$  se nahaja v zgornjem levem kotu šahovnice in je vedno belo. Prva koordinata narašča navzdol, druga pa v desno. Na primer, za  $n = 7, p = 2$  in  $q = 3$  naj program proizvede tak izpis:

```

oNoNoNo
xoNNNox
NNNDNNN
xoNNNox
oNoNoNo
NoxNxoN
oxoNoxo

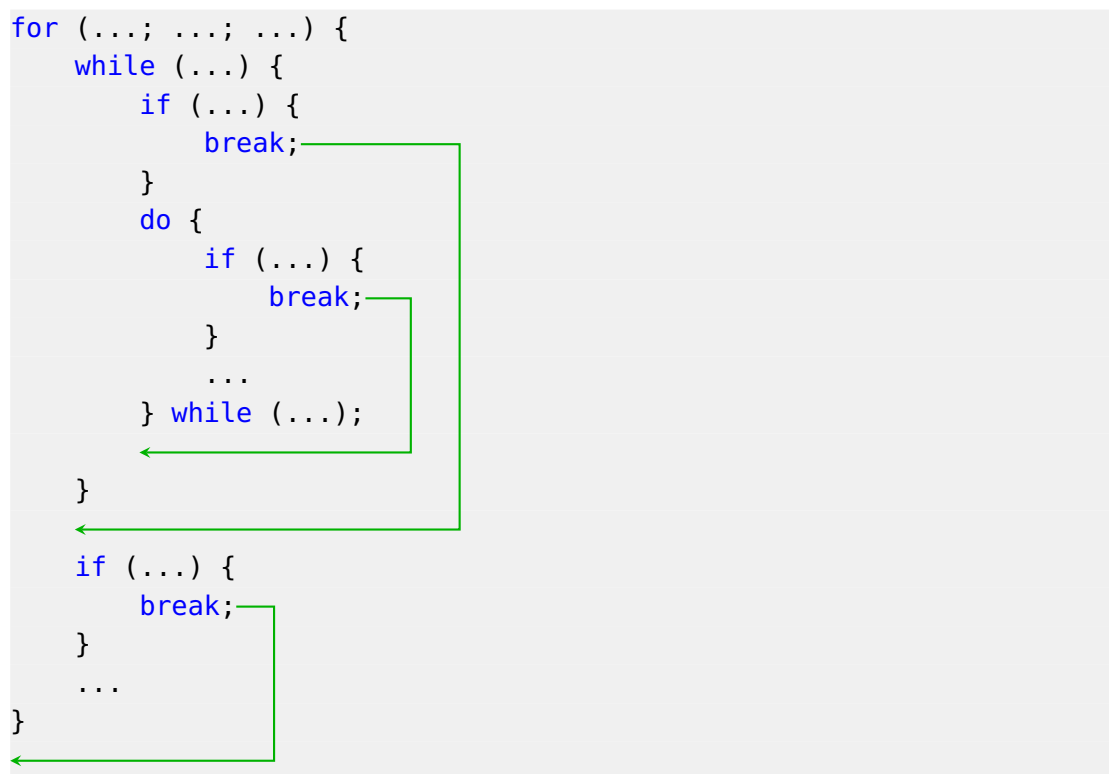
```

### 3.9 Stavka `break` in `continue`

Stavka `break` in `continue` omogočata predčasni zaključek zanke oziroma trenutnega obhoda zanke. Stavka sta sicer v nekaterih primerih dobrodošla, vendar pa z njima ne gre pretiravati, saj lahko negativno vplivata na preglednost programov.

#### 3.9.1 Stavek `break`

Stavek `break` lahko uporabljamo izključno znotraj zanke ali (kot bomo videli kasneje) stavka `switch`. Stavek takoj prekine izvajanje zanke, v kateri se nahaja, in skoči na mesto, ki neposredno sledi zanki. Delovanje stavka `break` prikazuje slika 3.6.



Slika 3.6 Delovanje stavka `break`.

**Primer 3.17.** Napišimo program, ki prebere število in izpiše prvih pet njegovih deliteljev (oziroma vse, če jih je manj kot toliko).

*Rešitev.* S spremenljivko (naj bo  $d$ ) se sprehodimo od 1 do  $n$  in vsakič preverimo, ali je število  $n$  deljivo s številom  $d$ . Če je, izpišemo število  $d$  in povečamo števec deliteljev. Če števec doseže vrednost 5, zanko prekinemo.

```
// koda 3.30 (krmilniKonstrukti/PrvihPetDeliteljev.java)
import java.util.Scanner;

public class PrvihPetDeliteljev {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        int stevec = 0;
        for (int d = 1; d <= n; d++) {
            if (n % d == 0) {
                System.out.println(d);
                stevec++;
                if (stevec == 5) {
                    break;
                }
            }
        }
        // stavek break skoči na to mesto
    }
}
```

Ko števec doseže vrednost 5, se izvede stavek break. Ta skoči na označeno mesto, nato pa se program zaključi. □

Stavek break prekine samo zanko, v kateri se *neposredno* nahaja. Na primer, v sledeči kodi prekine samo notranjo zanko:

```
// koda 3.31 (krmilniKonstrukti/PrimerBreak1.java)
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.printf("(%d, %d)%n", i, j);
        if (i == 2 && j == 1) {
            break;
        }
    }
}
// stavek break skoči na to mesto
}
```

Po izpisu parov (1, 1), (1, 2), (1, 3) in (2, 1) se notranja zanka prekine, nato pa se izvede stavek `i++` v glavi zunanje zanke. Notranja zanka se ponovno izvede (tokrat brez prekinitev). Izpišejo se še pari (3, 1), (3, 2) in (3, 3).

Če želimo prekiniti zanko, ki obdaja zanko, v kateri se nahaja stavek `break`, potem pred njeno glavo zapišemo poljubno oznako in to oznako podamo kot »argument« stavka `break`. V sledečem primeru stavek `break` prekine zunanjo zanko, zato bo par (2, 1) zadnji izpisani par.

```
// koda 3.32 (krmilniKonstrukti/PrimerBreak2.java)
zunanja:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.printf("(%d, %d)%n", i, j);
        if (i == 2 && j == 1) {
            break zunanja;
        }
    }
}
// stavek break skoči na to mesto
```

Stavku `break` se vedno lahko izognemo. Lahko si pomagamo s pogojnimi stavki in logičnimi spremenljivkami. Sledeča koda je enakovredna kodi 3.32:

```
// koda 3.33 (krmilniKonstrukti/PrimerBreak3.java)
boolean nadaljuj = true;
for (int i = 1; i <= 3 && nadaljuj; i++) {
    for (int j = 1; j <= 3 && nadaljuj; j++) {
        System.out.printf("(%d, %d)%n", i, j);
        if (i == 2 && j == 1) {
            nadaljuj = false;
        }
    }
}
}
```

Stavek `break` naj bi se uporabljal kvečjemu takrat, ko ima zanka poleg glavnega še kak dodaten (stranski) izstopni pogoj. V naslednjem primeru uporaba stavka `break` ni primerna in zgolj zamegli (in podaljša) kodo:

```
while (true) {
    System.out.println(n % 10);
    n /= 10;
    if (n == 0) {
        break;
    }
}
```

```
    }
}
```

Sledeča koda je krajša in tudi preglednejša, saj je izstopni pogoj precej bolje viden:

```
while (n > 0) {
    System.out.println(n % 10);
    n /= 10;
}
```

**Naloga 3.36** Napišite program, ki je enakovreden programu `PrvihPetDeliteljev` (koda 3.30), a ne uporablja stavka `break`.

### 3.9.2 Stavek continue

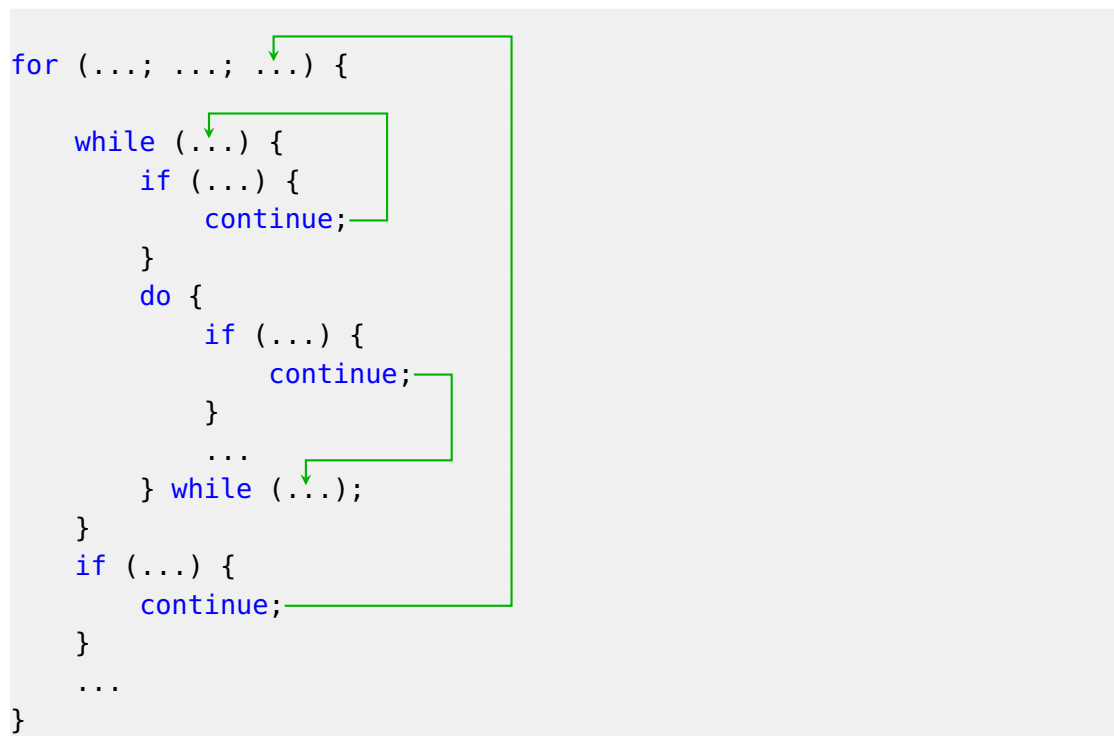
Stavek `continue` lahko nastopa zgolj znotraj zanke. Stavek preskoči vse stavke do konca telesa zanke, v kateri se nahaja, in nadaljuje s preverjanjem znančnega pogoja pri zankah *while* in *do* oziroma z izvedbo posodobitvenega stavka (tretje komponente glave) pri zanki *for*. Delovanje stavka `continue` prikazuje slika 3.7.

Tudi stavek `continue` deluje zgolj nad zanko, v kateri se *neposredno* nahaja. Na primer, v sledeči kodi sploh nima učinka, saj skoči na stavek `j++` v glavi notranje zanke, to pa bi se tedaj tako ali tako zgodilo:

```
// koda 3.34 (krmilniKonstrukti/PrimerContinue1.java)
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.printf("(%d, %d)%n", i, j);
        if (i == 2 && j == 1) {
            continue;
        }
    }
}
```

Če želimo skočiti na preverjanje pogoja oz. posodobitveni stavek pri zunanji zanki, pred njeno glavo postavimo oznako in jo podamo kot »argument« stavka `continue`:

```
// koda 3.35 (krmilniKonstrukti/PrimerContinue2.java)
zunanja:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.printf("(%d, %d)%n", i, j);
        if (i == 2 && j == 1) {
            continue zunanja;
        }
    }
}
```



Slika 3.7 Delovanje stavka continue.

```

    }
}
}

```

Stavek `continue` skoči na stavek `i++` v glavi zunanje zanke, kar pomeni, da bo paru (2, 1) sledil par (3, 1).

Tako kot stavek `break` lahko tudi stavek `continue` vedno nadomestimo s pogojnimi stavki in logičnimi spremenljivkami. Tudi razlogi za uporabo so podobni kot pri stavku `break`: stavek `continue` je smiseln le takrat, ko bi bila koda brez njega (znatno) daljša ali manj pregledna.

**Naloga 3.37** Kodo 3.35 prepišite tako, da ne bo uporabljala niti stavka `continue` niti stavka `break`.

### 3.9.3 Praštevila učinkoviteje

V razdelku 3.7.3 smo napisali prvo različico programa, ki izpiše vsa praštevila na intervalu  $[2, n]$ . Program za vsako število z intervala  $[2, n]$  prešteje njegove delitelje. Če ima natanko dva, je praštevilo:



```
// krmilniKonstrukti/Prastevila1.java
for (int kandidat = 2; kandidat <= n; kandidat++) {
    int stDeliteljev = 0;
    for (int d = 1; d <= kandidat; d++) {
        if (kandidat % d == 0) {
            stDeliteljev++;
        }
    }
    if (stDeliteljev == 2) {
        System.out.println(kandidat);
    }
}
```

Program deluje povsem pravilno, žal pa ni najbolj učinkovit. Kako bi ga lahko izboljšali?

Za začetek lahko ugotovimo, da nam ni treba prešteti vseh deliteljev tekočega števila. Če je število  $k$  (kandidat) deljivo z vsaj enim številom z intervala  $[2, k - 1]$ , lahko zaključimo, da  $k$  ni praštevilo. Program popravimo tako, da števec deliteljev nadomestimo z logično spremenljivko prastevilo. Spremenljivko na začetku postavimo na `true`, takoj ko se eno od deljenj izide, pa jo nastavimo na `false` in prekinemo notranjo zanko. Po koncu izvajanja notranje zanke preverimo njeno vrednost: če je ostala `true`, vemo, da se nobeno deljenje ni izšlo, zato trenutnega kandidata proglasimo za praštevilo.

```
// koda 3.36 (krmilniKonstrukti/Prastevila2.java)
for (int kandidat = 2; kandidat <= n; kandidat++) {
    boolean prastevilo = true;
    for (int d = 2; d < kandidat; d++) {
        if (kandidat % d == 0) {
            prastevilo = false;
            break;
        }
    }
    if (prastevilo) {
        System.out.println(kandidat);
    }
}
```

Program je mogoče še izboljšati. Vsa praštevila razen dvojke so liha, zato lahko število 2 izpišemo posebej, v zunanji zanki pa se sprehodimo le po lihih številih. Podobno lahko storimo v notranji zanki: nima smisla, da preverjamo, ali je liho število deljivo s sodim.

```
// koda 3.37 (krmilniKonstrukti/Prastevila3.java)
System.out.println(2);
for (int kandidat = 3; kandidat <= n; kandidat += 2) {
    boolean prastevilo = true;
    for (int d = 3; d < kandidat; d += 2) {
        if (kandidat % d == 0) {
            prastevilo = false;
            break;
        }
    }
    if (prastevilo) {
        System.out.println(kandidat);
    }
}
```

Zadnja izboljšava v tem razdelku se nanaša na zgornjo mejo v notranji zanki. Izkaže se, da je dovolj, da za število  $k$  preverimo deljivost s števili do  $\sqrt{k}$ . Razlog je enostaven: če število  $k$  nima nobenega delitelja na intervalu  $[2, \sqrt{k}]$ , ga na intervalu  $(\sqrt{k}, k-1]$  prav tako ne bo imelo (če je  $d \in (\sqrt{k}, k-1]$  delitelj števila  $k$ , je njegov delitelj tudi  $k/d \in [2, \sqrt{k}]$ ).

```
// koda 3.38 (krmilniKonstrukti/Prastevila4.java)
System.out.println(2);
for (int kandidat = 3; kandidat <= n; kandidat += 2) {
    boolean prastevilo = true;
    int meja = (int) Math.round(Math.sqrt(kandidat));
    for (int d = 3; d <= meja; d += 2) {
        if (kandidat % d == 0) {
            prastevilo = false;
            break;
        }
    }
    if (prastevilo) {
        System.out.println(kandidat);
    }
}
```

Metoda `Math.sqrt` izračuna kvadratni koren podanega števila, metoda `Math.round` pa dobljeno število zaokroži na najbližje celo število tipa `long`. To število nato pretvorimo v tip `int`.

Intuitivno nam je jasno, da je vsaka različica programa za iskanje praštevil hitrejša od prejšnje, saj pri isti vrednosti  $n$  opravi manj dela, a kako bi to izmerili? Preprosto! Najprej zakomentirajmo vse stavke `System.out.println(...)`, saj so

razmeroma počasni, poleg tega pa nam ne bodo prav nič povedali o hitrosti iskanja praštevil. Sedaj lahko čas izvajanja programa izmerimo tako, da njegov osrednji del obdamo s klicema metode `System.nanoTime`, ki vrne čas (v nanosekundah), ki je pretekel od določenega fiksnega trenutka. Razlika med zaporednima klicema te metode nam torej pove, koliko časa je preteklo med njima.

```
long zacetniCas = System.nanoTime();
for (int kandidat = ...; ...; ...) {
    ...
}
long razlika = System.nanoTime() - zacetniCas;
System.out.printf("Trajanje: %.2f milisekund%n", razlika / 1e6);
```

Zdaj lahko program poženemo, denimo, takole ...

```
terminal> echo 10000 | java Prastevila1
```

... in izpiše se trajanje programa za  $n = 10^4$ . Da dobimo realnejšo sliko, poženemo ukaz večkrat. Avtor je programe pognal za  $n \in \{10^4, 10^5\}$  in rezultate zbral v tabeli 3.8.

**Tabela 3.8** Časi (v milisekundah) izvajanja programov za iskanje praštevil na avtorjevem računalniku.

Program	$n = 10^4$	$n = 10^5$
Prastevila1	160	15 500
Prastevila2	26	1400
Prastevila3	15	700
Prastevila4	1,8	11

Program za izpis praštevil smo z razmeroma preprostimi triki bistveno pohitrili. Treba pa je opozoriti, da hitrost izvajanja ne sme biti naš prvi cilj. Najprej mora program pravilno delovati, šele zatem se lahko lotimo pohitritev. Včasih pa se jim lahko tudi izognemo: če se mora program izvesti enkrat samkrat, ni tako bistveno, ali se izvede v minuti, sekundi ali tisočinki sekunde, se pa zelo pozna, če ga pišemo eno uro ali en dan.

Zgodbe s praštevilami še nismo zaključili. V poglavju 5, ko se bomo seznanili s tabelami, bomo napisali še hitrejšo kodo.

**Naloga 3.38** Napišite program, ki prebere število  $n$  in izpiše število vseh njegovih deliteljev. Poskusite napisati karseda učinkovito kodo. Kako velik  $n$  še lahko obdelate v eni sekundi? Namig: če število  $n$  zapišemo kot  $n = p_1^{s_1} p_2^{s_2} \dots p_k^{s_k}$ , kjer so

$p_1, \dots, p_k$  medsebojno različna praštevila, potem hitro vidimo, da so njegovi delitelji števila oblike  $p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$  za vse možne  $k$ -terke  $(a_1, a_2, \dots, a_k)$  z lastnostmi  $0 \leq a_1 \leq s_1, 0 \leq a_2 \leq s_2, \dots, 0 \leq a_k \leq s_k$ . Število deliteljev števila  $n$  je potemtako enako številu takih  $k$ -terk.

### 3.9.4 Pitagorejska števila

**Primer 3.18.** Za potrebe te naloge naj bo število  $c$  *pitagorejsko*, če ga je mogoče zapisati kot  $c^2 = a^2 + b^2$ , kjer sta  $a$  in  $b$  pozitivni celi števili. Najmanjše tako število je 5, saj velja  $5^2 = 3^2 + 4^2$ . Napišimo program, ki prebere število  $n$  in izpiše število pitagorejskih števil na intervalu  $[1, n]$ . Na primer, pri  $n = 30$  je pravilen rezultat enak 10.

*Rešitev.* Kako se lotimo te naloge? Za vsako število  $c$  z intervala  $[1, n]$  preverimo, ali je pitagorejsko. Če je, povečamo števec.

*// koda 3.39 (krmilniKonstrukti/PitagorejskaStevila.java)*

```
import java.util.Scanner;

public class PitagorejskaStevila {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int meja = sc.nextInt();

        int stPitagorejskih = 0;
        for (int c = 1; c <= meja; c++) {
            if (število c je pitagorejsko) {
                stPitagorejskih++;
            }
        }
        System.out.println(stPitagorejskih);
    }
}
```

Odgovor na vprašanje, ali je število  $c$  pitagorejsko, lahko dobimo tako, da za vsak par števil  $a$  in  $b$  z intervala  $[1, c - 1]$  preverimo, ali velja  $c^2 = a^2 + b^2$ . Ker je seštevanje komutativno, se omejimo samo na pare z lastnostjo  $a < b$ . Kaj pa primer  $a = b$ ? Lahko ga mirno ignoriramo, ker iz  $a = b$  sledi  $c^2 = a^2 + b^2 = 2a^2 = (a\sqrt{2})^2$ . Če je število  $c$  celo, potem število  $a = c\sqrt{2}/2$  gotovo ni.

```
for (int c = 1; c <= meja; c++) {
    // je število c pitagorejsko?
    for (int a = 1; a < c - 1; a++) {
```

```

        for (int b = a + 1; b < c; b++) {
            if (c * c == a * a + b * b) {
                stPitagorejskih++;
            }
        }
    }
}

```

Preverimo, ali pri  $n = 30$  res dobimo 10:

```

terminal> echo 30 | java PitagorejskaStevila
11

```

Nekje smo ga očitno polomili. Ne preostane nam drugega, kot da se lotimo razhroščevanja.

Zgolj s preučevanjem programa bomo težko odkrili napako. Veliko več možnosti nam dajejo vmesni izpisi. Ker se števec `stPitagorejskih` očitno prevečkrat poveča, se nam splača preveriti, zakaj. Števec se poveča vsakokrat, ko odkrijemo trojico, zato jo na tistem mestu izpišimo:

```

for (int c = 1; c <= meja; c++) {
    // je število c pitagorejsko?
    for (int a = 1; a < c - 1; a++) {
        for (int b = a + 1; b < c; b++) {
            if (c * c == a * a + b * b) {
                System.out.printf("%d^2 = %d^2 + %d^2%n", c, a, b);
                stPitagorejskih++;
            }
        }
    }
}

```

Pri  $n = 30$  dobimo tak izpis:

```

5^2 = 3^2 + 4^2
10^2 = 6^2 + 8^2
13^2 = 5^2 + 12^2
15^2 = 9^2 + 12^2
17^2 = 8^2 + 15^2
20^2 = 12^2 + 16^2
25^2 = 7^2 + 24^2
25^2 = 15^2 + 20^2
26^2 = 10^2 + 24^2
29^2 = 20^2 + 21^2

```

```
30^2 = 18^2 + 24^2
11
```

Sedaj je jasno: za nekatera števila  $c$  (npr. 25) obstaja več parov  $(a, b)$  z lastnostjo  $c^2 = a^2 + b^2$ . Program poišče vse take pare, moral pa bi se zadovoljiti z enim samim. Po odkritju prvega para moramo zato prekiniti srednjo zanko:

```
for (int c = 1; c <= meja; c++) {
    // je število c pitagorejsko?
    srednja:
    for (int a = 1; a < c - 1; a++) {
        for (int b = a + 1; b < c; b++) {
            if (c * c == a * a + b * b) {
                //System.out.printf("%d^2 = %d^2 + %d^2%n", a, b, c);
                stPitagorejskih++;
                break srednja;
            }
        }
    }
}
```

Program sedaj deluje v skladu s pričakovanji. V razdelku 4.4.3 bomo videli, da se lahko (ne posebej elegantnega) stavka `break` na preprost način znebimo. □

**Naloga 3.39** Program pošnite za različne vrednosti  $n$  in vsakokrat izmerite čas izvajanja. Kako velik  $n$  lahko program obdela v eni sekundi?

**Naloga 3.40** Če poznamo števili  $c$  in  $a$ , poznamo tudi število  $b$ , saj velja  $b^2 = c^2 - a^2$ . Število  $c$  je torej pitagorejsko, če obstaja tak  $a$ , da bo vrednost  $c^2 - a^2$  popolni kvadrat. Napišite program z upoštevanjem tega dejstva. Kolikšen je sedaj največji  $n$ , ki ga program obdela v eni sekundi?

**Naloga 3.41** Napišite program, ki prebere števili  $a \geq 4$  in  $b \geq a$  ter za vsako sodo število z intervala  $[a, b]$  preveri Goldbachovo domnevo in izpiše ustrezen dokaz. Goldbachova domneva trdi, da je vsako sodo število, večje od 2, mogoče zapisati kot vsoto dveh praštevil.

### 3.10 Stavek `switch`

Stavek `switch` ima takšno obliko:

```
switch (izraz) {
    case konstanta1:
```

```

    stavki1;
    break;

case konstanta2:
    stavki2;
    break;
...

case konstantan:
    stavkin;
    break;

default:
    stavkin+1;
    break;
}

```

Gre za poseben primer verige pogojnih stavkov. Vrednost izraza v glavi stavka `switch` se po vrsti primerja s posameznimi konstantami. Če je vrednost izraza *izraz* enaka *konstanta*<sub>1</sub>, potem se izvršijo *stavki*<sub>1</sub>, vse ostalo pa se preskoči. V nasprotnem primeru se vrednost izraza primerja s konstanto *konstanta*<sub>2</sub>; če sta vrednosti enaki, se izvedejo *stavki*<sub>2</sub> in se vse ostalo preskoči, sicer pa se preveri ujemanje vrednosti izraza s konstanto *konstanta*<sub>3</sub> itd. Stavki *stavki*<sub>n+1</sub> se izvedejo samo v primeru, če je *izraz* različen od vseh konstant. Odsek `default` lahko tudi izpustimo.

Izraz v glavi stavka je lahko tipa `byte`, `short`, `int`, `char`, `String` ali `enum`.<sup>5</sup> Konstante v glavah posameznih odsekov `case` so lahko navedene kot čiste vrednosti oz. *literali* (npr. 42, 'a', "Dober dan" ...), kot poimenovane konstante (razdelek 6.10.1) ali pa kot konstante znotraj objekta tipa `enum`.

**Primer 3.19.** Napišimo program, ki prebere šolsko oceno in izpiše njen pomen (5 — odlično, 4 — prav dobro itd.), če je ocena veljavna, oziroma besedilo Neveljavna ocena!, če ni.

*Rešitev.* Nalogo lahko rešimo z verigo pogojnih stavkov, a rešitev s stavkom `switch` je lepša in učinkovitejša:

```

// koda 3.40 (krmilniKonstrukti/NazivOcene.java)
import java.util.Scanner;

public class NazivOcene {
    public static void main(String[] args) {

```

<sup>5</sup>Naštevnege tipa (`enum`) v tej knjigi ne bomo obravnavali. Povejmo le, da gre za tip, ki nam omogoča definicijo množice poimenovanih konstant.

```

Scanner sc = new Scanner(System.in);
int ocena = sc.nextInt();

switch (ocena) {
    case 5:
        System.out.println("odlično");
        break;

    case 4:
        System.out.println("prav dobro");
        break;

    case 3:
        System.out.println("dobro");
        break;

    case 2:
        System.out.println("zadostno");
        break;

    case 1:
        System.out.println("nezadostno");
        break;

    default:
        System.out.println("Neveljavna ocena!");
        break;
}
}
}

```

□

Kaj v stavku `switch` počnejo stavki `break`? Čas je, da dopolnimo definicijo stavka `break`. Stavek `break` prekine najbolj notranjo zanko *ali stavek switch*, v katerem se nahaja. Če stavek `break` izpustimo, se izvajanje nadaljuje pri naslednjem odseku `case`, in to ne glede na vrednost izraza v glavi stavka `switch`! Za lažje razumevanje sledi odsek programa, v tabeli 3.9 pa je prikazan njegov izpis v odvisnosti od vsebine niza jezik.

```

// koda 3.41 (krmilniKonstrukti/PrimerSwitch1.java)
switch (jezik) {
    case "sl":

```



```

        System.out.println("pet");

    case "en":
        System.out.println("five");
        break;

    case "de":
        System.out.println("fünf");

    case "it":
        System.out.println("cinque");

    case "pl":
        System.out.println("pięć");
}

```

**Tabela 3.9** Delovanje kode 3.41. (Vsaka beseda se izpiše v svoji vrstici.)

jezik	izpisane besede
sl	pet, five
en	five
de	fünf, cinque, pięć
it	cinque, pięć
pl	pięć

Ta funkcionalnost stavka switch je uporabna skoraj izključno v primerih, ko isto zaporedje stavkov velja za več vrednosti izraza. Na primer:

```

// koda 3.42 (krmilniKonstrukti/PrimerSwitch2.java)
switch (drzava) {
    case "Avstrija":
    case "Nemčija":
        jezik = "de";
        break;

    case "Združeno kraljestvo":
    case "Združene države Amerike":
    case "Kanada":
    case "Avstralija":
    case "Nova Zelandija":

```

```

        jezik = "en";
        break;
}

```

Če ima niz država vsebino `Avstrija`, potem stavek `switch` ne izvede samo praznega zaporedja stavkov, ki pripada konstanti `"Avstrija"`, ampak tudi stavke, ki pripadajo konstanti `"Nemčija"`.

Pri odseku `default` oziroma pri zadnjem odseku `case` (če odseka `default` ni) je stavek `break` dejansko odveč, kljub temu pa je priporočljivo, da ga pišemo, saj je potem manj verjetno, da ga bomo pozabili kje drugje. Manjkajoči stavki `break` v stavkih `switch` so vzrok za prenekatero zahrbtno napako!

Stavek `switch` nas lahko preseneti tudi glede dosega spremenljivk. Njegovo celotno telo tvori en sam blok, zato so spremenljivke, deklarirane v nekem odseku `case`, vidne tudi v vseh nadaljnjih odsekih. Na primer, sledeča koda se ne prevede:

```

switch (t) {
    case 1:
        int a = 3;
        ...
        break;

    case 2:
        int a = 4;    // spremenljivka a že obstaja
        ...
        break;
}

```

Čeprav je takšno delovanje na prvi pogled neobičajno, v resnici ni, saj je vsak blok zamejen s parom zavitih oklepajev (edina izjema je, kot smo videli v razdelku 3.7, deklaracija spremenljivke v glavi zanke `for`). Če torej želimo, da so spremenljivke, deklarirane znotraj odseka `case`, vidne samo v tistem odseku, moramo stavke zapreti v blok:

```

switch (t) {
    case 1: {
        int a = 3;
        ...
        break;
    }

    case 2: {
        int a = 4;    // v redu, to je druga spremenljivka a
        ...
    }
}

```

```

        break;
    }
}

```

**Naloga 3.42** Prepišite kodo 3.41 v verigo pogojnih stavkov. Nizov med seboj ne smete primerjati z dvojnimi enačjem (npr. `jezik == "sl"`), ampak zgolj z metodo `equals` (npr. `jezik.equals("sl")`) — rezultat tega izraza je `true`, če je vsebina niza `jezik` enaka `sl`, in `false`, če ni.

### 3.11 Pogojni operator

Precej javinih operatorjev smo že spoznali. Nekaj jih bomo še v nadaljnjih razdelkih in poglavjih, v tem razdelku pa si bomo ogledali *pogojni operator*.

Pogojni operator je edini javin trojiški operator. Uporabimo ga takole:

```
rezultat = pogoj ? izrazT : izrazF;
```

Desna stran gornjega prireditvenega stavka se imenuje *pogojni izraz*. Vrednost pogojnega izraza je vrednost izraza *izrazT*, če je *pogoj* resničen, oziroma vrednost izraza *izrazF*, če *pogoj* ni resničen. Tipa izrazov *izrazT* in *izrazF* morata biti priredljiva tipu spremenljivke *rezultat*. Operande pogojnega operatorja zaradi boljše preglednosti včasih obdamo z oklepaji.

Pogojni operator lahko nadomesti nekatere pogojne stavke. Na primer, namesto

```

if (tocke >= 50) {
    System.out.println("opravil");
} else {
    System.out.println("padel");
}

```

lahko pišemo

```
System.out.println((tocke >= 50) ? ("opravil") : ("padel"));
```

nikakor pa ne gre

```

(tocke >= 50) ? (System.out.println("opravil")) :
               (System.out.println("padel"));

```

saj koda `System.out.println(...)` nima vrednosti in zato ni izraz. Oglejmo si še nekaj primerov rabe pogojnega operatorja:

- Absolutna vrednost števila *a*:

```
int absolutna = (a > 0) ? (a) : (-a);
```

- Manjše izmed števil a in b:

```
int manjse = (a < b) ? (a) : (b);
```

- Najmanjše izmed števil a, b in c:

```
int najmanjse = (a < b) ? (a < c ? a : c) : (b < c ? b : c);
```

**Naloga 3.43** Napišite izraz, katerega vrednost je mediana števil a, b in c.

**Naloga 3.44** Nadomestite celoten stavek switch v programu NazivOcene (koda 3.40) z enim samim stavkom `System.out.println(...)`.

## 3.12 Delo z biti

Z bitnimi operatorji se v tej knjigi ne bomo veliko ukvarjali, kljub temu pa si zaslužijo svoj razdelek. Osnova za razumevanje bitnih operatorjev je *bitna predstavitev* celih števil. *Bit* je preprosto dvojiška številka — ničla ali enica. Vsi podatki v računalniku (ne samo cela števila) so predstavljeni z biti, torej z zaporedji ničel in enic. Digitalna vezja, iz katerih je računalnik sestavljen, namreč rokujejo samo z biti. Ti so v praksi predstavljeni z dvema različnima fizikalnima stanjema (npr. z »nizko« in »visoko« napetostjo ali pa z »majhnim« in »velikim« nabojem).

### 3.12.1 Bitni zapis celih števil

Da bomo bitne operatorje lažje razumeli, si najprej oglejmo, kako v zaporedje bitov pretvorimo celo število. Nenegativno število najprej pretvorimo v dvojiški številski sistem: zaporedoma ga celoštevilsko delimo z 2, dokler ne dobimo 0, in dobljene ostanke pri deljenju prepišemo v obratnem vrstnem redu. Nastali dvojiški zapis za tem le še z vodilnimi ničlami dopolnimo do ciljne dolžine (ta je 8 bitov pri tipu `byte`, 16 pri tipu `short`, 32 pri tipu `int` in 64 pri tipu `long`). Na primer, pri številu 42 gre takole:

$$42 : 2 = 21 \text{ (ostane 0)}$$

$$21 : 2 = 10 \text{ (ostane 1)}$$

$$10 : 2 = 5 \text{ (ostane 0)}$$

$$5 : 2 = 2 \text{ (ostane 1)}$$

$$2 : 2 = 1 \text{ (ostane 0)}$$

$$1 : 2 = 0 \text{ (ostane 1)}$$

Če ostanke prepisemo od spodaj navzgor, dobimo dvojiško število 101010. V tipu byte je število 42 torej zapisano kot 0010 1010, v tipu short kot 0000 0000 0010 1010 itd.

Negativno celo število pa v bite pretvorimo tako, da najprej v bitni zapis za ciljni celoštevilski tip (byte, short, int ali long) pretvorimo njegovo absolutno vrednost, potem enice spremenimo v ničle in obratno, na koncu pa prištejemo 1. (Takšni predstavitvi negativnih števil rečemo *dvojiški komplement*.) Recimo, da je naš ciljni tip byte. Pri številu  $-42$  torej pričnemo z zapisom števila 42 (ta je v tipu byte enak 0010 1010), nato pa ničle spremenimo v enice in obratno (1101 0101) ter prištejemo 1, da pridobimo končni zapis (1101 0110).

Seveda računalnik v dvojiškem zapisu tudi računa. Aritmetične operacije se izvajajo na enak način, kot bi se v desetiškem zapisu. Na primer, če želimo sešteti dve števili, ju desno poravnamo, nato pa seštevamo po bitih od zadaj naprej, pri čemer upoštevamo prenose (npr. po dvojiško je  $1 + 1 = 10$ , kar pomeni, da zapišemo 0, enico pa prištejemo k naslednji vsoti).

**Naloga 3.45** Števili 27 in  $-98$  pretvorite v dvojiški zapis v tipu byte, nato pa ju v tem zapisu seštejte in odštejte in rezultata pretvorite v desetiški zapis.

**Naloga 3.46** Števili  $-175$  in 153 pretvorite v dvojiški zapis v tipu short, nato pa ju v tem zapisu zmnožite in rezultat pretvorite v desetiški zapis.

### 3.12.2 Bitni operatorji

Vsi javanski operatorji, ne samo bitni, tako ali drugače delujejo nad biti, vendar pa se nam pri večini od njih tega ni treba zavedati. Na primer, čeprav se operacija seštevanja v računalniku izvede nad bitnima zapisoma operandov, lahko to dejstvo mirno zanemarimo in se delamo, kot da se števili seštejeta po desetiško. Bitnih operatorjev pa ne moremo razumeti, če ne poznamo bitne predstavitve.

Javanski bitni operatorji so zbrani v tabeli 3.10. Najprej bomo pojasnili operatorje  $\&$ ,  $|$ ,  $\wedge$  in  $\sim$ . Prvi trije so dvojiški, četrti pa je eniški. Tabela 3.11 prikazuje njihovo delovanje nad enobitnimi operandi. V primeru večbitnih operandov se bitna operacija izvede nad vsakim parom istoležnih bitov posebej. Na primer, vrednost izraza  $42 \& 27$  znaša 10:

$$\begin{array}{rcl} 42 & \rightarrow & 0\dots0\ 00101010 \\ 27 & \rightarrow & 0\dots0\ 00011011 \\ \hline & & 0\dots0\ 00001010 \rightarrow 10 \end{array}$$

Kako pa delujejo bitni pomiki? Izraz  $a \ll k$  se izračuna tako, da se biti števila  $a$  pomaknejo za  $k$  mest v levo, pri čemer se izpraznjena mesta na desni napolnijo z ničlami. Na primer, vrednost izraza  $42 \ll 2$  je enaka 168:

Tabela 3.10 Bitni operatorji.

Operator	Pomen
&	Bitni in
	Bitni ali
^	Bitni izključujoči ali
~	Bitni ne
<<	Bitni pomik v levo
>>	Bitni pomik v desno z razmnoževanjem skrajno levega bita
>>>	Bitni pomik v desno z razmnoževanjem ničle

Tabela 3.11 Operatorji &amp;, |, ^ in ~ nad enobitnimi operandi

p	q	p & q	p   q	p ^ q	~p
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	

$$\begin{array}{rcl}
 42 & \rightarrow & 0\dots0\,00101010 \\
 & & \hline
 & & 0\dots0\,10101000 \rightarrow 168
 \end{array}$$

Ni težko ugotoviti, da pomik v levo za  $k$  mest pomeni množenje števila z  $2^k$ .

Izraza  $a \gg k$  in  $a \ggg k$  pomakneta bitni zapis za  $k$  mest v desno, le da se v prvem primeru na izpraznjena mesta na levi vstavljajo kopije skrajno levega bita števila  $a$ , v drugem pa ničle. Pri pozitivnem številu  $a$  sta oba izraza enakovredna deljenju števila  $a$  z  $2^k$ .

Bitne operatorje lahko kombiniramo s prireditvenim operatorjem. Na primer, stavek  $a \&= b$  je okrajšava za  $a = a \& b$ , stavek  $a <<= k$  je krajši zapis stavka  $a = a << k$  itd.

Bitni operatorji so izjemno hitri, saj se izvajajo neposredno z digitalnimi vezji, brez kakršnihkoli vmesnih pretvorb. Predstavitev podatkov z biti je tudi prostorsko izjemno ekonomična. Bitne operatorje bomo zato pogosto našli v časovno ali prostorsko kritičnih aplikacijah. Kot zanimivost povejmo, da jih bomo veliko našli v programih za igranje šaha. Šahovnica ima 64 polj, kar je ravno dolžina tipa `long`. V eno samo število tipa `long` lahko torej »stlačimo« 64 enic in ničel. Te lahko predstavljajo položaje posameznih figur, njihove možne premike na prazni šahovnici itd. Operacije nad tako zapisanimi podatki (npr. izračun možnih ciljnih polj figure na

trenutni šahovnici) izrazimo z bitnimi operatorji.

**Naloga 3.47** Napišite izraz, katerega vrednost je enaka vrednosti skrajno desnega bita v številu  $n$ .

**Naloga 3.48** Napišite izraz, katerega vrednost je enaka vrednosti bita na položaju  $k$  v številu  $n$  (skrajno desni bit ima položaj 0, njegov levi sosed ima položaj 1 itd.).

**Naloga 3.49** Napišite stavek, ki v številu  $n$  nastavi bit na položaju  $k$  na vrednost  $b$ .

**Naloga 3.50** Napišite program, ki prebere število  $n$  in izpiše položaje vseh njegovih enic.

**Naloga 3.51** Napišite izraz, katerega vrednost je največja potenca števila 2, ki deli število  $n > 0$ . Na primer, pri  $n = 24$  je odgovor enak 8, pri  $n = 25$  pa 1.

### 3.13 Pregled operatorjev

Tabela 3.12 navaja vse javanske operatorje, urejene po padajoči prednosti. Operator `instanceof` bomo spoznali v poglavju 7, ostale pa smo že srečali. Operatorji na višjem prednostnem nivoju (tj. na nivoju z manjšo številko) vežejo močnejše od tistih na nižjem nivoju. Znotraj istega nivoja vsi operatorji vežejo enako močno. Na primer, izraz

```
a + -b * c < d << e && f == g
```

se izračuna takole:

```
((a + ((-b) * c)) < (d << e)) && (f == g)
```

Bodimo pozorni na to, da imajo primerjalni operatorji prednost pred operatorji `&`, `^` in `|`.

*Asociativnost* operatorjev nam pove, kako se izračuna izraz  $a_1 op_1 a_2 op_2 a_3 \dots a_{n-1} op_{n-1} a_n$ , kjer so  $op_1, op_2, \dots, op_{n-1}$  (dvojiški) operatorji na istem prednostnem nivoju. Če so operatorji *levoasociativni*, se izraz izračuna kot  $((\dots((a_1 op_1 a_2) op_2 a_3) \dots a_{n-1}) op_{n-1} a_n)$ , če so *desnoasociativni*, pa kot  $a_1 op_1 (a_2 op_2 (a_3 \dots (a_{n-1} op_{n-1} a_n) \dots))$ . Na primer, izraz

```
a * b / c % d * e
```

se izračuna kot

```
((a * b) / c) % d * e
```

**Tabela 3.12** Operatorji v javi.

Prednostni nivo	Operatorji	Asociativnost
1 (najvišji)	postfiksni ++ in --	leva
2	prefiksni ++ in --, eniški + in -, ~, !	desna
3	*, /, %	leva
4	+, -	leva
5	<<, >>, >>>	leva
6	<, >, <=, >=, instanceof	leva
7	==, !=	leva
8	&	leva
9	^	leva
10		leva
11	&&	leva
12		leva
13	pogojni operator (?:)	desna
14	=, +=, -=, *=, /=, %=, &=, ^=,  =, <<=, >>=, >>>=	desna

medtem ko se izraz

```
a *= b = c -= d <<= e
```

izračuna kot

```
a *= (b = (c -= (d <<= e)))
```

Pri eniških operatorjih glede asociativnosti nimamo prav dosti izbire (npr. izraz `!!a` se ne more izračunati drugače kot `!(!a)`), pri trojiškem operatorju pa desnoasociativnost pomeni, da se izraz

```
a ? b : c ? d : e
```

izračuna kot

```
a ? b : (c ? d : e)
```

**Naloga 3.52** Kakšna je vrednost sledečega izraza?

```
3 << 4 < 6 << 3 ? 9 >> 1 : (6 & ~5) == (2 ^ 1 ^ 1) ?
7 * 3 << 1 : 5 | 6
```

**Naloga 3.53** Na začetku imajo vse spremenljivke vrednost 3. Kakšne vrednosti



imajo po izvedbi sledečega stavka?

```
a += b *= c -= d |= e ^= f <=<= --g;
```

### 3.14 Povzetek

- Zaporedje stavkov se izvede preprosto tako, da se stavki izvršijo eden za drugim.
- Pogojni stavek ima dve obliki: s prvo (*if*) lahko dosežemo, da se določeno zaporedje stavkov izvrši samo tedaj, ko je izpolnjen določen pogoj, z drugo (*if-else*) pa lahko poleg tega navedemo tudi zaporedje stavkov, ki se izvrši, če podani pogoj ni izpolnjen.
- Pogoj v glavi pogojnega stavka, zanke ali stavka *switch* mora biti logični izraz. Logični izraz ima vrednost tipa *boolean*, torej *true* ali *false*. Logične izraze lahko med seboj povezujemo z logičnimi operatorji. Operatorja *&&* in *||* se izvajata kratkostično.
- Izvajanje zanke *while* se prične s preverjanjem pogoja v glavi zanke. Če pogoj ni izpolnjen, se izvajanje zanke takoj zaključi. V nasprotnem primeru najprej izvedemo telo zanke, nato pa se vrnemo na pogoj v glavi — in cikel se ponovi.
- Zanka *do* je podobna zanki *while*, le da se pogoj preverja ob koncu vsakega obhoda. To pomeni, da se telo zanke v vsakem primeru izvede vsaj enkrat.
- Zanko *for* praviloma uporabljamo v primerih, ko želimo določeno zaporedje stavkov izvršiti za vsako vrednost, ki jo zavzame števec, ko teče po podanem intervalu.
- S stavkom *break* lahko prekinemo izvajanje zanke ali stavka *switch*, v kateri oz. katerem se neposredno nahaja. Če želimo prekiniti katero od oklepajočih zank, si pomagamo z oznako.
- Stavek *continue* preskoči vse stavke do konca telesa zanke, v kateri se neposredno nahaja. Pri zankah *while* in *do* nadaljuje s preverjanjem zankega pogoja, pri zankah *for* pa s tretjo komponento glave. Če si pomagamo z oznako, lahko dosežemo, da stavek *continue* vpliva tudi na izvajanje katere od oklepajočih zank.
- Stavek *switch* je poseben primer verige pogojnih stavkov. Uporabljamo ga takrat, ko želimo zaporedje stavkov, ki se bo izvršilo, izbrati na podlagi ene od možnih vrednosti izraza.

- S pogojnim operatorjem lahko definiramo pogojne izraze. Pogojni izraz je sestavljen iz pogoja, podizraza, ki se izračuna v primeru izpolnjenosti pogoja, in podizraza, ki se izračuna takrat, ko pogoj ni izpolnjen.
- Bitni operatorji nam omogočajo izvajanje operacij nad posameznimi biti, ki tvorijo števila.
- Javanske operatorje lahko razvrstimo v prednostno lestvico. Poleg tega so nekateri operatorji levoasociativni, drugi pa desnoasociativni.

### ***Iz profesorjevega kabineta***

Profesor Doberšek meni, da je trikotnik parov  $(i, j)$  z lastnostjo  $1 \leq i < j \leq n$  mogoče izpisati samo z vgnezdено zanko:

```
for (int i = 1; i < n; i++) {
    for (int j = i + 1; j <= n; j++) {
        System.out.printf("%d:%d ", i, j);
    }
    System.out.println();
}
```

Asistent Slapšak vidi več. Med drugim tudi to, da bi bila enojna zanka čisto dovolj. No, v tem primeru bi morda rešitev lepše spisal s katero drugo zanko, denimo *while*.

Ko za vnovično merjenje moči med profesorjem in asistentom sliši docentka Javornik, prezirljivo prhne, češ kdo še danes potrebuje zanke, in iz rokava strese rešitev brez ene same take nebodijetreba.

Napišite asistentovo rešitev, docentkino pa si lahko mirno prihranite za poglavje ali dve kasneje, ko bomo o programiranju vedeli še malo več.

## 4 Metode

Pri programiranju se nam pogosto zgodi, da moramo isti postopek izvesti na več različnih mestih. Da se izognemo podvajanju, lahko kodo, ki izvrši tak postopek, preoblikujemo v *metodo* in jo vsakokrat izvedemo s preprostim *klicem*. Po nenapisanih pravilih naj bi pravzaprav vsak zaokrožen podproblem zapisali v obliki metode. Tako sestavljeni programi so preglednejši in jih je lažje vzdrževati, tudi če se vsaka metoda pokliče samo po enkrat.

### 4.1 Metoda in klic

Začnimo s preprostim primerom. Napišimo program, ki izpiše sledeči vzorec:

```
+
+
+
+
+++++++
+
+
+
+
```

Problem bi lahko rešili z devetimi stavki `System.out.println(...)`, vendar pa se ga bomo lotili z zankami. Na ta način bomo dobili rešitev, ki je ne bo težko nadgraditi v poljubno velike vzorce iste oblike.

Lotimo se dela. Naš vzorec je sestavljen iz

1. štirih vrstic, od katerih je vsaka sestavljena iz štirih presledkov in plusa;
2. ene vrstice z devetimi plusi;
3. štirih vrstic, od katerih je vsaka sestavljena iz štirih presledkov in plusa.

Če ta opis pretvorimo v program, dobimo sledeče:

```
// koda 4.1 (metode/Kriz1.java)
public class Kriz1 {
    public static void main(String[] args) {
        // 1
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= 4; j++) {
                System.out.print(" ");
            }
            System.out.println("+");
        }

        // 2
        for (int i = 1; i <= 9; i++) {
            System.out.print("+");
        }
        System.out.println();

        // 3
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= 4; j++) {
                System.out.print(" ");
            }
            System.out.println("+");
        }
    }
}
```

Vidimo, da je odsek kode, označen s številko 3, popolna kopija odseka, označenega s številko 1. Takšno podvajanje je v nasprotju z načelom »ne ponavljaj se« (angl. DRY — don't repeat yourself). Program s podvojenimi odseki kode je namreč daljši, kot bi lahko bil, poleg tega pa ga je tudi težje vzdrževati: če določen odsek popravimo ali nadgradimo, moramo (ponavadi) enako ravnati tudi z vsemi njegovimi kopijami.

Če se želimo izogniti ponavljanju, preoblikujemo odsek kode v *metodo*. Metoda v osnovi ni nič drugega kot poimenovan kos kode. V našem primeru bi definirali metodo z imenom `navpicniKрак` (za imena metod veljajo enaka pravila kot za imena spremenljivk), ki nariše enega od navpičnih krakov križa:

```
public static void navpicniKрак() {
    for (int i = 1; i <= 4; i++) {
        for (int j = 1; j <= 4; j++) {
            System.out.print(" ");
        }
    }
}
```

```

        System.out.println("+");
    }
}

```

Naš program (oziroma razred, če smo natančnejši) ima sedaj poleg metode `main` še metodo `navpicniKraK`.

Če želimo, da se bo koda metode izvršila, moramo metodo *poklicati*. To storimo tako, da navedemo ime metode, nato pa še par oklepajev:

```

// koda 4.2 (metode/Kriz2.java)
public class Kriz2 {
    public static void main(String[] args) {
        navpicniKraK(); // klic metode navpicniKraK

        for (int i = 1; i <= 9; i++) {
            System.out.print("+");
        }
        System.out.println();

        navpicniKraK(); // klic metode navpicniKraK
    }

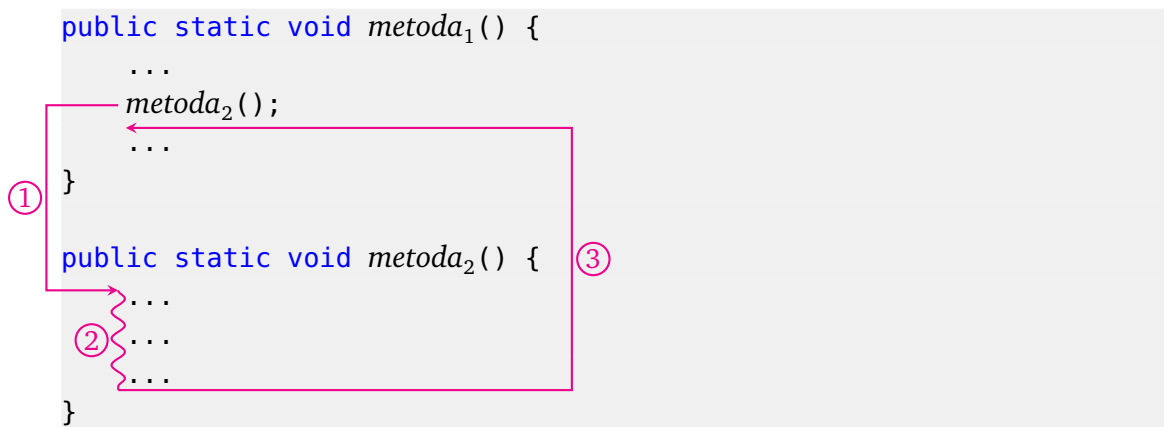
    public static void navpicniKraK() {
        for (int i = 1; i <= 4; i++) {
            for (int j = 1; j <= 4; j++) {
                System.out.print(" ");
            }
            System.out.println("+");
        }
    }
}

```

Vsak program se prične izvajati v metodi `main`. Ko pridemo do klica neke metode, se ta metoda nemudoma začne izvajati. Ko se klicana metoda zaključi, se vrnemo na mesto klica in nadaljujemo z naslednjim stavkom. Na sliki 4.1 se v metodi *metoda<sub>1</sub>* pokliče metoda *metoda<sub>2</sub>*. Ta se izvrši, nato pa metoda *metoda<sub>1</sub>* nadaljuje s svojim izvajanjem.

## 4.2 Parametri metode

Primer iz prejšnjega razdelka bomo sedaj posplošili. Namesto križa s krakom dolžine 4 bomo narisali križ s krakom dolžine  $n$ . Na primer, za  $n = 3$  izgleda naš križ takole:



Slika 4.1 Klic metode.

```

+
+
+
+++++++
+
+
+

```

V primeru iz prejšnjega razdelka je bil navpični krak sestavljen iz štirih vrstic s po štirimi presledki in plusom; sedaj imamo  $n$  vrstic s po  $n$  presledki in plusom. Sredinska linija je bila sestavljena iz devetih plusov, sedaj pa jih premore  $2n + 1$ . Poskusimo:

```

import java.util.Scanner;

public class Kriz3 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();    // (1)

        navpicniKraz();

        for (int i = 1; i <= 2 * n + 1; i++) {
            System.out.print("+");
        }
        System.out.println();
    }
}

```

```

        navpicniKрак();
    } // (2)

    public static void navpicniKрак() {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                System.out.print(" ");
            }
            System.out.println("+");
        }
    }
}

```

Program nam morda izgleda v redu, žal pa se prevajalnik ne strinja. Sporoči nam, da spremenljivka *n* v metodi *navpicniKрак* ni deklarirana:

```

Kriz3.java:20: error: cannot find symbol
        for (int i = 1; i <= n; i++) {
                           ^
symbol:   variable n
location: class Kriz3
...

```

Seveda: spremenljivka obstaja samo do konca bloka, v katerem je deklarirana, telo metode pa se obnaša kot blok. Spremenljivka *n*, deklarirana v vrstici (1), potemtaka obstaja samo do zaklepaja v vrstici (2). Metoda *navpicniKрак* spremenljivke *n* ne vidi.

Če si metode spremenljivk ne morejo kar tako deliti, kako si lahko potem posredujejo podatke, ki jih potrebujejo? Ta problem lahko rešimo s pomočjo *parametrov*. Parametri so spremenljivke, ki jih deklariramo znotraj para oklepajev v glavi metode. Parametre lahko v telesu metode uporabljamo popolnoma enako kot spremenljivke, deklarirane v sami metodi (tem rečemo tudi *lokalne spremenljivke*). Ko metodo pokličemo, moramo (zopet znotraj para oklepajev) navesti *argumente* — konkretne vrednosti za posamezne parametre. Argumenti se ob klicu metode po vrsti *skopirajo* v parametre v glavi metode.<sup>1</sup>

Metoda *navpicniKрак* potrebuje en sam zunanji podatek: višino kraka. Ta podatek bomo zato deklarirali kot parameter metode. Pri deklaraciji parametra moramo enako kot pri deklaraciji običajne spremenljivke navesti tip in ime. Pri tipu ni dileme (*int*, jasno), ime pa naj bo, denimo, *visina*:

<sup>1</sup>Terminologija tukaj ni enotna. Poleg izrazov *parameter* in *argument* se uporabljata tudi izraza *formalni parameter* in *dejanski parameter* (npr. v Mahnič in sod. (2008)) ter izraza *formalni argument* in *dejanski argument* (npr. v prevajalnikovih obvestilih o napakah).

```

public static void navpicniKrak(int visina) {
    for (int i = 1; i <= visina; i++) {
        for (int j = 1; j <= visina; j++) {
            System.out.print(" ");
        }
        System.out.println("+");
    }
}

```

Ob vsakem klicu metode moramo podati argument za parameter `visina`. V našem primeru je to kar število  $n$  (vrednost spremenljivke  $n$ ):

```

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();

    navpicniKrak(n);

    for (int i = 1; i <= 2 * n + 1; i++) {
        System.out.print("+");
    }
    System.out.println();

    navpicniKrak(n);
}

```

Ko se metoda `navpicniKrak` pokliče, se vrednost spremenljivke  $n$  skopira v parameter `visina`.

Parameter metode `navpicniKrak` bi lahko poimenovali tudi kar  $n$ . S tem ne bi bilo prav nič narobe, saj bi se ime  $n$  v metodi `navpicniKrak` nanašalo na povsem drugo spremenljivko kot ime  $n$  v metodi `main`. Vrednost spremenljivke  $n$  bi se ob klicu metode skopirala v parameter  $n$ , to pa je tudi vse, kar bi imeli spremenljivki skupnega. Tudi če bi parameter  $n$  v metodi `navpicniKrak` spreminjali, to ne bi imelo nikakršnega vpliva na spremenljivko  $n$  v metodi `main`. Na primer, sledeči program najprej izpiše 43, nato pa 42:

```

// koda 4.3 (metode/PrimerKlica.java)
public static void main(String[] args) {
    int a = 42;
    f(a);
    System.out.println(a); // 42
}

```



```
public static void f(int a) {
    a++;
    System.out.println(a); // 43
}
```

Parameter  $a$  v metodi  $f$  se »rodi« šele ob klicu metode  $f$ . Ob »rojstvu« dobi vrednost 42. Ko se metoda zaključi, ta parameter »umre«, nato pa se vrnemo v metodo `main`. Spremenljivka  $a$  v tej metodi še vedno obstaja in še vedno ima vrednost 42.

Tako parametre kot argumente med seboj ločimo z vejicami. Definicija metode in njen klic v splošnem torej izgledata tako:

```
public static void metoda( $T_1$   $p_1$ ,  $T_2$   $p_2$ , ...,  $T_n$   $p_n$ ) {
    ...
}

public static void ...(...) {
    ...
    metoda( $a_1$ ,  $a_2$ , ...,  $a_n$ );
    ...
}
```

Ob klicu se izvedejo sledeči prireditveni stavki:

```
 $p_1$  =  $a_1$ ;
 $p_2$  =  $a_2$ ;
...
 $p_n$  =  $a_n$ ;
```

Število argumentov se mora ujemati s številom parametrov, poleg tega pa morajo biti tipi argumentov priredljivi tipom istoležnih parametrov. Na primer, metodo z glavo

```
public static void f(int a, double b)
```

lahko pokličemo s kodo `f(3, 4.0)` ali `f(3, 4)`, klic `f(3.0, 4)` pa se ne prevede.

Vrnimo se na naš tekoči primer. Definirajmo in uporabimo še metodo za izris sredinske linije:

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();

    navpicniKrak(n);
    sredinskaLinija(n);
}
```

```

    navpicniKrak(n);
}

public static void sredinskaLinija(int n) {
    for (int i = 1; i <= 2 * n + 1; i++) {
        System.out.print("+");
    }
    System.out.println();
}

```

Metoda main sedaj le še prebere število  $n$ , vse ostalo pa opravita metodi `navpicniKrak` in `sredinskaLinija`. Metoda main ju zgolj pokliče. (Če ju ne bi, se metodi seveda ne bi izvedli.)

Pri izrisu križa (in tudi pri drugih podobnih nalogah) si lahko pomagamo z metodo, ki  $n$ -krat izpiše podani znak in po želji doda še prelom vrstice. Ta metoda potrebuje tri parametre:

- število izpisov znaka (tj. število  $n$ );
- znak, ki naj se izpisuje;
- podatek o tem, ali naj se na koncu izpiše še prelom vrstice.

Prvi parameter je tipa `int`, drugi tipa `char`, tretji pa tipa `boolean`. Metodi bomo dali ime `zaporedje`.

```

// koda 4.4
public static void zaporedje(int n, char znak, boolean prelom) {
    for (int i = 1; i <= n; i++) {
        System.out.print(znak);
    }
    if (prelom) {
        System.out.println();
    }
}

```

Z uporabo te metode postane preostanek programa še nekoliko preglednejši, saj lahko izris sredinske linije izrazimo kot izpis  $2n + 1$  znakov `+` z dodanim prelomom vrstice, izris vsake posamezne vrstice navpičnega kraka višine `visina` pa kot izpis `visina` presledkov in enega znaka `+` s sledečim prelomom vrstice.

```

// koda 4.5 (metode/Kriz4.java)
import java.util.Scanner;

public class Kriz4 {

```

```

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    navpicniKrak(n);
    sredinskaLinija(n);
    navpicniKrak(n);
}

public static void navpicniKrak(int visina) {
    for (int i = 1; i <= visina; i++) {
        zaporedje(visina, ' ', false);
        zaporedje(1, '+', true);
    }
}

public static void sredinskaLinija(int n) {
    zaporedje(2 * n + 1, '+', true);
}

public static void zaporedje(int n, char znak, boolean prelom) {
    for (int i = 1; i <= n; i++) {
        System.out.print(znak);
    }
    if (prelom) {
        System.out.println();
    }
}
}

```

Vrstni red metod v razredu je lahko poljuben. Kljub temu pa je smiselno, da metode vsaj v grobem razporedimo bodisi »od zgoraj navzdol« (najprej main, nato metode, ki jih main kliče, nato metode, ki jih kličejo te metode itd.) bodisi »od spodaj navzgor« (najprej metode na dnu hierarhije klicev, na koncu pa main).

**Naloga 4.1** Napišite program, ki prebere števili  $n$  in  $k$  in nariše šahovnico, sestavljeno iz  $n \times n$  polj, pri čemer je vsako polje sestavljeno iz  $k \times k$  znakov - (za bela polja) oziroma \* (za črna polja). Zgornje levo polje je vedno belo. Program smiselno razdelite na metode.

### 4.3 Vračanje vrednosti

Videli smo, da lahko kličoča metoda preko argumentov posreduje podatke klicani metodi. Prenos podatkov pa je mogoč tudi v obratni smeri: klicana metoda lahko proizvede rezultat, ki ga je nato (po zaključku klicane metode) mogoče uporabiti v kličoči metodi.

Napišimo metodo, ki sprejme dolžini stranic pravokotnika in izpiše njegov obseg:

```
public static void obseg(int a, int b) {
    System.out.println(2 * (a + b));
}
```

Sledeči klic izpiše obseg pravokotnika velikosti  $3 \times 4$ :

```
obseg(3, 4);
```

Ta klic sicer izpiše iskani obseg pravokotnika, vendar pa izpisanega rezultata ne moremo več uporabiti v nadaljnjih izračunih. To bi bilo mogoče, če bi lahko izračunani obseg shranili v spremenljivko. Žal pa metoda obseg v svoji trenutni obliki tega ne omogoča; stavek

```
int ob = obseg(3, 4);
```

se ne prevede.

Če želimo imeti možnost, da rezultat klica metode priredimo spremenljivki, mora metoda svoj rezultat *vrniti*. To storimo tako, da besedo `void` v glavi metode zamenjamo s tipom vrednosti, ki jo metoda vrne, samo vračanje pa izvršimo s stavkom `return`:

```
// metoda vrne vrednost tipa int
public static int obseg(int a, int b) {
    return 2 * (a + b);
}
```

Sedaj lahko rezultat klica metode priredimo spremenljivki. Stavek

```
int ob = obseg(3, 4);
```

se prevede, spremenljivka `ob` pa dobi vrednost 14. Klic `obseg(3, 4)` je *izraz*, ker ima svojo vrednost: to je tisto, kar metoda vrne, torej 14.

Stavek `return` *takoj* zaključí metodo, v kateri se nahaja, in nas vrne na mesto, kjer smo metodo poklicali. Klic metode se nato (navidez) nadomesti z vrnjeno vrednostjo. Za lažje razumevanje si natančno oglejmo, kako se izvede stavek `int ob = obseg(3, 4);`:

1. Pokliče se metoda `obseg`. Argument 3 se skopira v parameter `a`, argument 4

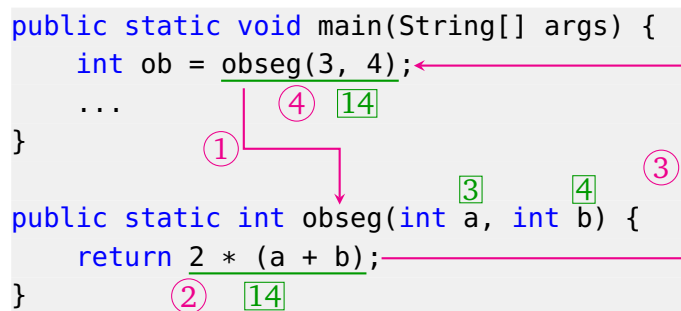
pa v parameter b.

2. Ker ima parameter a vrednost 3, parameter b pa 4, ima izraz  $2 * (a + b)$  vrednost 14. Izvede se torej stavek `return 14`.
3. Vrnemo se na klic `obseg(3, 4)`.
4. Ta klic se navidezno nadomesti z vrnjeno vrednostjo (številom 14). Celoten stavek se torej navidezno preoblikuje v stavek

```
int ob = 14;
```

5. Spremenljivki `ob` se priredi vrednost 14.

Opisani klic metode `obseg` je prikazan na sliki 4.2. Zaporedne številke korakov so obkrožene, vrednosti spremenljivk oz. izrazov pa uokvirjene.



Slika 4.2 Klic metode, ki vrne vrednost.

Rezultat klica lahko tudi neposredno uporabimo; ni nujno, da ga priredimo spremenljivki. Na primer:

```
if (obseg(a, b) > meja) {
    System.out.println("Pravokotnik je prevelik!");
}
```

Izraz `obseg(a, b)` se po izvedbi metode `obseg` navidez nadomesti z vrnjeno vrednostjo, ta pa se nato primerja s spremenljivko `meja`.

Splošna oblika glave metode, kot smo jo spoznali doslej, je potemtakem takšna:

```
public static R metoda( $T_1$   $p_1$ ,  $T_2$   $p_2$ , ...,  $T_n$   $p_n$ )
```

Tip  $R$  imenujemo *izhodni tip* metode. Tip vrednosti, ki jo metoda vrača, mora biti priredljiv tipu  $R$ . Če metoda ne vrača ničesar, mora biti  $R$  enak `void`. (Beseda `void` označuje odsotnost tipa.) Če  $R$  ni enak `void`, potem metoda *mora* vrniti vrednost

ustreznega tipa, in to v vseh možnih scenarijih izvajanja. Na primer, sledeča metoda se ne prevede, ker ne vrne vrednosti v primeru, ko vrednost spremenljivke *teza* pripada intervalu  $[5, 9]$ :

```
public static boolean preveri(int teza) {
    if (teza >= 10) {
        return true;
    } else if (teza < 5) {    // (1)
        return false;
    }
}
```

Nekoliko presentljivo je, da se metoda ne prevede niti tedaj, ko število 5 v vrstici (1) nadomestimo s številom 10. Prevajalnik pač ne ugotovi, da tretja veja ni mogoča. Problem lahko rešimo takole ...

```
if (teza >= 10) {
    return true;
} else {
    return false;
}
```

... lahko pa tudi takole ...

```
if (teza >= 10) {
    return true;
}
return false;
```

... ali kar takole:

```
return teza >= 10;
```

Omenimo še, da lahko stavek `return` uporabljamo tudi za predčasno prekinitev metode, ki ne vrača ničesar. V tem primeru pišemo samo

```
return;
```

#### Naloga 4.2 Napišite in preizkusite metodo

```
long fakultetaSPreskokom(int n, int p)
```

ki vrne zmnožek  $n(n-p)(n-2p)\dots$  (zadnji člen je število z intervala  $[1, p]$ ). Na primer, klic `fakultetaSPreskokom(11, 3)` naj vrne vrednost 880 ( $11 \cdot 8 \cdot 5 \cdot 2$ ).

#### Naloga 4.3 Napišite in preizkusite metodo

```
long fakulteta(int n)
```

ki vrne vrednost  $n!$ . Metoda naj ustrezno pokliče metodo iz prejšnje naloge.

## 4.4 Metode in podproblemi

Metode niso namenjene samo preprečevanju podvajanja kode. Gre za ključno orodje pri razbijanju obsežnejših problemov na manjše podprobleme. Kot smo že omenili, načela dobre prakse veleavajo, da vsak smiselno zaokrožen podproblem sprogramiramo v obliki metode. Ali določen kos kode tvori smiselno zaokrožen podproblem ali ne, je dostikrat bolj stvar občutka kot trdnih pravil, k sreči pa se tovrstni občutki z naraščanjem programerske kilometrine samodejno izboljšujejo.

### 4.4.1 (Že spet) praštevila

Tokrat ne bomo odkrivali še hitrejših algoritmov za izpis praštevil od 2 do  $n$ , ampak bomo že napisani program samo nekoliko preoblikovali. V vseh različicah, ki smo jih preizkusili, smo problem rešili tako, da smo se z zanko sprehodili po — najprej vseh, kasneje pa zgolj lihih — številih od 2 do  $n$  in za vsako število preverili, ali je praštevilo. Preverjanje praštevilskosti posameznega števila pa je smiselno zaokrožen problem, zato si zasluži svojo metodo. Metoda, rekli ji bomo `jePrastevilo`, sprejme liho število in vrne vrednost `true`, če je število praštevilo, in `false`, če ni. Program `Prastevila4` (koda 3.38) bi torej lahko prepisali takole:

```
// koda 4.6 (metode/Prastevila5.java)
import java.util.Scanner;

public class Prastevila5 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        System.out.println(2);
        for (int kandidat = 3; kandidat <= n; kandidat += 2) {
            if (jePrastevilo(kandidat)) {
                System.out.println(kandidat);
            }
        }
    }

    // Vrne true natanko v primeru, če je podano liho število
    // praštevilo.
}
```

```

public static boolean jePrastevilo(int lihoStevilo) {
    boolean prastevilo = true;
    int meja = (int) Math.round(Math.sqrt(lihoStevilo));
    for (int d = 3; d <= meja; d += 2) {
        if (lihoStevilo % d == 0) {
            prastevilo = false;
            break;
        }
    }
    return prastevilo;
}

```

Ker se takoj po izvedbi stavka `break` izvede stavek `return`, lahko kodo skrajšamo. Izkaže se, da logične spremenljivke ne potrebujemo:

```

public static boolean jePrastevilo(int lihoStevilo) {
    int meja = (int) Math.round(Math.sqrt(lihoStevilo));
    for (int d = 3; d <= meja; d += 2) {
        if (lihoStevilo % d == 0) {
            return false;
        }
    }
    return true;
}

```

■ **Naloga 4.4** Program iz naloge 3.32 smiselno razbijte na metode.

#### 4.4.2 Prijateljska števila

**Primer 4.1.** Naj bo  $S(n)$  vsota deliteljev števila  $n$  brez števila  $n$  samega. Na primer,  $S(20) = 1 + 2 + 4 + 5 + 10 = 22$ . Števili  $a$  in  $b$  sta *prijateljski*, če so izpolnjeni sledeči pogoji:

- $a \neq b$ ;
- $S(a) = b$ ;
- $S(b) = a$ .

Prijateljski števili sta, na primer, 220 in 284, saj sta različni, poleg tega pa velja  $S(220) = 284$  in  $S(284) = 220$ . Napišimo program, ki prebere število  $n$  in za vsako število od 1 do  $n$  izpiše njegovega prijatelja, če ta obstaja.



*Rešitev.* Najprej razčistimo tole: če ima število  $k$  prijatelja, je ta lahko samo eden — število  $S(k)$ . Problem lahko torej rešimo tako, da za vsako število  $k \in \{1, \dots, n\}$  izračunamo  $k' = S(k)$ , nato pa preverimo, ali velja  $k \neq k'$  in  $S(k') = k$ . Če sta oba pogoja izpolnjena, je  $k'$  prijatelj števila  $k$ , sicer pa vemo, da  $k$  nima prijatelja.

Izračun vsote  $S(\cdot)$  se nam splača zapisati v obliki metode, še posebej zato, ker ga bomo potrebovali dvakrat. Metoda sprejme število tipa `int`, vrne pa prav tako število tipa `int` — vrednost  $S(\cdot)$  za podano število.

```
// koda 4.7 (metode/Prijatelji.java)
import java.util.Scanner;

public class Prijatelji {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        for (int a = 1; a <= n; a++) {
            int b = vsotaDeliteljev(a);
            if (a != b && vsotaDeliteljev(b) == a) {
                System.out.printf("%d -> %d%n", a, b);
            }
        }
    }

    // Vrne vsoto deliteljev podanega števila, pri čemer
    // števila samega ne vključi v vsoto.
    public static int vsotaDeliteljev(int stevilo) {
        int vsota = 0;
        for (int d = 1; d < stevilo; d++) {
            if (stevilo % d == 0) {
                vsota += d;
            }
        }
        return vsota;
    }
}
```

□

**Naloga 4.5** Izboljšajte metodo `vsotaDeliteljev` tako, da bo zanka potovala kvečjemu do (zaokroženega) kvadratnega korena podanega števila. Primerjajte trajanje izhodiščnega in izboljšanega programa za različne vrednosti  $n$ .

**Naloga 4.6** Izpeljite formulo za izračun vsote deliteljev podanega števila in jo uporabite v metodi `vsotaDeliteljev`. Izziva se lahko lotite na podoben način kot naloge 3.38.

#### 4.4.3 Pitagorejska števila

V razdelku 3.9.4 smo napisali program, ki izpiše število pitagorejskih števil na intervalu  $[1, n]$ . Problema smo se lotili takole:

```
int stPitagorejskih = 0;
for (int c = 1; c <= meja; c++) {
    if (število c je pitagorejsko) {
        stPitagorejskih++;
    }
}
System.out.println(stPitagorejskih);
```

Pogoj v stavku `if` naravnost kriči po klicu metode. Ta metoda, imenujmo jo kar `jePitagorejsko`, sprejme število in vrne `true` natanko v primeru, ko je število pitagorejsko. Ker lahko iz metode predčasno izstopimo s stavkom `return`, se spotoma znebimo še »grdega« stavka `break`.

```
// koda 4.8 (metode/PitagorejskaStevila.java)
import java.util.Scanner;

public class PitagorejskaStevila {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int meja = sc.nextInt();

        int stPitagorejskih = 0;
        for (int c = 1; c <= meja; c++) {
            if (jePitagorejsko(c)) {
                stPitagorejskih++;
            }
        }
        System.out.println(stPitagorejskih);
    }

    public static boolean jePitagorejsko(int c) {
        for (int a = 1; a < c - 1; a++) {
            for (int b = a + 1; b < c; b++) {
                if (c * c == a * a + b * b) {
```

```

        return true;
    }
}
return false;
}
}

```

**Naloga 4.7** Po zgledu naloge 3.40 prepisite metodo `jePitagorejsko` tako, da bo imela eno samo zanko.

## 4.5 Rekurzija

Kot že vemo, lahko metoda pokliče poljubno drugo metodo, le argumente ustreznih tipov ji mora posredovati. Sedaj pa si bomo zastavili na prvi pogled prismuknjeno vprašanje: ali lahko metoda pokliče tudi samo sebe? Odgovor se glasi: da, lahko! Še več: izkaže se, da ta ideja ni tako čudaška, kot se morda zdi, in da je z *rekurzijo*, kot pravimo situaciji, ko metoda kliče samo sebe, mogoče številne probleme rešiti precej bolj elegantno, kot bi jih lahko, če rekurzija ne bi bila mogoča.

Spomnimo se, kako se izvede klic metode:

1. Ustvari se parametri klicane metode. Ti se obnašajo kot navadne spremenljivke.
2. Argumenti se skopirajo v parametre.
3. Klicana metoda se izvede do stavka `return` oziroma do konca, če ne vsebuje tega stavka.
4. Vrnemo se na mesto klica v kličočni metodi.
5. Če je klicana metoda vrnila vrednost, se njen klic navidez nadomesti s to vrednostjo.
6. Kličočna metoda se po običajnih pravilih izvaja naprej.

Rekurzivni klic se izvrši popolnoma enako kot klic katerekoli druge metode; edina razlika je ta, da kličočna in klicana metoda sovpadata.

Če nismo previdni, nas lahko rekurzija privede do programa, ki se ne ustavi. Oglejmo si preprost primer:

```

public static void f(int n) {
    f(n - 1);
    System.out.println(n);
}

```

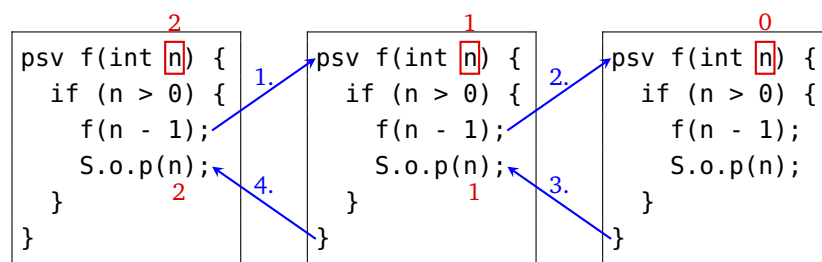
Kaj se zgodi, če metodo *f* pokličemo, denimo, z argumentom 2? Klic *f*(2) sproži klic *f*(1). Ta sproži klic *f*(0), ta *f*(-1) itd. Vidimo, da se izvajanje nikoli ne izteče, in to ne glede na začetno vrednost parametra *n*. Stavek `System.out.println(n)` se nikoli ne izvede.<sup>2</sup>

Če metodo *f* nekoliko dopolnimo ...

```
// koda 4.9 (metode/Rekurzija.java)
public static void f(int n) {
    if (n > 0) {
        f(n - 1);
        System.out.println(n);
    }
}
```

... pa so razmere povsem drugačne. Program se sedaj ustavi za vsako vrednost *n*. Oglejmo si, kako se izvede klic *f*(2). Ta klic sproži klic *f*(1), ta pa *f*(0). Klic *f*(0) se takoj zaključi, saj pogoj v glavi stavka `if` ni izpolnjen. Izvajalnik se zato vrne na izvajanje klica *f*(1). V okviru tega klica se je stavek *f*(0) pravkar zaključil, sedaj pa je treba izvesti še stavek `System.out.println(1)`. Ko se izpiše število 1, se klic *f*(1) zaključi, zato se izvajalnik vrne na izvajanje klica *f*(2). V okviru tega klica se je stavek *f*(1) ravnokar uspešno zaključil, zato izvajalnik nadaljuje s stavkom `System.out.println(2)`. Izpiše se število 2 in klic *f*(2) se s tem zaključi. Celotno zaporedje klicev je prikazano na sliki 4.3.

Pomembno se je zavedati, da ima vsak klic metode svojo kopijo spremenljivke *n*. Po vrnitvi iz rekurzivnega klica ima spremenljivka *n* točno takšno vrednost, kot jo je imela tik pred klicem. (Ta lastnost seveda velja za vse klice metod, ne samo za rekurzivne. Rekurzivni klici se obnašajo natanko tako kot nerekurzivni.)



**Slika 4.3** Primer klica rekurzivne metode (začetni klic je *f*(2)).

<sup>2</sup>V praksi se izvajanje dokaj hitro zaključí, ker programu zmanjka *sklada*, razmeroma majhnega kosa pomnilnika, namenjenega hranjenju vrednosti parametrov in lokalnih spremenljivk klicanih metod. V vsakem klicu metode (tudi če gre za eno in isto metodo) se parametri in lokalne spremenljivke ustvarijo *na novo*, nekaj pomnilnika pa je potrebnega tudi za sam klic, saj si mora izvajalnik shraniti podatek o tem, kam naj se vrne, ko metoda opravi svoje delo.

Rekurzivne metode so pogosto povezane z matematičnimi rekurenčnimi relacijami. Lep primer je *Fibonaccijevo zaporedje*. To je zaporedje števil  $F_0, F_1, F_2 \dots$ , pri čemer je člen  $F_n$  definiran takole:

$$F_n = \begin{cases} 0 & \text{pri } n = 0; \\ 1 & \text{pri } n = 1; \\ F_{n-2} + F_{n-1} & \text{sicer.} \end{cases}$$

Na primer,  $F_3 = F_1 + F_2 = 1 + (F_0 + F_1) = 1 + (0 + 1) = 1 + 1 = 2$ .

Definicijo splošnega Fibonaccijevega člena  $F_n$  zlahka prepišemo v javansko kodo:

```
// koda 4.10
public static int fib(int n) {
    if (n <= 1) {
        return n;
    }
    int pp = fib(n - 2);
    int p = fib(n - 1);
    return pp + p;
}
```

Oglejmo si, kako se izvede klic `fib(3)`:

```
fib(3) pokliče fib(1)
    fib(1) vrne 1
fib(3) nastavi pp = 1
fib(3) pokliče fib(2)
    fib(2) pokliče fib(0)
        fib(0) vrne 0
    fib(2) nastavi pp = 0
    fib(2) pokliče fib(1)
        fib(1) vrne 1
    fib(2) nastavi p = 1
    fib(2) izračuna pp + p in vrne 1
fib(3) nastavi p = 1
fib(3) izračuna pp + p in vrne 2
```

Kodo 4.10 bi lahko tudi skrajšali. Spremenljivki `pp` in `p` smo namreč dodali le za lažjo simulacijo klica, v resnici pa ju ne potrebujemo:

```
// koda 4.11 (metode/Fibonacci.java)
public static int fib(int n) {
    if (n <= 1) {
        return n;
    }
}
```

```

    }
    return fib(n - 2) + fib(n - 1);
}

```

Metoda `fib` je kratka in jedrnata, žal pa ni posebej hitra. Že izračun `fib(50)` traja nekaj deset sekund, na rezultat klica `fib(100)` pa bi bržkone čakali več let.<sup>3</sup> V razdelku 5.9 bomo poiskali vzrok za takšno neučinkovitost in ugotovili, da lahko metodo z drobno dopolnitvijo bistveno pohitrimo.

**Naloga 4.8** Prepišite metodo `fib` tako, da ne bo uporabljala rekurzije. Metoda se mora tudi pri  $n = 10^6$  izvesti v manj kot sekundi.

**Naloga 4.9** Prepišite metodo zaporedje (koda 4.4) tako, da ne bo vsebovala nobene zanke. Metoda si lahko pomaga s katero drugo metodo, seveda pa tudi ta ne sme vsebovati zank.

**Naloga 4.10** Napišite program za izpis praštevil od 1 do  $n$ , ki nima niti ene zanke.

## 4.6 Povzetek

- Metoda je poimenovan kos kode. Metodo lahko pokličemo in s tem izvršimo kodo, ki jo vsebuje.
- Metoda lahko sprejema parametre. Parametri se znotraj metode obnašajo kot spremenljivke. Ob klicu take metode moramo podati vrednosti (argumente), ki se bodo po vrsti skopirale v parametre metode. Tipi argumentov morajo biti zato priredljivi tipom pripadajočih parametrov.
- Metoda lahko vrne vrednost določenega tipa. Klic takšne metode je izraz, njegova vrednost pa je enaka vrednosti, ki jo vrne metoda.
- Metode so ključno orodje pri izdelavi programov, saj nam omogočajo delitev večjega in težje obvladljivega problema na manjše in lažje obvladljive podprobleme. Ponavadi za vsak smiselno zaokrožen podproblem napišemo svojo metodo.
- Metoda je rekurzivna, če kliče samo sebe. Rekurzija je močno programersko orodje, ki nam omogoča, da marsikateri problem rešimo bistveno elegantneje, kot bi ga brez nje.

<sup>3</sup>Pri takih vrednostih  $n$  bi imeli težave tudi zaradi omejitev celoštevilskih tipov, a recimo, da nam to v tem trenutku ni pomembno.

### *Iz profesorjevega kabineta*

»Število je sodo, če je njegov predhodnik lih, in liho, če je njegov predhodnik sod,« v družbi svojega najzvestejšega asistenta modruje profesor Doberšek. »To pomeni, da lahko metodi za preverjanje sodosti oziroma lihosti napišem enostavno takole:

```
public static boolean jeSodo(int n) {
    return jeLiho(n - 1);
}

public static boolean jeLiho(int n) {
    return jeSodo(n - 1);
}
```

To bi moralo delovati, mar ne, Jože?»

»Načeloma da,« v slogu znamenitih šal z radia Erevan prične asistent Slapšak, praskajoč se po malomarno obriti bradi, »a se mi kljub temu zdi, da nekaj manjka.

»Kaj naj bi to bilo?»

»Res ne bi vedel. Bi povprašal Genovefo, a ravno predava na temo ... hm, kako je že rekla? ... robnih pogojev, da, robnih pogojev!«

»Robnih pogojev! Saj vem, da rada hodi po robu, a da bi o svojih podvigih še predavala?! Le kam smo prišli! *O tempora, o mores!*«

Dopolnite metodi tako, da bosta vrnili `true` natanko tedaj, ko bo podano število sodo oziroma liho. Vaše dopolnitve si seveda ne smejo pomagati z operatorjem %, zadošča pa, da metodi delujeta le za pozitivne vrednosti parametra `n`.





## 5 Tabele

Nekaterih navidez preprostih problemov še vedno ne znamo rešiti. Na primer, znamo brati posamične rezultate atletskega tekmovanja in poiskati prvaka, toda kako bi si vse prebrane rezultate zapomnili (in jih po želji kasneje priklicali)? V takih situacijah nam pridejo prav *tabele* — zaporedja spremenljivk istega tipa.

### 5.1 Motivacijski primer

Začnimo z nalogo. Napišimo program, ki prebere število  $n$  in  $n$  števil, nato pa izpiše vsa prebrana števila v istem vrstnem redu.

Ta problem izgleda enostaven — in dejansko tak tudi je. Kljub temu pa ga z znanjem, ki smo si ga pridobili do sedaj, ne moremo rešiti. Vsako prebrano število bi namreč morali shraniti v spremenljivko in nato izpisati vse spremenljivke. Že pri fiksni vrednosti  $n$  (npr.  $n = 1000$ ) bi bila ta naloga mukotrpna, pri splošnem  $n$  pa nemogoča, saj bi potrebovali vnaprej neznano število spremenljivk. Kot bomo videli, pa problem zlahka rešimo s pomočjo tabele.

### 5.2 Tabela

Tabelo si lahko predstavljamo kot zaporedje celic, od katerih lahko vsaka hrani vrednost vnaprej določenega tipa. Ta tip mora biti za vse celice enak. Celice se obnašajo kot medsebojno neodvisne spremenljivke: iz vsake celice posebej lahko beremo in v vsako celico posebej lahko pišemo. Celice so dostopne prek *indeksov*, ki se pričnejo z ničlo. Prva celica ima potemtakem indeks 0, druga 1, tretja 2 itd. Podatek, ki ga posamezna celica vsebuje, se imenuje *element* tabele.<sup>1</sup>

Tabelo v javi deklariramo in inicializiramo na podoben način kot običajno spremenljivko: podamo njen tip in ime, pri inicializaciji pa še vsebino. Tip tabele z elementi tipa  $T$  se glasi  $T[]$ . Elemente tabele navedemo znotraj para zavrtih oklepajev. Sledeči stavek izdela tabelo z imenom *tocke* in tipom  $\text{int}[]$ , njeni elementi pa so števila 70, 50, 80, 40 in 60:

---

<sup>1</sup>Terminologija tukaj ni povsem dorečena. Beseda *element* lahko predstavlja posamezno spremenljivko (celico) ali pa njeno vrednost. Pomen je praviloma razviden iz konteksta, zato nam ta dvojnost ne bo povzročala težav.

```
int[] tocke = {70, 50, 80, 40, 60};
```

Tabela je prikazana na sliki 5.1.

	0	1	2	3	4	indeksi
tocke:	70	50	80	40	60	elementi

**Slika 5.1** Primer tabele tipa `int[]`.

### 5.3 Dostop do elementov tabele

Do elementa tabele  $t$  na indeksu  $i$  dostopamo z izrazom  $t[i]$ . Vsak element posebej lahko preberemo ali spremenimo. *Dolžino* (ali *velikost*) tabele  $t$  (število njenih elementov) pridobimo z izrazom  $t.length$ . Indeksi elementov tabele so tako od 0 do  $t.length - 1$ . Sledeči stavki se nanašajo na tabelo *tocke* na sliki 5.1:

```
System.out.println(tocke[0]);    // 70
System.out.println(tocke[3]);    // 40
tocke[3] = 90;
System.out.println(tocke[3]);    // 90
System.out.println(tocke.length); // 5
System.out.println(tocke[tocke.length - 1]); // 60
```

Če poskusimo dostopati do elementa na neveljavnem indeksu, se sproži izjema tipa `ArrayIndexOutOfBoundsException`. Na primer, stavek

```
System.out.println(tocke[5]);
```

bi se sicer normalno prevedel, v času izvajanja pa bi povzročil sledeči izpis:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 5 out of bounds for length 5
    at metoda(datoteka:vrstica)
```

Izjeme tipa `ArrayIndexOutOfBoundsException` sodijo med najpogostejše izredne dogodke, ki jih lahko doživimo pri programiranju. Premajhen ali prevelik indeks je lahko posledica napačno postavljenih meja zanke, prekratke tabele, nepričakovanega uporabnikovega vnosa ... Ko se taka izjema sproži, poskusimo izkoristiti podatke, ki nam jih ponudi izvajalnik. Iz obvestila o izjemi lahko razberemo položaj stavka, ki je povzročil izjemo, in indeks, s katerim smo želeli dostopati do tabele. Če ti podatki še ne zadoščajo za »sanacijo« kode, si pomagamo z izpisi vrednosti spremenljivk, izločanjem (komentiranjem) nebistvene kode in podobnimi tehnikami.

## 5.4 Izdelava tabele

Videli smo, da je tabelo  $t$  z elementi  $e_0, e_1, \dots, e_{n-1}$  tipa  $T$  mogoče izdelati takole:

```
T[] t = {e0, e1, ..., en-1};
```

Če ne vemo vnaprej, kakšni bodo elementi tabele, ali pa če je tabela predolga, da bi izrecno navedli vse njene elemente, jo lahko izdelamo tudi takole:

```
T[] t = new T[n];
```

Ta stavek izdelava tabelo dolžine  $n$ , v vsako od njenih celic pa se vpiše *privzeta vrednost* za tip  $T$ . Privzete vrednosti za posamezne tipe so zbrane v tabeli 5.1. (Referenčne tipe in vrednost `null` bomo spoznali v razdelku 5.10.) Na primer, stavek

```
double[] poraba = new double[12];
```

izdelava tabelo `poraba` z dvanajstimi elementi `0.0`.

**Tabela 5.1** Privzete vrednosti za posamezne tipe.

Tip	Privzeta vrednost
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0
boolean	false
char	'\0' (znak s kodo 0)
poljuben referenčni tip	null

Zapis `new T[n]` lahko uporabimo kot izraz (npr. kot »argument« stavka `return`), zapisa  $\{e_0, e_1, \dots, e_{n-1}\}$  pa ne moremo, saj prevajalnik iz njega ne more razbrati tipa tabele, ki bi jo radi izdelali. Če želimo v okviru izraza ustvariti tabelo in v isti sapi podati še njene elemente, se poslužimo sledečega zapisa:

```
new T[] {e0, e1, ..., en-1}
```

Ta izraz izdelava tabelo, njegova vrednost pa je izdelana tabela oziroma, kot bomo spoznali kasneje, kazalec nanjo.

Na primer, če želimo, da metoda vrne tabelo tipa `int[]` z elementi 1, 2 in 3, lahko to naredimo takole ...

```
int[] rezultat = {1, 2, 3};
return rezultat;
```

... lahko pa oba stavka združimo v enega:

```
return new int[]{1, 2, 3};
```

Če v tem stavku izpustimo odsek `new int[]`, se program ne bo prevedel.

Ne glede na to, kako izdelamo tabelo, se njena dolžina določi ob izdelavi (bodisi eksplicitno z navedbo dolžine bodisi implicitno z navedbo elementov tabele) in je kasneje *ni več mogoče spreminjati*. Tabele ne moremo niti »krčiti« niti »raztezati«, spreminjamo lahko zgolj posamezne elemente. Lahko pa ustvarimo *ново* tabelo z istim imenom:

```
int[] t = {1, 2, 3};
t = new int[]{4, 5};
```

Po izvedbi druge vrstice do tabele `{1, 2, 3}` ni več mogoče dostopati, čeprav v tistem trenutku morda še obstaja v pomnilniku. Spremenljivka `t` zdaj predstavlja tabelo `{4, 5}`.

## 5.5 Sprehod po tabeli

### 5.5.1 Zanki *for in for-each*

Sprehod — obisk vsakega posameznega elementa tabele — je ena od najosnovnejših operacij za delo s tabelami. Številne druge operacije, denimo računanje vsote in iskanje maksimalnega elementa, temeljijo na sprehodu. Po elementih tabele `t` se lahko sprehodimo tako, da v zanki *for* s števcem `i` potujemo od 0 do `t.length - 1` in z izrazom `t[i]` dostopamo do posameznih elementov:

```
for (int i = 0; i < t.length; i++) {
    // i: trenutni indeks
    // t[i]: element tabele na indeksu i
    ...
}
```

Na primer, sledeča koda najprej izdelata tabelo petih znakov, nato pa po vrsti izpiše veljavne indekse in elemente tabele na teh indeksih:

```
// koda 5.1 (tabele/IzpisFor.java)
char[] samoglasniki = {'a', 'e', 'i', 'o', 'u'};
for (int i = 0; i < samoglasniki.length; i++) {
    System.out.printf("Element na indeksu %d: %c%n",
        i, samoglasniki[i]);
}
```

```
}
```

Kadar potrebujemo samo posamezne elemente, indeksi pa nas ne zanimajo, se lahko po tabeli sprehodimo s posebno obliko zanke *for*, ki ji pravimo *zanka for-each*. Če je *t* tabela tipa *T[]*, se lahko po njenih elementih sprehodimo takole:

```
for (T element: t) {
    ...
}
```

V prvem obhodu zanke dobi spremenljivka *element* vrednost *t[0]*, v drugem *t[1]*, ..., v zadnjem pa *t[t.length - 1]*. Ta oblika zanke je precej kompaktnejša od klasične zanke *for*, indeksov elementov pa nimamo na voljo (razen če uvedemo ločen števec, ki ga pred zanko postavimo na 0, v zanki pa povečujemo za 1). Zanka *for-each* je zato primerna le za »bralne« sprehode; če želimo vsaki posamezni celici prirediti določeno vrednost, se poslužimo klasične zanke *for*.

Na primer, sledeča koda izpiše posamezne elemente tabele samoglasniki:

```
// koda 5.2 (tabele/IzpisForEach.java)
for (char znak: samoglasniki) {
    System.out.println(znak);
}
```

### 5.5.2 Vsota in maksimum

Ko obiskujemo elemente tabele, jih lahko mimogrede še seštevamo. Sledeča metoda vrne vsoto elementov podane tabele:

```
// koda 5.3 (tabele/Vsota.java)
public static int vsota(int[] t) {
    int vsota = 0;
    for (int element: t) {
        vsota += element;
    }
    return vsota;
}
```

Na primer, klic `vsota(new int[]{5, -2, 9, 3})` vrne vrednost 15.

Iskanje maksimalnega elementa tabele je prav tako enostavno. Sledeča metoda predpostavlja, da tabela vsebuje vsaj en element; če je njena dolžina enaka 0, se sproži izjema tipa `ArrayIndexOutOfBoundsException`.<sup>2</sup>

```
// koda 5.4 (tabele/Maksimum.java)
```

<sup>2</sup>Tabela dolžine 0 se sliši nenavadno, vendar pa java dopušča stavke oblike `T[] t = new T[0]`.

```

public static int maksimum(int[] t) {
    int naj = t[0];
    for (int element: t) {
        if (element > naj) {
            naj = element;
        }
    }
    return naj;
}

```

Spremenljivka `naj` hrani največji element izmed doslej obravnavanih.

Če nas zanima indeks maksimalnega elementa, se sprehodimo po indeksih tabele in vzdržujemo indeks doslej največjega elementa:

```

// koda 5.5 (tabele/Maksimum.java)
public static int indeksMaksimuma(int[] t) {
    int iNaj = 0;
    for (int i = 0; i < t.length; i++) {
        if (t[i] > t[iNaj]) {
            iNaj = i;
        }
    }
    return iNaj;
}

```

### Naloga 5.1 Napišite metodo

```
public static int steviloDeljivih(int[] t, int delitelj)
```

ki vrne število elementov podane tabele, ki so deljivi s podanim deliteljem.

**Naloga 5.2** Napišite metodo, ki sprejme tabelo tipa `int[]` in izpiše njeno vsebino v obratnem vrstnem redu.

**Naloga 5.3** Pokažite, da lahko v zanki znotraj metode `indeksMaksimuma` števec `i` inicializiramo z vrednostjo 1 namesto z vrednostjo 0.

**Naloga 5.4** Napišite metodo, ki sprejme tabelo tipa `int[]` in vrne tabelo iste dolžine, ki vsebuje njeno kumulativno vsoto. To pomeni, da mora biti element izhodne tabele na indeksu `i` enak vsoti elementov vhodne tabele na indeksih 0, 1, ..., `i`.

**Naloga 5.5** Napišite program, ki prebere število `n` in zaporedje `n` celih števil, izpiše pa največji element in število največjih elementov. Na primer, za `n = 10` in

zaporedje  $\langle 5, 6, 8, 1, 8, 2, 8, 6, 7, 4 \rangle$  naj program izpiše 8 in 3. Ali lahko nalogo rešite brez uporabe tabel?

**Naloga 5.6** Program iz prejšnje naloge preizkusite po receptu iz razdelka 2.10. Odstranite vse spremne izpise, pripravite si množico vhodnih in pripadajočih izhodnih datotek ter ročno ali s pomočjo skripte preverite, ali program za vsak vhod proizvede izpis, ki je enak referenčnemu izhodu.

## 5.6 Uporaba tabel

### 5.6.1 Rešitev motivacijskega primera

Sedaj lahko končno rešimo motivacijski primer iz razdelka 5.1. Spomnimo se:

**Primer 5.1.** Napišimo program, ki prebere število  $n$  in zaporedje  $n$  števil, nato pa izpiše zaporedje v istem vrstnem redu.

*Rešitev.* Sedaj je jasno: izdelamo tabelo dolžine  $n$  in vanjo po vrsti prepisemo prebrana števila (prvo število v celico z indeksom 0, drugo v celico z indeksom 1 itd.), nato pa celotno tabelo izpišemo.

```
// koda 5.6 (tabele/PriklicZaporedja.java)
import java.util.Scanner;

public class PriklicZaporedja {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Koliko števil želite vnesti? ");
        int n = sc.nextInt();

        int[] stevila = new int[n];
        for (int i = 0; i < n; i++) {
            System.out.print("Vnesite število: ");
            stevila[i] = sc.nextInt();
        }

        for (int stevilo: stevila) {
            System.out.println(stevilo);
        }
    }
}
```

□

**Naloga 5.7** Napišite program, ki bere števila, dokler uporabnik ne vnese ničle, nato pa naj izpiše vsa vnesena števila (razen ničle). Lahko predpostavite, da uporabnik ne bo vnesel več kot, denimo, 100 števil. (Kako bi se naloge lotili, če tega ne bi mogli predpostaviti?)

### 5.6.2 Pogostost ocen

**Primer 5.2.** Napišimo program, ki prebere število učencev in njihove ocene (od 1 do 5) in izpiše število posameznih ocen. Na primer, pri desetih učencih in ocenah 5, 3, 5, 4, 1, 3, 2, 5, 1 in 4 naj program izpiše, da imamo dve enici, eno dvojko, dve trojki, dve štirici in tri petice.

*Rešitev.* Kako bi se lotili te naloge? Potrebujemo pet števecov: števec enic (st1), števec dvojk (st2), števec trojk (st3), števec štiric (st4) in števec petic (st5). Na začetku vse števec postavimo na 0, v zanki, v kateri beremo ocene, pa v odvisnosti od prebrane ocene povečamo ustrezni števec. Na koncu vse števec izpišemo.

```
// koda 5.7 (tabele/StatistikaOcen1.java)
import java.util.Scanner;

public class StatistikaOcen1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Vnesite število učencev: ");
        int stUcencev = sc.nextInt();

        int st1 = 0;
        int st2 = 0;
        int st3 = 0;
        int st4 = 0;
        int st5 = 0;

        for (int i = 1; i <= stUcencev; i++) {
            System.out.print("Vnesite oceno: ");
            int ocena = sc.nextInt();

            switch (ocena) {
                case 1:
                    st1++;
                    break;

                case 2:
```



```

        st2++;
        break;

    case 3:
        st3++;
        break;

    case 4:
        st4++;
        break;

    case 5:
        st5++;
        break;
    }
}

System.out.printf("Število ocen 1: %d\n", st1);
System.out.printf("Število ocen 2: %d\n", st2);
System.out.printf("Število ocen 3: %d\n", st3);
System.out.printf("Število ocen 4: %d\n", st4);
System.out.printf("Število ocen 5: %d\n", st5);
}
}

```

Program sicer deluje, a je okoren in neprilagodljiv. Če bi ga želeli prilagoditi za univerzo, kjer ocene segajo do 10, bi nam to še uspelo, kaj pa bi naredili, če bi želeli izdelati podobno statistiko po točkah na izpitu, pri katerem je število možnih točk enako 100? Bi ustvarili in vzdrževali 100 ločenih spremenljivk? Če še vedno vztrajate: kako bi dosegli, da razpon možnih ocen ali točk vnese uporabnik?

Odgovor je na dlani. Namesto petih ločenih števecv izdelamo *tabelo* petih števecv:

```
int[] stevci = new int[5];
```

Celica `stevci[0]` bo hranila število enic, celica `stevci[1]` število dvojek itd. Posebna inicializacija ni več potrebna, saj gornji stavek že sam nastavi vrednosti števecv na 0.

V prvi različici programa smo po branju ocene s stavkom `switch` izbrali in povečali ustrezni števec. Stavka `switch` ne potrebujemo več, saj sta ocena in indeks števca tesno povezana: če preberemo oceno `ocena`, moramo povečati števec z indeksom `ocena - 1`:

```
stevci[ocena - 1]++;
```

Končne vrednosti števecv lahko izpišemo s preprostim sprehodom:

```
for (int i = 0; i < stevci.length; i++) {
    System.out.printf("Število ocen %d: %d%n", i + 1, stevci[i]);
}
```

Program je sedaj bistveno bolj jedrnat, predvsem pa ga zlahka prilagodimo za univerzitetne ocene ali pa za točke na izpitu.

```
// koda 5.8 (tabele/Statistika0cen2.java)
import java.util.Scanner;

public class Statistika0cen2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Vnesite število učencev: ");
        int stUcencev = sc.nextInt();

        int[] stevci = new int[5];
        for (int i = 1; i <= stUcencev; i++) {
            System.out.print("Vnesite oceno: ");
            int ocena = sc.nextInt();
            stevci[ocena - 1]++;
        }
        for (int i = 0; i < stevci.length; i++) {
            System.out.printf("Število ocen %d: %d%n",
                              i + 1, stevci[i]);
        }
    }
}
```

□

**Naloga 5.8** Program prilagodite tako, da bo spodnjo in zgornjo mejo intervala ocen oz. točk vnesel uporabnik.

**Naloga 5.9** Napišite program, ki izpiše pogostost posameznih malih črk angleške abecede v podani vrstici besedila. Ostale znake naj program ignorira. Koda

```
Scanner sc = new Scanner(System.in);
char[] vrstica = sc.nextLine().toCharArray();
```

prebere vrstico besedila in jo prepíše v tabelo tipa `char[]`.

### 5.6.3 (Še zadnjič) praštevila

V razdelkih 3.7.3 in 4.4.1 smo napisali več programov za izpis praštevil od 1 do  $n$ . Čeprav so naši programi tekli čedalje hitreje, so vsi izvirali iz istega osnovnega postopka: sprehodi se po številih od 1 do  $n$  in za vsako preveri, ali je praštevilo. V tem razdelku pa si bomo ogledali drugačen pristop, ki (za ceno večje porabe pomnilnika) deluje še bistveno hitreje. Postopek se imenuje *Eratostenovo sito*, na listu papirja pa bi ga izvedli takole:

- Zapišemo zaporedje števil od 2 do  $n$ .
- Spremenljivko  $p$  nastavimo na 2.
- Ponavljamo, dokler je  $p \leq \sqrt{n}$ :
  - Pobrišemo vse večkratnike števila  $p$  (razen števila  $p$  samega).
  - Spremenljivko  $p$  nastavimo na prvo od preostalih števil, ki je večje od  $p$ .
- Praštevila so natanko vsa števila, ki ostanejo.

Na primer, če iščemo praštevila od 1 do 25, pričnemo s takšnim zaporedjem:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Nastavimo  $p = 2$  in pobrišemo vse večkratnike tega števila razen števila 2 samega. Dobimo

2 3 5 7 9 11 13 15 17 19 21 23 25

Spremenljivko  $p$  sedaj nastavimo na 3 in odstranimo vse njegove večkratnike. Nastane zaporedje

2 3 5 7 11 13 17 19 23 25

Vzamemo še  $p = 5$ :

2 3 5 7 11 13 17 19 23

Naslednja vrednost za  $p$  je 7, vendar pa je  $7^2 > 25$ , zato na tem mestu zaključimo. Števila, ki jih nismo pobrisali, so praštevila.

Pri tej nalogi ne gre brez tabele, saj moramo hraniti zaporedje števil. Zaporedje bi lahko predstavili s tabelo tipa `int[]`, precej učinkovitejšo rešitev pa dobimo s tabelo tipa `boolean[]`, v kateri ima element na indeksu  $i$  vrednost `true` natanko tedaj, ko je število  $i$  že pobrisano. Ker imamo  $n$  števil, mora obstajati tudi indeks  $n$ , zato bo dolžina tabele enaka  $n + 1$ .<sup>3</sup>

Eratostenovo sito lahko v delujoč program pretopimo takole:

<sup>3</sup>Lahko bi bila tudi  $n$  ali celo  $n - 1$ , toda na ta način si prihranimo kakšno sitnost.

```
// koda 5.9 (tabele/Prastevila6.java)
import java.util.Scanner;

public class Prastevila6 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        // pobrisano[i] == true <==> število i je pobrisano
        boolean[] pobrisano = new boolean[n + 1];
        int meja = (int) Math.round(Math.sqrt(n));

        int p = 2;
        while (p <= meja) {
            // pobriši števila 2 * p, 3 * p, 4 * p, ...
            for (int i = 2 * p; i <= n; i += p) {
                pobrisano[i] = true;
            }
            // poišči najmanjše nepobrisano število nad p
            do {
                p++;
            } while (p <= meja && pobrisano[p]);
        }

        // izpiši vsa nepobrisana števila (to so ravno praštevila)
        for (int i = 2; i <= n; i++) {
            if (!pobrisano[i]) {
                System.out.println(i);
            }
        }
    }
}
```

Kot smo videli v tabeli 3.8, se je naš doslej najučinkovitejši program za izpis praštevil na avtorjevem računalniku izvršil v 1,8 milisekunde pri  $n = 10^4$  in v 11 milisekundah pri  $n = 10^5$ . Kako hiter pa je program `Prastevila6`, če tako kot pri prejšnjih programih zakomentiramo vse izpise (ti so namreč počasni)? Avtor je pri  $n = 10^4$  izmeril čas 0,78 milisekunde (razmerje med časoma izvajanja obeh programov znaša 2,3), pri  $n = 10^5$  pa 4,3 milisekunde (razmerje 2,6). Pri  $n = 10^6$  je razmerje še bolj v prid Eratostenovemu situ: 130 milisekund proti 26, torej 5,0. Algoritem, star več kot dve tisočletji, še danes velja za enega od najučinkovitejših postopkov za iskanje praštevil.

**Naloga 5.10** Če upoštevamo dejstvo, da smo 2-kratnik, 3-kratnik, ...,  $(p - 1)$ -kratnik pravkar odkritega praštevila  $p$  pobrisali že v prejšnjih korakih, lahko program še nekoliko izboljšamo. Realizirajte nakazano izboljšavo ter dobljeni in prvotni program primerjajte po porabi časa.

**Naloga 5.11** Koda 5.9 je časovno učinkovita, a je zaradi potencialno velike tabele prostorsko potratna: pri  $n = 10^9$  zasede kar 1 GB pomnilnika. Kako bi porabo pomnilnika zmanjšali za faktor 8? (Namig: razdelek 3.12.)

## 5.7 Iskanje v urejeni tabeli

Napišimo metodo

```
public static int poisci(int[] t, int x)
```

ki vrne indeks elementa  $x$  v naraščajoče urejeni tabeli  $t$ , če element v tabeli obstaja, oziroma  $-1$ , če ne.

Metodo bomo napisali v treh različicah. Prva različica ne izkorišča dejstva, da je tabela urejena, zato jo je mogoče uporabiti tudi za neurejene tabele. Zamisel je enostavna: za vsak indeks od 0 do  $t.length - 1$  preverimo, ali je element tabele na tem indeksu enak  $x$ . Če je, s stavkom `return` nemudoma vrnemo trenutni indeks, sicer pa — ne naredimo ničesar in potujemo po zanki naprej. Če pridemo do konca zanke, vemo, da elementa nismo našli, zato vrnemo  $-1$ .

```
// koda 5.10 (tabele/Iskanje1.java)
public static int poisci(int[] t, int x) {
    for (int i = 0; i < t.length; i++) {
        if (t[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Druga različica. Glede na to, da je tabela naraščajoče urejena, nima smisla, da nadaljujemo z iskanjem, ko trenutni element tabele preseže iskano vrednost  $x$ . Indeks  $i$  zato povečujemo, dokler ne pridemo do konca tabele oziroma dokler je trenutni element strogo manjši od vrednosti  $x$ . Ko se zanka izteče, preverimo, ali smo se ustavili pri elementu z vrednostjo  $x$ . Če smo se, vrnemo `true`, sicer pa `false`.

```
// koda 5.11 (tabele/Iskanje2.java)
public static int poisci(int[] t, int x) {
    int i = 0;
```

```

while (i < t.length && t[i] < x) {
    i++;
}
return (i < t.length && t[i] == x) ? (i) : (-1);
}

```

Ta različica metode je od prve hitrejša samo tedaj, ko iskanega elementa *ni* v tabeli. Medtem ko prva različica prepotuje celotno tabelo ne glede na vrednost neobstoječega elementa, pri drugi v povprečju preiščemo samo polovico tabele.

Zgodbe še ni konec. Izkaže se, da lahko napišemo še *bistveno* učinkovitejšo rešitev.

Poskusimo takole. Pričnimo z elementom, ki se nahaja na sredini tabele. Imamo tri možnosti:

- Sredinski element je enak iskanemu. Imenitno, našli smo ga!
- Iskani element je večji od sredinskega. V tem primeru vemo, da se iskani element lahko nahaja le v desni polovici tabele. Elementi levo od sredinskega so namreč kvečjemu še manjši.
- Iskani element je manjši od sredinskega. V tem primeru se iskani element lahko nahaja le v levi polovici tabele.

V prvem primeru zgolj vrnemo indeks sredinskega elementa, v drugem in tretjem pa nas sicer še čaka nekaj dela, vendar pa smo njegov obseg že zmanjšali za približno polovico. Po preostali polovici tabele bi se lahko sprehodili, še bolje pa je, da naš trik ponovimo: obiščemo sredinski element *v preostali polovici* in ga primerjamo z iskanim, nato pa iskanje bodisi zaključimo (če sta elementa enaka) ali pa izločimo polovico preostanka tabele, s čimer obseg dela zmanjšamo na četrtno prvotnega. Razpolavljanje ponavljamo, dokler elementa ne najdemo oziroma dokler se dolžina podtabele, ki jo preiskujemo, ne zmanjša na 0.

Opisanemu postopku pravimo *dvojiško iskanje*. Kako ga sprogramiramo? Naj bo *aktivni* del tabele tisti del, v katerem se element še lahko nahaja. Aktivni del na začetku obsega celotno tabelo, po prvem koraku samo polovico tabele, po drugem samo četrtno itd. Naj spremenljivki *lm* (»leva meja«) in *dm* (»desna meja«) hranita indeks prvega in zadnjega elementa aktivnega dela tabele. Na začetku potemtakem postavimo *lm* na 0, *dm* pa na *t.length - 1*. Indeks sredinskega elementa izračunamo kot  $s = (lm + dm) / 2$ . Če je iskani element večji od sredinskega, postavimo *lm* na *s + 1*, če je manjši, pa *dm* na *s - 1*. Aktivni del tabele smo tako uspešno posodobili; spremenljivki *lm* in *dm* (zopet) vsebujeta indeksa njegovih meja. Postopek se izteče, ko je sredinski element enak iskanemu ali pa ko aktivni del tabele postane prazen (*lm > dm*). Če se to zgodi, vrnemo  $-1$ , saj elementa v tabeli očitno ni.

```
// koda 5.12 (tabele/Iskanje3.java)
public static int poisci(int[] t, int x) {
    int lm = 0;
    int dm = t.length - 1;

    while (lm <= dm) {
        int s = (lm + dm) / 2;
        if (t[s] == x) {
            return s;
        }
        if (x > t[s]) {
            lm = s + 1;
        } else {
            dm = s - 1;
        }
    }
    return -1;
}
```

Na sliki 5.2 je prikazan primer dvojiškega iskanja. Ni težko ugotoviti, da je ta postopek neprimerno hitrejši od navadnega (*linearne*) iskanja. Pri linearnem iskanju moramo v tabeli z  $n$  elementi v najslabšem primeru preveriti vseh  $n$  elementov, v povprečju pa  $n/2$ . Pri dvojiškem iskanju pa moramo tudi v najslabšem primeru preveriti zgolj  $\lceil \log_2(n+1) \rceil$  elementov. (Prvi korak nam dolžino aktivnega dela tabele zmanjša na  $\lfloor n/2 \rfloor$ , drugi na  $\lfloor n/4 \rfloor$  itd. Število korakov je potemtakem enako odgovoru na vprašanje, kolikokrat moramo število  $n$  celoštevilsko deliti z 2, da pridemo do ničle.) Razlika je več kot očitna: pri  $n = 1000$  imamo  $n/2 = 500$  in  $\lceil \log_2(n+1) \rceil = 10$ , pri  $n = 10^6$  pa  $n/2 = 500\,000$  in  $\lceil \log_2(n+1) \rceil = 20$ .

**Naloga 5.12** Vse tri različice iskanja preizkusite na tabeli  $\{0, 2, 4, 6, \dots, 2M\}$  za primerno veliko vrednost  $M$  (npr.  $M = 10^7$ ). Z vsakim postopkom poiščite isti nabor elementov (npr.  $\{2M/10, 18M/10, 2M/10-1, 18M/10+1\}$ ). S pomočjo metode `System.nanoTime` vsakokrat izmerite porabo časa. So dobljene meritve skladne s pričakovanji?

**Naloga 5.13** Recimo, da tabela  $f$  predstavlja monotono naraščajočo matematično funkcijo  $f$ , definirano na celoštevilskem intervalu  $[0, n-1]$  (element  $f[i]$  je enak vrednosti  $f(i)$ ). Kako bi za podani  $y$  učinkovito poiskali tak  $x$ , da velja  $f(x) = y$  (če obstaja)?

**Naloga 5.14** Algoritem dvojiškega iskanja sprogramirajte z uporabo rekurzije. Namig: rekurzivna različica metode `poisci` sprejme še dva dodatna parametra — levo in desno mejo trenutne podtabele.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
lm							s	dm						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
lm										s	dm			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
lm								s	dm					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	10	15	21	27	30	31	34	37	39	42	50	58	61	75
										lm	dm	s		

Slika 5.2 Dvojiško iskanje (iščemo število 42).

**Naloga 5.15** Vrednosti v tabeli najprej monotonno naraščajo, nato pa monotonno padajo (npr. {2, 3, 5, 8, 12, 11, 9}). Kako bi učinkovito (brez linearnega iskanja) poiskali maksimum take tabele? (Namig: bi bilo tabelo smiselno razdeliti na tri dele?)

## 5.8 Urejanje tabel

Videli smo, kako učinkovito je dvojiško iskanje, žal pa deluje samo za urejene tabele. Če torej nameravamo po podatkih pogosto iskati, se jih splača najprej urediti. To dejstvo nam je znano tudi iz vsakdanjega življenja. Telefonski imeniki<sup>4</sup> in stvarna kazala niso zaman urejeni po abecedi.

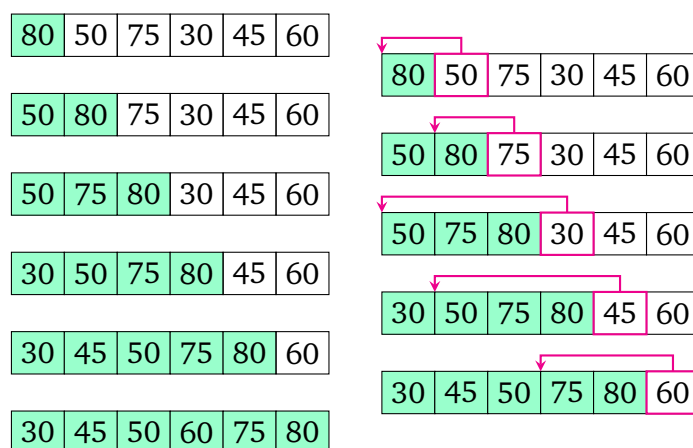
Urejanje je eden od najstarejših in največkrat rešenih računalniških problemov. Že bežno iskanje po spletu razkrije več kot dvajset različnih algoritmov. Izbrali si bomo algoritem *navadnega vstavljanja*, ki sicer ni najučinkovitejši, je pa preprost in razumljiv. Za razliko od številnih drugih postopkov je tudi *stabilen*, kar pomeni, da med seboj ne zamenjuje enakih elementov. Pri urejanju enostavnih podatkov, kot so števila in nizi, ta lastnost sicer ni pomembna, pri sestavljenih podatkih pa je drugače:

<sup>4</sup>Za mlajše generacije: to so (bile) obsežne knjige s seznamami telefonskih naročnikov ter njihovih poštnih naslovov in stacionarnih telefonskih števil. V predinternetni dobi jih je imela večina gospodinjstev.



če urejamo osebe po starosti, si pogosto želimo, da algoritem ohrani medsebojni vrstni red enako starih oseb.

Denimo, da urejamo tabelo z  $n$  elementi. Pri algoritmu navadnega vstavljanja je tabela vseskozi razdeljena na dva dela: na levi, že urejeni del in na desni, še ne urejeni del. Levi del na začetku zavzema samo prvi element tabele (en sam element je vedno urejen), desni pa vse ostale elemente. Algoritem nato izvede  $n-1$  obhodov zanke. V vsakem obhodu se osredotoči na prvi element neurejenega dela tabele in ga vstavi na ustrezno mesto v urejenem delu tabele. Urejeni del se na ta način podaljša za en element, neurejeni pa se skrči. Ko se zanka izteče, urejeni del zavzema celotno tabelo. Primer opisanega postopka je prikazan na sliki 5.3.



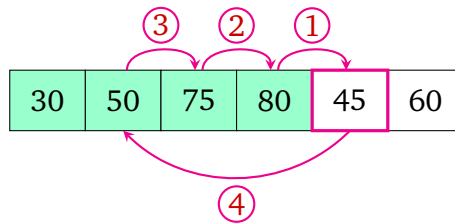
Slika 5.3 Urejanje z navadnim vstavljanjem.

Algoritem navadnega vstavljanja lahko potemtakem zapišemo takole:

```
public static void uredi(int[] t) {
    for (int i = 1; i < t.length; i++) {
        int el = t[i];
        vstavi element el v podtabelo t[0..i-1]
    }
}
```

Ostane nam še problem vstavljanja elementa  $el$  ( $t[i]$ ) v že urejeni del tabele, torej v podtabelo  $t[0..i-1]$ .<sup>5</sup> Problem lahko rešimo tako, da se z zanko pomikamo od indeksa  $i$  proti začetku tabele in pri tem sproti premikamo elemente za eno mesto v desno. Ustavimo se, ko pridemo do elementa, ki ni večji od elementa  $el$ , če takega elementa ni, pa s premikanjem prenehamo, ko prispemo do prve celice tabele. V celico, na kateri se ustavimo, vpišemo element  $el$ . Slika 5.4 prikazuje primer opisanega postopka.

<sup>5</sup>Zapis  $t[a..b]$  ni veljavna javanska koda. Uporabljamo ga zgolj za potrebe razlage.



Slika 5.4 Vstavljanje elementa na ustrezno mesto.

Sedaj smo opremljeni z vsem potrebnim za zapis metode, ki tabelo uredi z algoritmom navadnega vstavljanja:

```
// koda 5.13 (tabele/Urejanje.java)
public static void uredi(int[] t) {
    for (int i = 1; i < t.length; i++) {
        int el = t[i];
        int j = i - 1;
        while (j >= 0 && t[j] > el) {
            t[j + 1] = t[j];
            j--;
        }
        t[j + 1] = el;
    }
}
```

**Naloga 5.16** Napišite metodo, ki sprejme tabelo in njene elemente na sodih indeksih med seboj uredi naraščajoče, elemente na lihih indeksih pa padajoče. Na primer, tabelo {6, 3, 4, 8, 5, 7} naj pretvori v {4, 8, 5, 7, 6, 3}.

**Naloga 5.17** Metodo uredi dopolnite tako, da bo sprejela še parameter, ki podaja smer urejanja (naraščajoče ali padajoče). Nalogo rešite s čim manj spremembami metode uredi.

## 5.9 Memoizacija

V razdelku 4.5 smo napisali rekurzivno metodo za izračun  $n$ -tega Fibonaccijevega števila:

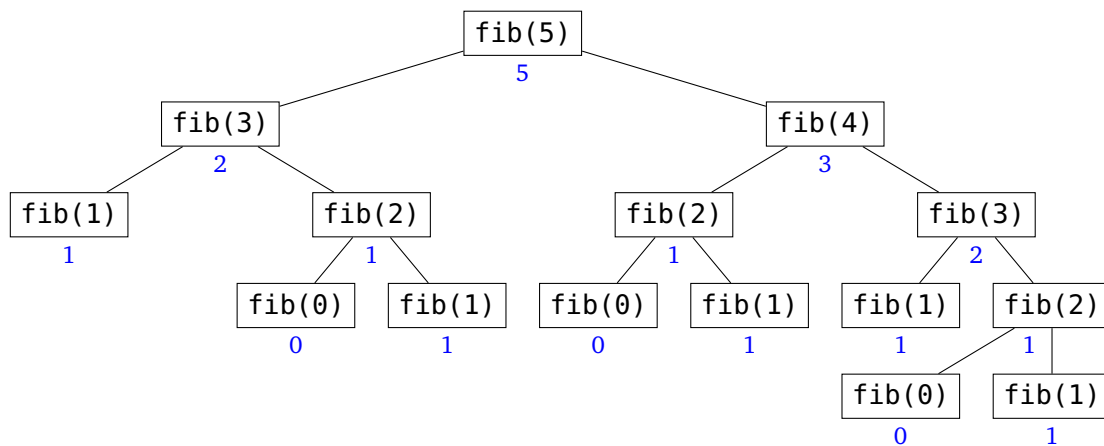
```
public static int fib(int n) {
    if (n <= 1) {
        return n;
    }
}
```

```

    }
    return fib(n - 2) + fib(n - 1);    // (1)
}

```

Metoda je sicer elegantna, a precej neučinkovita. Na avtorjevem računalniku traja izračun vrednosti  $\text{fib}(45)$  dobre štiri sekunde, vrednosti  $\text{fib}(50)$  okrog 50 sekund, vrednosti  $\text{fib}(55)$  pa skoraj 10 minut. Zakaj je metoda tako počasna? Slika 5.5 prikazuje drevo rekurzivnih klicev, ki nastane pri klicu  $\text{fib}(5)$ . Ta klic sproži klica  $\text{fib}(3)$  in  $\text{fib}(4)$ , klic  $\text{fib}(3)$  sproži klica  $\text{fib}(1)$  in  $\text{fib}(2)$ , klic  $\text{fib}(4)$  sproži klica  $\text{fib}(2)$  in  $\text{fib}(3)$  itd. Če drevo bolje pogledamo, bomo hitro videli, v katerem grmu tiči zajec. Klic  $\text{fib}(3)$ , denimo, se kar dvakrat izračuna, čeprav je rezultat v obeh klicih enak. Klic  $\text{fib}(2)$  se izračuna celo trikrat.



Slika 5.5 Drevo klicev, ki jih sproži klic  $\text{fib}(5)$ .

Večkratno izvajanje istih klicev lahko odpravimo s pomočjo tabele, v katero shranjujemo že izračunane rezultate metode  $\text{fib}$ . Pred izvedbo vrstice (1) preverimo, ali smo Fibonaccijevo število za trenutno vrednost parametra  $n$  že izračunali in shranili v tabelo. Če smo ga, shranjeno vrednost enostavno vrnemo, sicer pa vrednost izračunamo po rekurzivni formuli in jo shranimo v tabelo. Tabela mora biti dovolj velika, da lahko vanjo shranimo vrednosti do vključno  $\text{fib}(n)$ , kjer je  $n$  začetna vrednost parametra  $n$ . To pomeni, da bo tabela obsegala  $n + 1$  celic. (Lahko bi jih tudi samo  $n - 1$ . Vrednosti  $\text{fib}(0)$  in  $\text{fib}(1)$  namreč nima smisla shranjevati, saj ju ni treba računati.) Ker je  $\text{fib}(k) > 0$  za vse  $k \geq 1$ , se dogovorimo, da ničla v tabeli predstavlja še ne izračunano vrednost.

Opisani postopek — pomnjenje že izračunanih vrednosti metode — se imenuje *memoizacija* (*sic!*), sprogramiramo pa ga takole:<sup>6</sup>

<sup>6</sup>V besedi res ni črke *r*, saj izvira iz angleške besede *memo* (slov. zabeležka, zabeležiti).

```

public static int fib(int n, int[] memo) {
    if (n <= 1) {
        return n;
    }
    if (memo[n] > 0) {
        // vrednost fib(n, memo) smo že izračunali
        return memo[n];
    }
    // vrednosti fib(n, memo) še nismo izračunali,
    // zato jo izračunamo in shranimo v tabelo
    memo[n] = fib(n - 2, memo) + fib(n - 1, memo);
    return memo[n];
}

```

Preden metodo `fib` pokličemo, ji moramo podati že izdelano memoizacijsko tabelo. Ker nismo še ničesar izračunali, mora tabela vsebovati same ničle.

```

// koda 5.14 (tabele/FibonacciMemo.java)
import java.util.Scanner;

public class FibonacciMemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

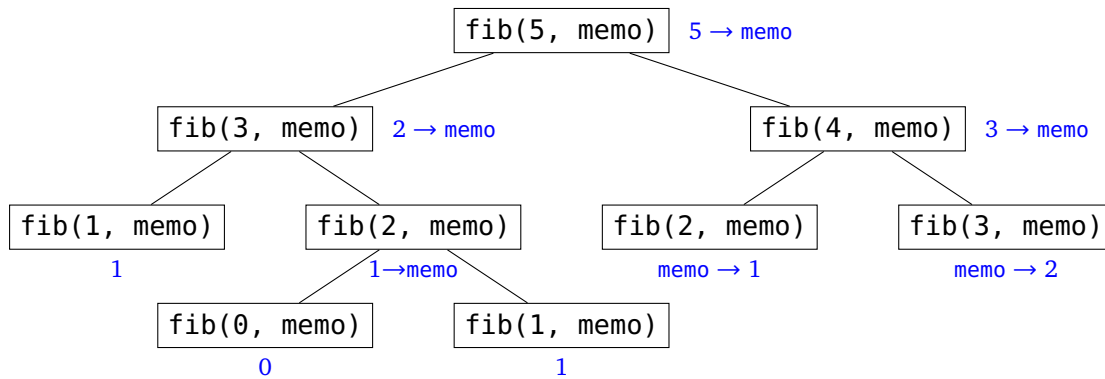
        int[] memo = new int[n + 1];
        System.out.println(fib(n, memo));
    }

    public static int fib(int n, int[] memo) {
        ...
    }
}

```

Kot prikazuje slika 5.6, je drevo za klic `fib(5, memo)` bistveno manjše od klica `fib(5)` v izhodiščni različici. Ko se klic `fib(2, memo)` drugič sproži, smo njegov rezultat že izračunali in shranili v celico `memo[2]`, zato ga takoj vrnemo. Enako velja za drugo izvedbo klica `fib(3, memo)`. Razlika bi se pri večjih vrednostih parametra  $n$  še veliko bolj poznala. Medtem ko se nam pri izhodiščni različici ustavi že pri krepko manj kot 100, bi memoizirana različica svoj rezultat izstrelila tudi pri  $n = 10^6$ , če nam zaradi prevelike globine rekurzije ne bi zmanjkalo sklada. (Ker iterativna različica nima tovrstnih težav, je pri Fibonaccijevem zaporedju dejansko boljša izbira. To pa ne pomeni, da je vedno tako. Rekurzija je močno in razširjeno programersko

orodje, le drevo rekurzivnih klicev ne sme biti previsoko.)



Slika 5.6 Drevo klicev, ki jih sproži klic `fib(5, memo)`.

**Naloga 5.18** Fibonaccijevo zaporedje lahko posplošimo tako ( $k \geq 2$  je neko fiksno število):

$$F_n^{(k)} = \begin{cases} 0 & \text{pri } n < k-1; \\ 1 & \text{pri } n = k-1; \\ F_{n-k}^{(k)} + F_{n-k+1}^{(k)} + \dots + F_{n-1}^{(k)} & \text{sicer.} \end{cases}$$

Napišite iterativno (tj. nerekurzivno) in rekurzivno metodo za izračun vrednosti  $F_n^{(k)}$ . Pri rekurzivni različici uporabite memoizacijo.

**Naloga 5.19** Napišite iterativno in rekurzivno različico metode za izračun vrednosti  $\binom{n}{k}$  pri podanih parametrih  $n \in [0, 1000]$  in  $k \in [0, n]$ . Pomagajte si z dejstvom  $\binom{n}{0} = 1$  in formulo

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

**Naloga 5.20** Koliko klicev `fib(0)` in `fib(1)` se v odvisnosti od parametra  $n$  izvede pri izhodiščni in koliko pri memoizirani različici metode `fib`?

## 5.10 Primitivni in referenčni tipi

Kaj izpiše sledeči odsek kode?

```
// koda 5.15 (tabele/KajIzpiše.java)
int[] a = {1, 2, 3};
System.out.println(Arrays.toString(a));    // (1)
```

```

int[] b = a;           // (2)
b[0] = 42;             // (3)
System.out.println(Arrays.toString(b)); // (4)
System.out.println(Arrays.toString(a)); // (5)

```

Metoda `Arrays.toString` vrne vsebino tabele  $\{e_0, e_1, \dots, e_{n-1}\}$  v obliki niza  $[e_0, e_1, \dots, e_{n-1}]$ . Vrstica (1) torej izpiše besedilo `[1, 2, 3]`. Pričakovali bi, da vrstica (4) izpiše `[42, 2, 3]`, vrstica (5) pa spet `[1, 2, 3]`. Napaka! Dejanski izpis je tak:

```

[1, 2, 3]
[42, 2, 3]
[42, 2, 3]

```

Kako je to mogoče? Vrstica (2) bi morala tabelo `a` skopirati v tabelo `b`, vrstica (3) pa spremeniti prvi element tabele `b`, kar v nobenem primeru ne bi smelo vplivati na tabelo `a` ... Preden razrešimo tole uganko, povejmo nekaj več o tipih v javi.

Java pozna dve vrsti tipov: *primitivne* in *referenčne* tipe. Primitivnih tipov je natanko osem: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` in `char`. Vsi ostali tipi (npr. `String`, `Scanner`, `int[]` ...), so referenčni. Spremenljivka primitivnega tipa vsebuje *ciljno vrednost*, torej celo število, realno število, logično vrednost ali znak (ta, kot vemo, pod kožo ni nič drugega kot celo število). Na primer, po izvedbi stavka

```
int n = 10;
```

je v spremenljivki `n` zapisano število 10 (slika 5.7, levo). Spremenljivke referenčnih tipov pa ne vsebujejo ciljnih vrednosti, ampak *kazalce* nanje. Kazalec »kaže«<sup>1</sup> izvajalniku pot do ciljne vrednosti, saj vsebuje pomnilniški naslov, na katerem se nahaja. Javanski kazalci se imenujejo tudi *reference* ali (lepše slovensko) *sklici*, a v tej knjigi bomo dosledno uporabljali izraz *kazalec*.

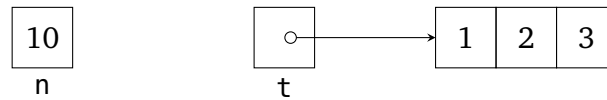
Po izvedbi stavka

```
int[] t = {1, 2, 3};
```

spremenljivka `t` torej ne vsebuje tabele z elementi 1, 2 in 3, ampak kazalec nanjo (slika 5.7, desno). Rečemo tudi, da spremenljivka `t` *kaže* na tabelo `{1, 2, 3}`. Besedna zveza »tabela `t`«, ki smo jo uporabljali doslej, torej ni točna (pravilno bi bilo reči »tabela, na katero kaže spremenljivka `t`«), a jo bomo zaradi enostavnosti še nadalje uporabljali. Vseskozi pa se bomo zavedali, da je `t` v resnici zgolj kazalec.

Navidez nenavadno obnašanje kode 5.15 lahko sedaj končno razjasnimo. Stavek

```
int[] a = {1, 2, 3};
```



**Slika 5.7** Spremenljivka primitivnega tipa (n) vsebuje ciljno vrednost, spremenljivka referenčnega tipa (t) pa kazalec nanjo.

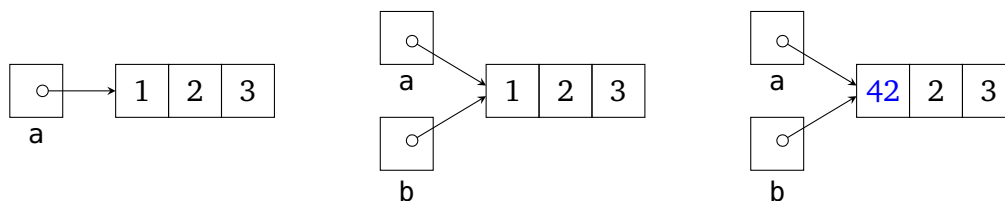
izdela tabelo z elementi 1, 2 in 3 in nanjo pokaže s spremenljivko a (slika 5.8, levo). Stavek

```
int[] b = a;
```

nato skopira vrednost spremenljivke a v spremenljivko b. Ker spremenljivka a vsebuje kazalec na tabelo z vsebino {1, 2, 3}, bo enak kazalec (tj. kazalec na isto reč) vsebovala tudi spremenljivka b. Sedaj imamo torej dva kazalca, ki kažeta na isto tabelo (slika 5.8, sredina). Stavek

```
b[0] = 42;
```

zatem vpiše vrednost 42 v prvo celico tabele, na katero kaže spremenljivka b (slika 5.8, desno). Ker spremenljivka a kaže na isto tabelo, bosta oba stavka `System.out.println(...)` izpisala niz [42, 2, 3].



**Slika 5.8** Levo: stanje po izvedbi stavka `int[] a = {1, 2, 3}`. Sredina: stanje po izvedbi stavka `int[] b = a`. Desno: stanje po izvedbi stavka `b[0] = 42`.

Spremenljivke se ne kopirajo samo pri eksplicitnem prirejanju, ampak tudi pri klicu metode. Oglejmo si nekoliko kompleksnejši primer:

```
// koda 5.16 (tabele/Kazalci.java)
import java.util.Arrays;

public class Kazalci {
    public static void main(String[] args) {
        int a = 42;
        int[] t = {1, 2, 3};
        int[] u = {4, 5, 6};    // (1)
```

```

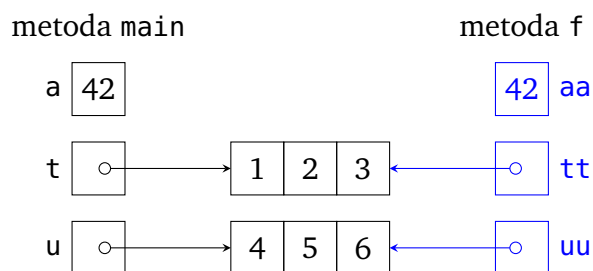
    int[] v = f(a, t, u); // (2)
    System.out.println(a);
    System.out.println(Arrays.toString(t));
    System.out.println(Arrays.toString(u));
    System.out.println(Arrays.toString(v));
}

public static int[] f(int aa, int[] tt, int[] uu) {
    aa = 66;
    tt[1] = 10;
    uu = new int[]{7, 8}; // (3)
    return uu;
}
}

```

Kaj se tukaj dogaja?

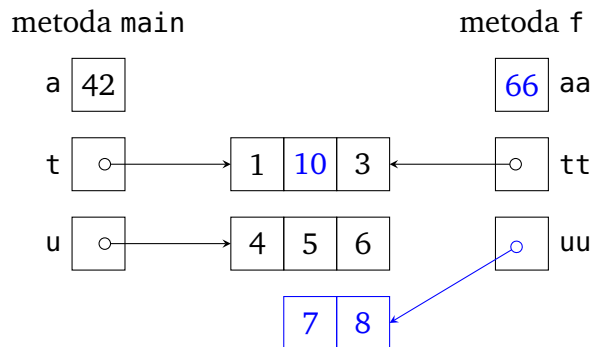
- Po izvedbi vrstice (1) obstajajo spremenljivke *a*, *t* in *u*. Spremenljivka *a* vsebuje vrednost 42, spremenljivki *t* in *u* pa kazalca na dve ločeni tabeli. Ob klicu metode *f* se argumenti *a*, *t* in *u* po vrsti skopirajo v parametre *aa*, *tt* in *uu* (slika 5.9).
- V metodi *f* dobi spremenljivka *aa* vrednost 66, drugi element tabele, na katero kaže spremenljivka *tt*, postane 10, spremenljivka *uu* pa postane kazalec na novo tabelo z elementoma 7 in 8 (slika 5.10).
- Po vrnitvi v metodo *main* se kazalec na tabelo z elementoma 7 in 8 priredi spremenljivki *v*. Spremenljivke *aa*, *tt* in *uu* seveda izginejo (slika 5.11).



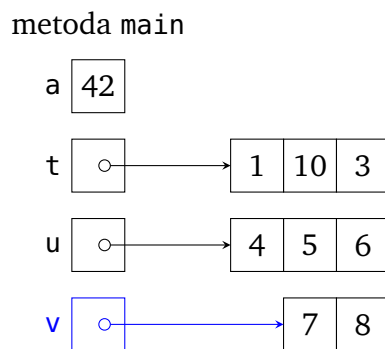
**Slika 5.9** Stanje ob klicu metode *f* v kodi 5.16.

Program potemtakem proizvede sledeči izpis:





Slika 5.10 Stanje po izvedbi vrstice (3) v kodi 5.16.



Slika 5.11 Stanje po vrnitvi iz metode f (tj. po izvedbi vrstice (2)) v kodi 5.16.

```
[1, 10, 3]
[4, 5, 6]
[7, 8]
```

**Naloga 5.21** Napišite metodo

```
public static int primerjaj(int[] a, int[] b)
```

ki vrne 1, če kazalca **a** in **b** kažeta na isto tabelo, 2, če kazalca kažeta na ločeni, a enaki tabeli (tabeli imata enako dolžino in enako zaporedje istoležnjih elementov), in 0, če ne velja nič od tega.

**Naloga 5.22** Kaj izpiše sledeči program?

```
import java.util.Arrays;

public class Telovadba {
```

```

public static void main(String[] args) {
    int[] t = {1, 1, 1};
    int[] u = t;
    int[] v = f(u);
    int[] w = g(v);
    h(v);
    System.out.println(Arrays.toString(t));
    System.out.println(Arrays.toString(u));
    System.out.println(Arrays.toString(v));
    System.out.println(Arrays.toString(w));
}

public static int[] f(int[] t) {
    t[0]++;
    return t;
}

public static int[] g(int[] t) {
    t[0]++;
    return Arrays.copyOf(t, t.length);
}

public static void h(int[] t) {
    t[0]++;
}
}

```

## 5.11 Dvodimenzionalne tabele

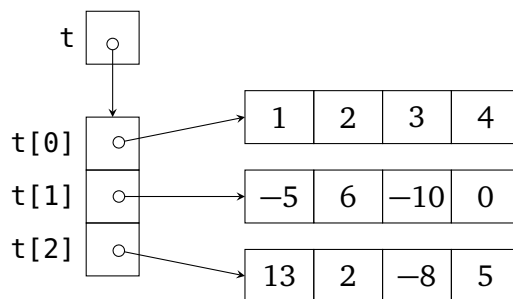
Včasih je smiselno, da podatke razporedimo v dve ali celo več dimenzij. Na primer, čeprav bi podatke o količini električne energije, ki so jo posamezna gospodinjstva porabila po posameznih mesecih preteklega leta, lahko zapisali kar v običajno enodimenzionalno tabelo, je bolj smiselno uporabiti dvodimenzionalno. Vrstice v tej tabeli bi predstavljale posamezna gospodinjstva, stolpci pa posamezne mesece. Celica v  $i$ -ti vrstici in  $j$ -tem stolpcu bi tako hranila količino električne energije, ki jo je  $i$ -to gospodinjstvo porabilo v  $j$ -tem mesecu. Brez večjih težav si lahko zamislimo tudi primer, ko bi nam prišla prav tridimenzionalna tabela. Prva dimenzija predstavlja politične stranke, druga volilne enote, tretja pa starostne skupine volilcev. Celica na koordinatah  $(i, j, k)$  tako hrani število glasov, ki ga je  $k$ -ta starostna skupina v  $j$ -ti volilni enoti namenila  $i$ -ti stranki.

Če smo natančni, so v javi vse tabele enodimenzionalne; dvo- in večdimenzionalne ne obstajajo. Kot bomo videli, pa lahko kljub temu na tak način obravnavamo tabele, ki vsebujejo kazalce na druge tabele.

### 5.11.1 Izdelava in dostop

Doslej smo se ukvarjali le s tabelami z elementi primitivnih tipov. Vendar pa lahko v javi deklariramo in izdelamo tabelo z elementi poljubnega tipa. Na primer, lahko izdelamo tabelo z elementi tipa `String`; tip take tabele je torej `String[]`. Ali pa tabelo z elementi tipa `int[]`; njen tip je potemtakem `int[][]`. Ker je `int[]` referenčni tip, tabela tipa `int[][]` ne vsebuje tabel tipa `int[]`, ampak kazalce nanje. Na primer, tabela `t` (oziroma tabela, na katero kaže kazalec `t`, če smo natančnejši) na sliki 5.12 vsebuje kazalce na tri tabele s po štirimi celoštevilskimi elementi. To tabelo izdelamo takole:

```
int[][] t = {
    {1, 2, 3, 4},
    {-5, 6, -10, 0},
    {13, 2, -8, 5}
};
```



Slika 5.12 Primer tabele tipa `int[][]`.

Ni nujno, da imajo vse tabele na drugem nivoju enako število elementov. Na primer, sledeči stavek ustvari tabelo, sestavljeno iz kazalca na tabelo dolžine 3, kazalca na tabelo dolžine 0 in kazalca na tabelo dolžine 5:

```
int[][] u = {
    {1, 2, 3},
    {},
    {4, 5, 6, 7, 8}
};
```

Tabelo kazalcev na tabele si lahko predstavljamo kot dvodimenzionalno tabelo. Naj takoj opozorimo, da ta predstava ni skladna z dejanskim stanjem, saj so v javi vse tabele enodimenzionalne. Dvodimenzionalna tabela je samo *pogled* na tabelo kazalcev na tabele. Kljub netočnosti pa bomo to predstavo pogosto uporabljali, saj nam bo poenostavila terminologijo. Na primer, tabelo *t* s slike 5.12 si lahko predstavljamo kot tabelo s tremi vrsticami in štirimi stolpci (slika 5.13), pri čemer pojem *vrstica* označuje posamezno tabelo na drugem nivoju, pojem *stolpec* pa zaporedje istoležnih elementov v vrsticah (prvi elementi vseh vrstic tvorijo prvi stolpec, drugi elementi tvorijo drugi stolpec itd.). Izraz *stolpec* ima smisel le pri pravokotnih tabelah, pa še tam se moramo zavedati, da gre zgolj za miselni konstrukt.

	0	1	2	3
0	1	2	3	4
1	-5	6	-10	0
2	13	2	-8	5

**Slika 5.13** Enostavnejši, a netočen pogled na tabelo tipa `int[][]`.

Z izrazom `t[i][j]` dostopamo do elementa z indeksom *j* v vrstici z indeksom *i*. Na primer, vrednost izraza `t[1][2]` je `-10`. Z izrazom `t[i]` dostopamo do vrstice z indeksom *i* (vrednost tega izraza je kazalec na vrstico), do posameznih stolpcev pa — to nas ne bi smelo presenetiti — ne moremo dostopati. Na primer, stavek

```
System.out.println(Arrays.toString(t[2]));
```

izpiše niz `[13, 2, -8, 5]`.

Pravokotno dvodimenzionalno tabelo z elementi tipa *T* lahko izdelamo tudi takole:

```
T[][] t = new T[m][n];
```

Tabela *t* je sestavljena iz *m* kazalcev na tabele s po *n* elementi (oziroma iz *m* vrstic in *n* stolpcev v alternativnem pogledu). Vsi elementi posameznih vrstic imajo privzeto vrednost za tip *T* (npr. 0 za tip `int`).

Stavek

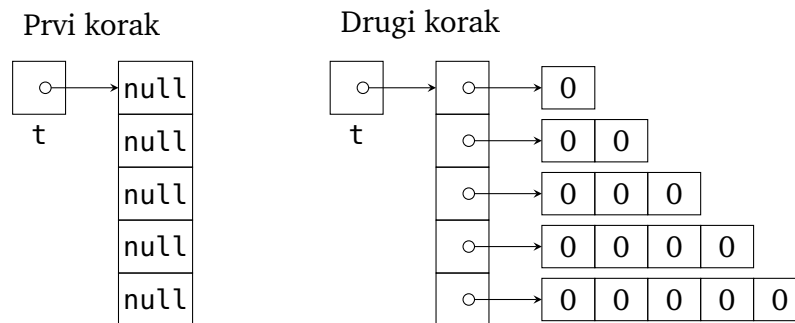
```
T[][] t = new T[m][];
```

izdela tabelo z *m* kazalci, od katerih bo vsak lahko kazal na poljubno dolgo tabelo tipa *T*[]. V tem trenutku pa ti kazalci vsebujejo privzeto vrednost za referenčne tipe — vrednost `null` (tabela 5.1). Kazalec z vrednostjo `null` ne kaže nikamor.

Gornji stavek lahko uporabimo za izdelavo nepravokotnih tabel. Na primer, sledeča koda izdela tabelo z petimi vrsticami, pri čemer je dolžina vrstice z indeksom  $i$  enaka  $i + 1$  (slika 5.14):

```
// koda 5.17 (tabele/Stopnice.java)
int[][] t = new int[5][];
for (int i = 0; i < t.length; i++) {
    t[i] = new int[i + 1];
}
```

Izraz `t.length` vrne dolžino tabele `t`, torej število vrstic.



Slika 5.14 Dvostopenjska izdelava dvodimenzionalne tabele.

### Naloga 5.23 Napišite metodo

```
public static int[][] pascalovTrikotnik(int n)
```

ki izdela in vrne dvodimenzionalno tabelo z  $n + 1$  vrsticami, pri čemer  $i$ -ta vrstica odraža  $i$ -to vrstico Pascalovega trikotnika. Vrstica z indeksom 0 naj bo torej tabela {1}, vrstica z indeksom 1 naj bo tabela {1, 1} itd.

**Naloga 5.24** Kakšna je po izvedbi sledečih stavkov vsebina tabel, na katere kažejo kazalci `a`, `b`, `c` oziroma `t`?

```
int[] a = {1, 2};
int[] b = a;
int[] c = {1, 2};
int[][] t = {a, b, c, {1, 2}, a};
a[0]--;
b[1]++;
t[2][0] -= 2;
t[3][1] *= 3;
```

```
t[4] = t[2];
```

### 5.11.2 Sprehod

Za sprehod po dvodimenzionalni tabeli potrebujemo dvojno zanko: z zunanjo se sprehodimo po indeksih vrstic (od 0 do `t.length - 1`), z notranjo pa po elementih posameznih vrstic (od 0 do `t[i].length - 1`). Sledeča metoda po vrsticah izpiše vsebino dvodimenzionalne tabele:

```
// koda 5.18 (tabele/Izpis2D.java)
public static void izpisiPoIndeksih(int[][] t) {
    for (int i = 0; i < t.length; i++) {
        for (int j = 0; j < t[i].length; j++) {
            System.out.print(t[i][j] + " ");
        }
        System.out.println();
    }
}
```

Po dvodimenzionalni tabeli se lahko sprehodimo tudi z dvojno zanko *for-each*. Zunanja zanka potuje po vrsticah, notranja pa po elementih vrstice. Pri tabeli tipa `T[][]` je tip posamezne vrstice enak `T[]`, tip posameznega elementa vrstice pa `T`, zato je glava zunanje zanke enaka

```
for (T[] vrstica: t)
```

glava notranje pa

```
for (T element: vrstica)
```

Na primer, naslednja metoda izpiše vsebino tabele tipa `int[][]`:

```
// koda 5.19 (tabele/Izpis2D.java)
public static void izpisiPoElementih(int[][] t) {
    for (int[] vrstica: t) {
        for (int element: vrstica) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Sedaj pa napišimo metodo

```
public static int[] vsotaPoVrsticah(int[][] t)
```

ki izdelava in vrne tabelo s toliko elementi, kot je vrstic tabele  $t$ , pri čemer  $i$ -ti element izhodne tabele podaja vsoto  $i$ -te vrstice tabele  $t$ . Slika 5.15 prikazuje primer vhodne in pripadajoče izhodne tabele.

	0	1	2	3			
0	1	2	3	4	⇒	0	10
1	-5	6	-10	0		1	-9
2	13	2	-8	5		2	12

Slika 5.15 Vsote posameznih vrstic dvodimenzionalne tabele.

Nalogo uženemo takole: izdelamo tabelo rezultat ustrezne velikosti, nato pa se sprehodimo po vrsticah tabele  $t$  in za vsako vrstico izračunamo njeno vsoto. Vsoto  $i$ -te vrstice shranimo v  $i$ -to celico tabele rezultat. To tabelo (pravilneje: kazalec nanjo) na koncu vrnemo.

```
// koda 5.20 (tabele/VsotaPoVrsticah.java)
public static int[] vsotaPoVrsticah(int[][] t) {
    int[] rezultat = new int[t.length];


    for (int i = 0; i < t.length; i++) {
        // izračunaj vsoto vrstice z indeksom i
        // (vsoto tabele t[i]) ...
        int vsota = 0;
        for (int j = 0; j < t[i].length; j++) {
            vsota += t[i][j];
        }
        // ... in jo shrani v celico rezultat[i]
        rezultat[i] = vsota;
    }
    return rezultat;
}
```

Napišimo še metodo, ki na podlagi podane dvodimenzionalne tabele  $t$  izdelava in vrne tabelo s toliko elementi, kot je stolpcev tabele  $t$ , pri čemer  $i$ -ta celica izhodne tabele vsebuje indeks največjega elementa v  $i$ -tem stolpcu (slika 5.16). Predpostavili bomo, da imajo vse vrstice enako število elementov (sicer nima smisla govoriti o stolpcih) in da tabela  $t$  vsebuje vsaj eno vrstico.

Ponovno si pripravimo tabelo rezultat. Ob predpostavkah, da ima tabela  $t$  vsaj eno vrstico in da so vse vrstice enako dolge, lahko število stolpcev zapišemo

	0	1	2	3
0	1	2	3	4
1	-5	6	-10	0
2	13	2	-8	5



	0	1	2	3
	2	1	0	2

**Slika 5.16** Indeksi maksimumov v posameznih stolpcih dvodimenzionalne tabele.

kot `t[0].length`; takšna bo torej dolžina tabele rezultat. Nato se sprehodimo po stolpcih, pri čemer za vsak stolpec poiščemo indeks največjega elementa in ga vpišemo v pripadajočo celico tabele rezultat. Upoštevamo, da je stolpec z indeksom `j` sestavljen iz elementov `t[0][j]`, `t[1][j]`, ..., `t[t.length - 1][j]`.

```
// koda 5.21 (tabele/ImaxPoStolpcih.java)
public static int[] imaxPoStolpcih(int[][] t) {
    int stStolpcev = t[0].length;
    int[] rezultat = new int[stStolpcev];

    for (int j = 0; j < stStolpcev; j++) {
        // izračunaj indeks maksimuma v stolpcu z indeksom j ...
        int iMax = 0;
        for (int i = 1; i < t.length; i++) {
            if (t[i][j] > t[iMax][j]) {
                iMax = i;
            }
        }
        // ... in ga shrani v rezultat[j]
        rezultat[j] = iMax;
    }
    return rezultat;
}
```

#### Naloga 5.25 Napišite metodo

```
public static int[] globalniMaksimum(int[][] t)
```



ki vrne tabelo  $\{i^*, j^*\}$ , pri čemer sta  $i^*$  in  $j^*$  indeksa vrstice in stolpca največjega elementa tabele  $t$ .

**Naloga 5.26** Napišite program, ki prebere število kolesarjev ( $n$ ), število etap ( $k$ ) in rezultate posameznih etap ( $n$  vrstic s po  $k$  števili, ki predstavljajo dosežke (čase v sekundah) posameznih kolesarjev v posameznih etapah), izpiše pa lestvico kolesarjev za vsako posamezno etapo in za celotno dirko.

**Naloga 5.27** Napišite metodo

```
public static int[][] zmnozi(int[][] a, int[][] b)
```

ki vrne produkt matrik, predstavljenih s tabelama  $a$  (velikosti  $m \times n$ ) in  $b$  (velikosti  $n \times k$ ). Rezultat je matrika velikosti  $m \times k$ . (Če je matrika  $C$  produkt matrik  $A$  in  $B$ , potem njene elemente računamo po formuli  $c_{ij} = \sum_{r=1}^n a_{ir} b_{rj}$ .)

### 5.11.3 Najcenejše poti v grafu

Ob besedi *graf* morda pomislite na krivuljo, ki ponazarja potek matematične funkcije, v računalništvu pa ima ta pojem običajno drugačen pomen. Graf je množica *vozlišč* in *povezav* med njimi. Z grafi lahko potemtakem predstavimo najrazličnejša omrežja. Na primer, vozlišča so lahko kraji, povezave pa cestni odseki med njimi. V socialnih omrežjih so vozlišča osebe, povezave pa njihova medsebojna prijateljstva.

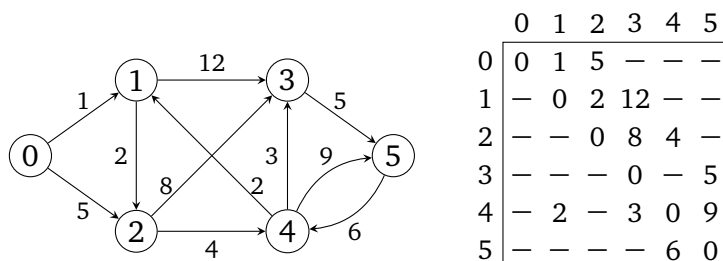
Predpostavili bomo, da ima graf  $n$  vozlišč, označenih z indeksi od 0 do  $n - 1$ . Povezave naj bodo usmerjene (vsaka povezava ima svoje izvirno in ciljno vozlišče). Za vsak par vozlišč  $i$  in  $j$  ( $i \neq j$ ) naj obstaja kvečjemu ena povezava  $i \rightarrow j$ , povezav  $i \rightarrow i$  pa naj ne bo. Povezave naj bodo označene z nenegativnimi celimi števili, ki jim bomo rekli *cene*, v praksi pa lahko predstavljajo tudi kaj drugega. Na primer, pri cestnem omrežju lahko cena povezave med dvema vozliščema predstavlja ceno prevoza, lahko pa gre preprosto za cestno razdaljo med pripadajočima krajema.

Graf z opisanimi lastnostmi lahko predstavimo s celoštevilsko kvadratno tabelo velikosti  $n \times n$ , v kateri je element na koordinatah  $(i, j)$  (vrstica z indeksom  $i$ , stolpec z indeksom  $j$ ) enak

- 0, če je  $i = j$ ;
- $-1$ , če je  $i \neq j$  in povezava  $i \rightarrow j$  ne obstaja;<sup>7</sup>
- cena povezave  $i \rightarrow j$ , če povezava  $i \rightarrow j$  obstaja.

<sup>7</sup>Če bi bile cene lahko negativne, bi morali najti nek drug način za predstavitev neobstoječe povezave. Če ne drugače, bi si lahko pomagali z dodatno tabelo tipa `boolean[][]`, ki za vsak par vozlišč  $i$  in  $j$  pove, ali obstaja povezava  $i \rightarrow j$ .

Takšni tabeli pravimo *matrika sosednosti*.<sup>8</sup> Primer grafa in pripadajoče matrike sosednosti je prikazan na sliki 5.17.



**Slika 5.17** Graf in njegova matrika sosednosti (vrednost  $-1$  je zaradi preglednosti zapisana kot  $-$ ).

Zaporedje vozlišč  $v_1, v_2, \dots, v_n$ , tako da za vsak  $i \in \{1, \dots, n-1\}$  obstaja povezava  $v_i \rightarrow v_{i+1}$ , imenujemo *pot*. Naj bo *cena poti* enaka vsoti cen povezav na poti. V primeru na sliki 5.17 je cena poti  $1 \rightarrow 2 \rightarrow 3$  enaka 10. Od vozlišča 1 do vozlišča 3 je sicer mogoče priti tudi ceneje. Pot  $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$  s ceno 9 je najcenejša med vsemi potmi med tema dvema vozliščema.

Napišimo metodo

```
public static int[][] najcenejsePoti(int[][] graf)
```

ki izdelava in vrne tabelo velikosti  $n \times n$ , v kateri element na koordinatah  $(i, j)$  (pri  $i \neq j$ ) hrani ceno najcenejše poti med vozliščema  $i$  in  $j$ . Elementi na diagonali naj bodo enaki 0, element z vrednostjo  $-1$  pa naj pove, da med pripadajočima vozliščema ni nobene poti. Za primer na sliki 5.17 je matrika najcenejših poti prikazana kot spodnja desna tabela na sliki 5.18.

Dogovorimo se, da zapis  $i \overset{k}{\rightsquigarrow} j$  predstavlja najcenejšo pot od vozlišča  $i$  do vozlišča  $j$ , pri kateri indeks nobenega vmesnega vozlišča ne presega  $k$ . Pri grafu s slike 5.17 se zapisa  $1 \overset{0}{\rightsquigarrow} 3$  in  $1 \overset{1}{\rightsquigarrow} 3$  nanašata na pot  $1 \rightarrow 3$ , zapisa  $1 \overset{2}{\rightsquigarrow} 3$  in  $1 \overset{3}{\rightsquigarrow} 3$  na pot  $1 \rightarrow 2 \rightarrow 3$ , zapisa  $1 \overset{4}{\rightsquigarrow} 3$  in  $1 \overset{5}{\rightsquigarrow} 3$  pa na pot  $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$ .

Matriko najcenejših poti bomo izdelali kot kopijo matrike sosednosti, nato pa jo bomo postopoma »izboljševali«. V prvi iteraciji bomo za vsak par vozlišč  $i$  in  $j$  preverili, ali se nam pri potovanju od  $i$  do  $j$  splača narediti ovinek prek vozlišča 0. Če je pot  $i \rightarrow 0 \rightarrow j$  cenejša od neposredne poti  $i \rightarrow j$ , bomo v element na koordinatah  $(i, j)$  vpisali ceno poti  $i \rightarrow 0 \rightarrow j$ , sicer pa elementa ne bomo spremenili. (Če neposredna pot med vozliščema ne obstaja, si predstavljajmo, kot da obstaja,

<sup>8</sup>Pravokotni dvodimenzionalni tabeli se pogosto reče *matrika*, čeprav, če smo pikolovski, pojma nista povsem enakovredna: matrika je matematični koncept, tabela pa programerski.

	0	1	2	3	4	5
0	0	1	5	—	—	—
1	—	0	2	12	—	—
2	—	—	0	8	4	—
3	—	—	—	0	—	5
4	—	2	—	3	0	9
5	—	—	—	—	6	0

$k=0$

	0	1	2	3	4	5
0	0	1	3	13	—	—
1	—	0	2	12	—	—
2	—	—	0	8	4	—
3	—	—	—	0	—	5
4	—	2	4	3	0	9
5	—	—	—	—	6	0

$k=1$

	0	1	2	3	4	5
0	0	1	3	11	7	—
1	—	0	2	10	6	—
2	—	—	0	8	4	—
3	—	—	—	0	—	5
4	—	2	4	3	0	9
5	—	—	—	—	6	0

$k=2$

	0	1	2	3	4	5
0	0	1	3	11	7	16
1	—	0	2	10	6	15
2	—	—	0	8	4	13
3	—	—	—	0	—	5
4	—	2	4	3	0	8
5	—	—	—	—	6	0

$k=3$

	0	1	2	3	4	5
0	0	1	3	10	7	15
1	—	0	2	9	6	14
2	—	6	0	7	4	12
3	—	—	—	0	—	5
4	—	2	4	3	0	8
5	—	8	10	9	6	0

$k=4$

	0	1	2	3	4	5
0	0	1	3	10	7	15
1	—	0	2	9	6	14
2	—	6	0	7	4	12
3	—	13	15	0	11	5
4	—	2	4	3	0	8
5	—	8	10	9	6	0

$k=5$

**Slika 5.18** Posodabljanje matrike najcenejših poti s Floyd-Warshallovim algoritmom za primer s slike 5.17.

vendar pa ima neskončno ceno.) Po zaključku prve iteracije bo matrika torej hranila cene poti  $i \overset{0}{\rightsquigarrow} j$ .

V drugi iteraciji bomo upoštevali tudi ovinke prek vozlišča 1: če je pot  $i \overset{0}{\rightsquigarrow} 1 \overset{0}{\rightsquigarrow} j$  cenejša od poti  $i \overset{0}{\rightsquigarrow} j$ , spremenimo element matrike s koordinatama  $(i, j)$ , sicer pa ga pustimo pri miru. V tretji iteraciji upoštevamo tudi ovinke prek vozlišča 2 itd. Po iteraciji z indeksom  $k$  bo element na koordinatah  $(i, j)$  potemtako hranil ceno poti  $i \overset{k}{\rightsquigarrow} j$ .

Ni težko ugotoviti, da moramo izvesti  $n$  iteracij. Tedaj bo namreč vsak element matrike vseboval ceno najcenejše poti med pripadajočima vozliščema, pri kateri indeks nobenega vmesnega vozlišča ni večji od  $n - 1$ . Ta pogoj pa pri grafu z vozlišči  $0, 1, \dots, n - 1$  izpolnjujejo vse poti.

Opisani postopek imenujemo *Floyd-Warshallov algoritem* (Cormen in sod., 2009), sprogramiramo pa ga s trojno zanko. Za vsako iteracijo (zunanja zanka) posodobimo najcenejšo pot za vsak par medsebojno različnih vozlišč (srednja in notranja zanka):

```
// koda 5.22 (tabele/NajcenejsePoti.java)
public static int[][] najcenejsePoti(int[][] graf) {
    // matriko najcenejših poti izdelamo kot kopijo matrike
    // sosednosti
```

```

int stVozlisc = graf.length;
int[][] cene = new int[stVozlisc][stVozlisc];
for (int i = 0; i < stVozlisc; i++) {
    for (int j = 0; j < stVozlisc; j++) {
        cene[i][j] = graf[i][j];
    }
}

for (int k = 0; k < stVozlisc; k++) {
    for (int i = 0; i < stVozlisc; i++) {
        for (int j = 0; j < stVozlisc; j++) {
            if (i != j) {
                // posodobimo ceno poti od i do j
                cene[i][j] = izberi(cene[i][j], cene[i][k],
                                    cene[k][j]);
            }
        }
    }
}
return cene;
}

```

Metoda *izberi* primerja trenutno znano najcenejšo pot med vozliščema  $i$  in  $j$  ( $i \rightsquigarrow^{k-1} j$ ) s potjo  $i \rightsquigarrow^{k-1} k \rightsquigarrow^{k-1} j$  in vrne ceno cenejše izmed obeh poti.

```

public static int izberi(int trenutna, int odsek1, int odsek2) {
    if (odsek1 < 0 || odsek2 < 0) {
        return trenutna;
    }
    if (trenutna < 0) {
        return odsek1 + odsek2;
    }
    return Math.min(trenutna, odsek1 + odsek2);
}

```

Slika 5.17 prikazuje potek algoritma za graf s slike 5.17. Za primer si oglejmo, kako se spreminja element na koordinatah (0, 3). Ker vozlišči 0 in 3 nista povezani, je začetna vrednost elementa enaka  $-1$ . Po prvi iteraciji ( $k = 0$ ) je vrednost še vedno  $-1$ , saj pot  $0 \rightsquigarrow^0 3$  ne obstaja. Po drugi iteraciji ( $k = 1$ ) upoštevamo poti prek vozlišč 0 in 1. Najcenejša taka pot med vozliščema 0 in 3 je  $0 \rightarrow 1 \rightarrow 3$ , njena cena pa je enaka 13. V naslednji iteraciji ( $k = 2$ ) najdemo pot  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$  s ceno 11. Pri  $k = 3$  ni nobene spremembe, pri  $k = 4$  pa najdemo še cenejšo pot:

$0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 3$ . Njena cena znaša 10. V zadnji iteraciji ( $k = 5$ ) ne najdemo nobene cenejše poti, zato lahko zaključimo, da ima najcenejša pot med vozliščema 0 in 3 ceno 10.

**Naloga 5.28** Metodo `najcenejsePoti` vključite v program, ki graf prebere s standardnega vhoda, na standardni izhod pa izpiše matriko najcenejših poti. V prvi vrstici vhoda naj bo zapisano število vozlišč ( $n$ ) in število povezav ( $m$ ), v vsaki od naslednjih  $m$  vrstic pa naj bodo podana števila  $i$ ,  $j$  in  $c$ , ki povedo, da je cena povezave  $i \rightarrow j$  enaka  $c$ .

**Naloga 5.29** Naj bo graf podan s tabelo velikosti  $n \times n$ , v kateri ima element na koordinatah  $(i, j)$  vrednost `true` natanko tedaj, ko obstaja povezava  $i \rightarrow j$ . Napišite metodo

```
public static boolean[][] dosegljivost(boolean[][] graf)
```

ki za podani graf vrne tabelo velikosti  $n \times n$ , v kateri ima element na koordinatah  $(i, j)$  vrednost `true` natanko tedaj, ko obstaja pot od vozlišča  $i$  do vozlišča  $j$ . Namig: ne izumljajte svojega algoritma, pač pa prilagodite Floyd-Warshallovega.

**Naloga 5.30** Napišite metodo

```
public static int[] najcenejseOdVozlisca(int[][] graf, int vozl)
```

ki vrne tabelo dolžine  $n$ , v kateri element na indeksu  $i$  hrani ceno najcenejše poti od vozlišča `vozl` do vozlišča  $i$ .

Naloge bi se sicer lahko lotili s Floyd-Warshallovim algoritmom, vendar pa je problem iskanja najcenejših poti od fiksnega vozlišča (recimo mu  $v$ ) do vseh ostalih mogoče bistveno učinkoviteje rešiti z *Dijkstrovim* algoritmom. Algoritem vzdržuje celoštevilsko tabelo z  $n$  elementi, v kateri element na indeksu  $i$  podaja trenutno znano ceno najcenejše poti od vozlišča  $v$  do vozlišča  $i$ . Na začetku je element te tabele na indeksu  $v$  enak 0, vsi ostali pa so enaki  $-1$  (ta vrednost dejansko predstavlja neskončno ceno). Nato v vsaki iteraciji poiščemo vozlišče  $w$ , ki ima med vsemi neobravnavanimi vozlišči najnižjo ceno poti od vozlišča  $v$  (v prvi iteraciji bo to kar vozlišče  $v$ ), in v tabeli posodobimo cene poti od vozlišča  $v$  do vseh sosedov vozlišča  $w$  (preverimo, ali je vsota cene poti od vozlišča  $v$  do vozlišča  $w$  in cene povezave od vozlišča  $w$  do njegovega sosedu  $s$  manjša od doslej znane cene poti od vozlišča  $v$  do vozlišča  $s$ ). Po  $n$  iteracijah tabela vsebuje dejanske cene najcenejših poti od vozlišča  $v$  do vseh ostalih.

Na primer, za graf s slike 5.17 in vozlišče  $v = 0$  bi algoritem tekel tako, kot prikazuje tabela 5.2. Z zvezdico so označena vozlišča, ki smo jih že obravnavali.

**Tabela 5.2** Potek Dijkstrovega algoritma za graf s slike 5.17 in izvorno vozlišče 0.

Izbrano vozlišče	Cena poti od 0 do ...					
	0	1	2	3	4	5
0	0	—	—	—	—	—
0	0*	1	5	—	—	—
1	0*	1*	3	13	—	—
2	0*	1*	3*	11	7	—
4	0*	1*	3*	10	7*	16
3	0*	1*	3*	10*	7*	15
5	0*	1*	3*	10*	7*	15*

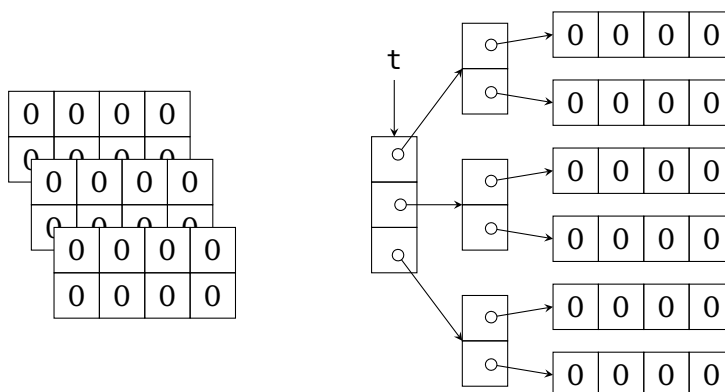
## 5.12 Tridimenzionalne tabele

Tri- in večdimenzionalne tabele pridejo v poštev razmeroma redko, kljub temu pa je prav, da jih znamo uporabljati. V tem razdelku se bomo omejili samo na tridimenzionalne tabele.

Oglejmo si konkreten primer. Stavek

```
int[][][] t = new int[3][2][4];
```

izdela tabelo, ki si jo lahko predstavljamo kot kvader velikosti  $3 \times 2 \times 4$  ali (enostavneje) kot skupek treh dvodimenzionalnih tabel (»strani«) velikosti  $2 \times 4$  (slika 5.19, levo), v resnici pa gre za tabelo treh kazalcev, od katerih vsak kaže na tabelo dveh kazalcev, od katerih vsak kaže na tabelo štirih celoštevilskih elementov (slika 5.19, desno). Vsi celoštevilski elementi so enaki 0.

**Slika 5.19** Tridimenzionalna tabela: poenostavljen pogled (levo) in pravilen pogled (desno).

Z izrazom  $t[i]$  dostopamo do strani z indeksom  $i$ ,  $t[i][j]$  je kazalec na vrstico z indeksom  $j$  na tej strani,  $t[i][j][k]$  pa je element z indeksom  $k$  v tej vrstici. Z izrazom  $t.length$  dobimo število strani tabele.

Tabelo lahko inicializiramo tudi z izrecno podanimi začetnimi vrednostmi. Na primer, v sledeči tabeli ima element  $t[i][j][k]$  vrednost  $(i+1)(j+1)(k+1)$ :

```
int[][][] t = {
    {
        {1, 2, 3, 4},
        {2, 4, 6, 8}
    },
    {
        {2, 4, 6, 8},
        {4, 8, 12, 16}
    },
    {
        {3, 6, 9, 12},
        {6, 12, 18, 24}
    }
};
```

V splošnem seveda ni nujno, da imajo vse strani enako število vrstic, prav tako pa ni nujno, da imajo vse vrstice enako število elementov.

**Primer 5.3.** Denimo, da si na neki srednji šoli izberemo določeno množico dijakov in predmetov ter beležimo ocene, s katerimi so izbrani dijaki zaključili izbrane predmete v posameznih letnikih. Rezultate zberemo v tridimenzionalno tabelo ocena: element na koordinatah  $(d, l, p)$  podaja oceno dijaka  $d$  v letniku  $l$  pri predmetu  $p$  (vse troje štejemo od 0 naprej). Napišimo metodo

```
public static int[][] petkarji(int[][][] ocena)
```

ki vrne tabelo, v kateri element na koordinatah  $(p, l)$  podaja število dijakov, ki so predmet  $p$  v letniku  $l$  zaključili z oceno 5.

Predpostavili bomo, da je tabela ocena regularna, torej da ima vsaka stran enako število vrstic, vsaka vrstica pa enako število elementov. Prav tako bomo predpostavili, da nobena dimenzija ni dolga 0.

*Rešitev.* Izdelati moramo tabelo velikosti  $P \times L$ , kjer je  $P$  število izbranih predmetov,  $L$  pa število letnikov. Ker je tabela ocena regularna, lahko število letnikov dobimo preprosto kot `ocena[0].length`, število predmetov pa kot `ocena[0][0].length`:

```
// koda 5.23 (tabele/Petkarji.java)
public static int[][] petkarji(int[][][] ocena) {
```

```

    int stDijakov = ocena.length;
    int stLetnikov = ocena[0].length;
    int stPredmetov = ocena[0][0].length;
    int[][] rezultat = new int[stPredmetov][stLetnikov];
    ...
}

```

Sedaj moramo za vsak par predmeta  $p$  in letnika  $l$  izračunati število dijakov, ki so predmet  $p$  v letniku  $l$  zaključili s petico. To število pridobimo tako, da se sprehodimo po elementih  $ocena[0][l][p]$ ,  $ocena[1][l][p]$ , ...,  $ocena[D-1][l][p]$  (kjer je  $D$  število dijakov) in preštejemo vrednosti 5. Dobljeni rezultat shranimo v izhodno tabelo na koordinati  $(p, l)$ .

```

public static int[][] petkarji(int[][][] ocena) {
    ...
    for (int p = 0; p < stPredmetov; p++) {
        for (int l = 0; l < stLetnikov; l++) {
            int stPetkarjev = 0;
            for (int d = 0; d < stDijakov; d++) {
                if (ocena[d][l][p] == 5) {
                    stPetkarjev++;
                }
            }
            rezultat[p][l] = stPetkarjev;
        }
    }
    return rezultat;
}

```

□

### Naloga 5.31 Napišite metodo

```
public static int najlazjiPredmet(int[][][] ocena)
```

ki vrne indeks predmeta z največjo povprečno oceno (po vseh letnikih skupaj).

### Naloga 5.32 Napišite metodo

```
public static boolean[][] konstantnezi(int[][][] ocena)
```

ki vrne tabelo, v kateri je element na koordinatah  $(p, d)$  enak `true` natanko v primeru, če je dijak  $d$  predmet  $p$  v vseh letnikih zaključil z enako oceno.



**Naloga 5.33** Naj element `temperatura[l][m][d]` podaja maksimalno dnevno temperaturo v letu  $l$ , mesecu  $m$  in dnevu  $d$  (vse troje štejmo od 0 naprej). Napišite metodo

```
public static int[] najtoplejseLetoZaMesec(int[][][] temperatura)
```

ki vrne tabelo, v kateri element  $m$  podaja leto, v katerem je bila povprečna temperatura meseca  $m$  najvišja. (Recimo: januar je bil najtoplejši leta 3, februar leta 0 itd.) Lahko predpostavite, da nobeno leto ni prestopno.

## 5.13 Povzetek

- Tabela je zaporedje spremenljivk istega tipa. Do posameznih celic (tj. spremenljivk) oziroma elementov (tj. vrednosti spremenljivk) tabele dostopamo prek indeksov. Prvi element ima indeks 0, drugi 1 itd. Vsak element posebej lahko (neodvisno od drugih) preberemo in vsakega posebej lahko spremenimo.
- Dolžino tabele (število njenih elementov) podamo ob izdelavi tabele in je kasneje ne moremo več spreminjati.
- Po elementih tabele se lahko sprehodimo bodisi posredno (z običajno zanko *for*, ki teče po indeksih tabele) bodisi neposredno (z zanko *for-each*).
- Ena od tipičnih operacij na tabeli je iskanje elementa. Če je tabela urejena, lahko uporabimo učinkovito dvojiško iskanje.
- Obstajajo številni algoritmi za urejanje tabel. Algoritem navadnega vstavljanja v vsaki iteraciji izbere prvi element v desnem, še neurejenem delu tabele in ga vstavi v levi, že urejeni del tabele, s čimer urejeni del tabele podaljša za en element.
- Tabele nam pridejo prav tudi pri memoizaciji — pomnjenju že izračunanih vrednosti (rekurzivne) metode.
- V javi se tipi delijo na primitivne in referenčne. Spremenljivka primitivnega tipa hrani ciljno vrednost, spremenljivka referenčnega tipa pa kazalec na ciljno vrednost. Ker so vsi tabelarični tipi referenčni, spremenljivka takega tipa ne hrani tabele, ampak zgolj kazalec nanjo. Tabele zato ne moremo skopirati s preprostimi prirejanjem spremenljivk.
- Tabelo kazalcev na tabele si lahko predstavljamo kot dvodimenzionalno tabelo, čeprav to v resnici ni. Posamezne tabele na drugem nivoju (tabele, na katere kažejo kazalci v glavni tabeli) lahko obravnavamo kot vrstice dvodimenzionalne tabele. Če imajo vse vrstice enako število elementov, lahko govorimo tudi o stolpcih.

***Iz profesorjevega kabineta***

»Od nekdanj sem bil za ta črne,« se pridruša prof. Doberšek, listajoč po dnevnem časopisu, »in vedno bom!«

»Nič nimam proti ta črnim, a vseeno imam raje tabele,« obotavlja odgovori as. Slapšak, ki svojemu v akademskih in drugih krogih nadvse spoštovanemu nadrejenemu sila nerad ugovarja. »Na primer, le kako bi z vašimi ta črnimi izpisal zaporedje  $n$  prebranih števil v obratnem vrstnem redu? Skoraj prepričan sem, da za to potrebujemo tabele!«

»Seveda, vaši ta beli se nenehno obračajo! Obračajo po vetru! Genovefa, mar nimam prav?«

»Ah, profesor, saj že poznate odgovor. Da in ne, kot vedno! Ta črni vam zaporedja prebranih števil res ne bodo obrnili. Vendar pa lahko odslovimo tudi tabele. Kaj tabele, še zankam se lahko odrečemo!«

»To bi pa res rad videl,« se vmeša asistent, pozabivši na svojo siceršnjo pokornost. »Gotovo si boš pomagala s kakšnim nizom ali pa z objektom tipa `java.util.List`.«

»Ne, nič takega! Zadoščala bosta tipa `int` in `boolean`. Dobro, pa `Scanner`, a temu bi se res težko izognila.«

Ima docentka prav ali se samo širokousti? Napišite program ali pa dokažite, da problema ni mogoče rešiti brez tabel (ali česa podobnega).

## 6 Razredi in objekti

Podobno kot tabela je tudi *objekt* skupek spremenljivk, vendar pa obstajajo pomembne razlike. Spremenljivke v tabeli pripadajo istemu tipu, tipi spremenljivk v objektu pa so lahko med seboj različni. Spremenljivke v tabeli naslavljamo z indeksi, v objektu pa z imeni. Ker so tipi in imena spremenljivk v objektu lahko poljubni, moramo imeti možnost, da jih izrecno navedemo. To storimo v *razredu* — konstrukt, ki vsebuje deklaracije tipov in imen spremenljivk, ki pripadajo posameznim objektom, po želji pa tudi definicije metod, ki jih lahko izvajamo nad posameznimi objekti. Razred je pravzaprav definicija tipa: objekt, ki pripada določenemu razredu, pripada tipu, ki ga ta razred določa.

### 6.1 Razredi, objekti, atributi

Objekt izdelamo, ko želimo množico spremenljivk, ki niso nujno istega tipa, obravnavati kot celoto. Denimo, da bi v programu želeli delati s podatki o osebah. Za vsako osebo bi radi hranili ime, priimek, leto rojstva, podatke o stalnem bivališču itd. Lahko bi sicer hranili tabelo tipa `String[]` za imena oseb, še eno tako tabelo za njihove priimke, tabelo tipa `int[]` za leta rojstva itd., a taka rešitev bi bila precej nepregledna in bi jo bilo težko vzdrževati in nadgrajevati. Bolje bi bilo, da bi podatke o vsaki osebi obravnavali skupaj, kot *objekt*, nato pa bi definirali tabelo takih objektov.

Tudi če so spremenljivke, ki jih želimo obravnavati kot celoto, istega tipa, je včasih bolj naravno definirati objekt kot tabelo. Tabela je smiselna, kadar spremenljivke tvorijo zaporedje, torej kadar je spremenljivka z indeksom 0 tudi logično prva spremenljivka, spremenljivka z indeksom 1 logično druga itd. Tipičen primer je, denimo, poraba električne energije po posameznih mesecih v koledarskem letu. Jasno je, da element z indeksom 0 predstavlja porabo v januarju, element z indeksom 1 porabo v februarju itd. Kadar pa ne moremo govoriti o vrstnem redu spremenljivk, je v splošnem bolj smiselno izdelati objekt. Ulomek, denimo, sicer lahko predstavimo s tabelo z dvema elementoma (npr. v celico z indeksom 0 zapišemo števec, v celico z indeksom 1 pa imenovalce), a ga raje predstavimo z objektom, saj ni posebnega razloga, da bi indeks 0 dodelili števcu, indeks 1 pa imenovalcu (ali obratno).

V obsežnejših programih je pogosto vsak koncept iz sveta, ki ga modeliramo, predstavljen z objektom (ali množico objektov). Na primer, v programu za podporo

knjižnici bi z objekti predstavili knjižnico ter posamezne knjige in člane, pa tudi neoprijemljive koncepte, kot je npr. izposoja, rezervacija ipd.

Objekti in tabele imajo določene skupne točke:

- Tako tabelo kot objekt ustvarimo z operatorjem `new`.
- Spremenljivka tabelaričnega tipa vsebuje kazalec na tabelo, ne tabele same. Spremenljivka objektnega tipa prav tako vsebuje kazalec na objekt, ne objekta samega. Tako tabelarični kot objektni tipi so torej referenčni tipi.

Med objekti in tabelami pa obstajajo tudi pomembne razlike:

- Kot smo že povedali, so vse spremenljivke v tabeli istega tipa, v objektu pa to ni nujno.
- Vse spremenljivke v tabeli naj bi imele enak osnovni pomen (npr. poraba električne energije), v objektu pa je njihova edina skupna točka ta, da pripadajo isti celoti (npr. podatkom o osebi).
- Spremenljivkam v tabeli (ali njihovim vrednostim) pravimo *elementi*, spremenljivkam v objektu (ali njihovim vrednostim) pa *atributi*.
- V tabeli so spremenljivke dostopne prek indeksov (*tabela[indeks]*), v objektu pa prek imen (*objekt.ime*).
- Tip tabele je povsem določen s tipom elementov. Na primer, tabela tipa `int[][]` je sestavljena iz elementov tipa `int[]`. Pri objektih pa je določitev tipa bolj zapletena, saj je treba podati tipe in imena atributov. To storimo s posebnim programskim konstruktom, ki mu pravimo *razred*.

Spomnimo se, da lahko tabelo izdelamo z operatorjem `new`, nato pa jo inicializiramo s pomočjo indeksov in prireditvenega operatorja:

```
int[] tocke = new int[3];
tocke[0] = 10;
tocke[1] = 20;
tocke[2] = 30;
```

Kot vemo, je spremenljivka `tocke` dejansko kazalec na tabelo treh elementov (slika 6.1, levo).

Če želimo izdelati in inicializirati objekt, pa moramo najprej definirati njegov tip. Kot smo povedali, to naredimo z razredom. Definirajmo razred za objekte, ki predstavljajo ulomke:<sup>1</sup>

<sup>1</sup>Ta razred bomo uporabljali tudi v nadaljnjih razdelkih in poglavjih.

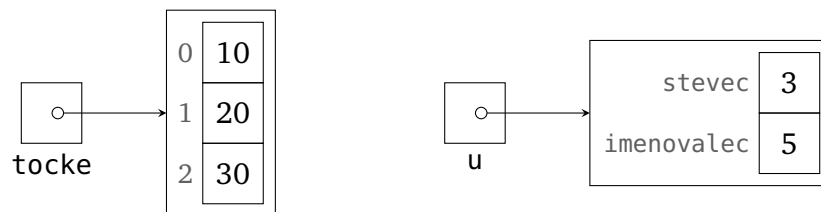
```
// koda 6.1 (razrediInObjekti/ulomek0/Ulomek.java)
public class Ulomek {
    int stevec;      // deklaracija atributa
    int imenovalec;  // deklaracija atributa
}
```

Vsak objekt razreda (tipa) `Ulomek` bo predstavljal nek ulomek. Ker smo v razredu `Ulomek` deklarirali atributa `stevec` in `imenovalec` (oba sta tipa `int`), bo tudi vsak objekt tipa `Ulomek` vseboval atributa `stevec` in `imenovalec` tipa `int`. Atribut `stevec` bo seveda predstavljal števec, atribut `imenovalec` pa imenovalec ulomka.

Izdelajmo in inicializirajmo objekt tipa `Ulomek`, ki predstavlja ulomek 3/5:

```
Ulomek u = new Ulomek();
u.stevec = 3;
u.imenovalec = 5;
```

Objekt, ki smo ga ustvarili, si lahko predstavljamo kot »škatlo« z dvema predalčkoma: eden se imenuje `stevec` in vsebuje število 3, drugi pa se imenuje `imenovalec` in vsebuje število 5. Spremenljivka `u` ne vsebuje samega objekta, ampak — tako kot pri tabelah — le kazalec nanj (slika 6.1, desno).



**Slika 6.1** Kazalec na tabelo tipa `int[]` (levo) in kazalec na objekt tipa `Ulomek` (desno).

Podobno kot pri tabelah bomo terminologijo tudi tukaj nekoliko poenostavili: namesto »objekt, na katerega kaže kazalec *k*« bomo pogosto pisali kar »objekt *k*«, seveda pa se moramo zavedati razlike med spremenljivko objektnega tipa in samim objektom.

Vsak javno dostopen zunanji razred definiramo v svoji datoteki.<sup>2</sup> Ime datoteke mora biti enako imenu razreda. Razred `Ulomek` torej definiramo v datoteki `Ulomek.java`, v ločenem razredu oz. datoteki (npr. `TestUlomek` oz. `TestUlomek.java`) pa ga preizkusimo. V razredu `TestUlomek` bomo definirali metodo `main`, v njej pa bomo ustvarili dva objekta tipa `Ulomek` in izpisali njuna atributa:

```
// koda 6.2 (razrediInObjekti/ulomek0/TestUlomek.java)
public class TestUlomek {
```

<sup>2</sup>Obstajajo tudi *notranji* razredi, a o tem (precej) kasneje. Zaenkrat bodo vsi naši razredi zunanji.

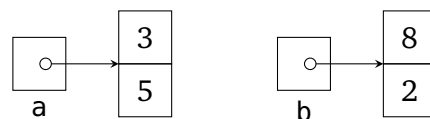
```

public static void main(String[] args) {
    Ulomek a = new Ulomek();
    a.stevec = 3;
    a.imenovalec = 5;
    System.out.printf("%d/%d%n", a.stevec, a.imenovalec);

    Ulomek b = new Ulomek();
    b.stevec = 8;
    b.imenovalec = 2;    // (1)
    System.out.printf("%d/%d%n", b.stevec, b.imenovalec);
}
}

```

Po izvedbi vrstice (1) se v pomnilniku nahajata objekta, ki predstavljata ulomka 3/5 in 8/2, spremenljivki a in b pa kažeta nanju (slika 6.2).



**Slika 6.2** Objekta tipa `Ulomek` in kazalca nanju.

Razred `Ulomek` za razliko od razreda `TestUlomek` ne vsebuje metode `main`, zato ga ne moremo pognati; lahko ga samo prevedemo. Vendar pa nam razreda `Ulomek` ni treba ločeno prevajati, saj prevajalnik sam ugotovi, da se v razredu `TestUlomek` sklicujemo na razred `Ulomek`. Ukaz

```
terminal> javac TestUlomek.java
```

bo tako prevedel oba razreda, `TestUlomek` in `Ulomek`. Prevedeni razred `TestUlomek` nato poženemo na običajen način:

```
terminal> java TestUlomek
```

Razredu, namenjenemu preizkušanju nekega drugega razreda, bomo rekli *testni razred*. Razred `TestUlomek` je torej testni razred za razred `Ulomek`.<sup>3</sup>

Razred `Ulomek` torej definira zgradbo objektov za predstavitev ulomkov. Oglejmo si še nekaj smiselnih primerov razredov:

- Razred za predstavitev kompleksnih števil:

<sup>3</sup>Posebne testnega razreda v resnici ne potrebujemo, saj lahko razred `Ulomek` vsebuje enako metodo `main`. Zavoljo večje jasnosti pa bomo praviloma pisali ločene testne razrede.

```
public class KompleksnoStevilo {
    double re;
    double im;
}
```

Vsak objekt tega razreda predstavlja kompleksno število. V atributu `re` je zapisana realna, v atributu `im` pa imaginarna komponenta.

- Razred za predstavitev točke v prostoru:

```
public class Tocka {
    double x;
    double y;
    double z;
}
```

- Razred za predstavitev datuma:

```
public class Datum {
    int dan;      // npr. 23
    int mesec;    // npr. 7
    int leto;     // npr. 2003
}
```

- Razred za predstavitev poštnega naslova:

```
public class PostniNaslov {
    String ulicaInHisnaStevilka; // npr. "Gozdna 42a"
    int postnaStevilka;           // npr. 3000
    String posta;                 // npr. "Celje"
}
```

- Razred za predstavitev predmeta na fakulteti:

```
public class Predmet {
    String ime;           // npr. "Programiranje 1"
    String sifra;         // npr. "63277"
    int steviloKT;        // število kreditnih točk, npr. 6
    boolean jeObvezen;    // true: predmet je obvezen
}
```

- Razred za predstavitev študenta:

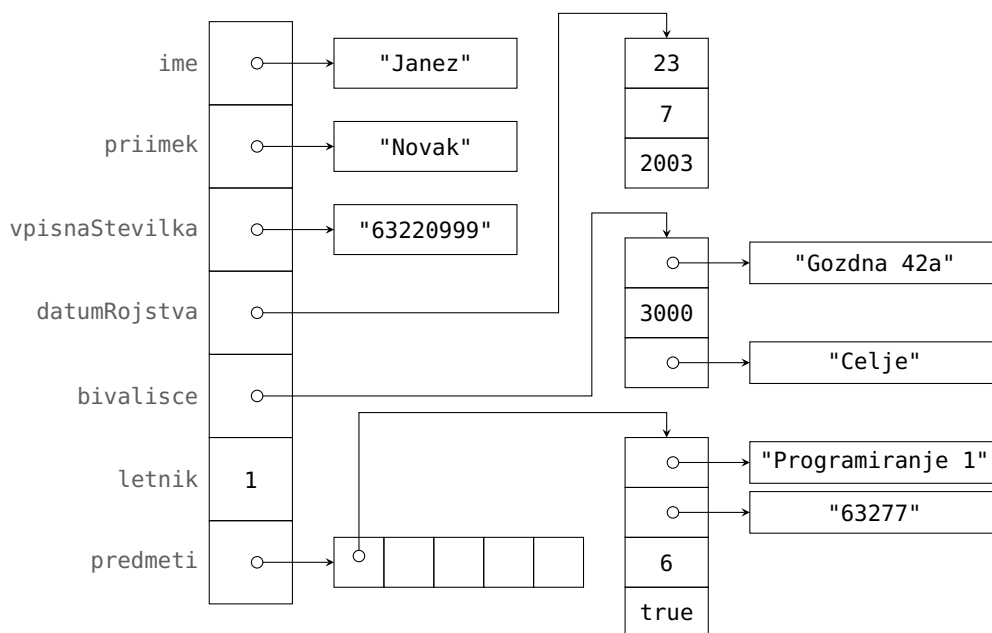
```

public class Student {
    String ime;           // npr. "Janez"
    String priimek;       // npr. "Novak"
    String vpisnaStevilka; // npr. "63220999"
    Datum datumRojstva;
    PostniNaslov bivalisce;
    int letnik;           // npr. 1
    Predmet[] predmeti;
}

```

Z izjemo atributa `letnik` so vsi atributi v tem razredu referenčnega tipa. Atribut `ime` je torej kazalec na objekt tipa `String`, atribut `datumRojstva` je kazalec na objekt tipa `Datum` itd. Atribut `predmeti` kaže na tabelo elementov tipa `Predmet`, ti pa kažejo na posamezne objekte tipa `Predmet`.

Primer objekta tipa `Student` je prikazan na sliki 6.3. V tabeli `predmeti` smo zaradi preglednosti prikazali le en kazalec na objekt tipa `Predmet`.



Slika 6.3 Primer objekta tipa `Student`.

**Naloga 6.1** Napišite razrede `Datum`, `PostniNaslov`, `Predmet` in `Student`. V posebnem testnem razredu po zgledu slike 6.3 izdelajte objekt tipa `Student`. Te razrede bomo v nadaljnjih nalogah še dopolnjevali.



## 6.2 Dostopna določila

Koncepte, ki jih bomo v nadaljevanju predstavili, bomo ilustrirali na primeru razreda `Cas`.<sup>4</sup> Objekt razreda `Cas` predstavlja trenutek v dnevu, izražen z uro in minuto. Razred bomo vseskozi dopolnjevali in po potrebi tudi spreminjali. Preizkušali ga bomo v testnem razredu `TestCas`.

Začetna različica razreda `Cas` je taka:

```
public class Cas {
    int ura;
    int minuta;
}
```

V testnem razredu izdelajmo objekt razreda `Cas`, ki predstavlja trenutek 10:35:

```
public class TestCas {
    public static void main(String[] args) {
        Cas trenutek = new Cas();
        trenutek.ura = 10;
        trenutek.minuta = 35;
    }
}
```

Z razredom `Cas` je navidez vse v najlepšem redu. Vendar pa smo zamolčali, da krši načelo dobre prakse, ki pravi, naj *atributi ne bodo vidni izven razreda*. Če nimamo res tehtnega razloga, naj do atributov ne bi dostopali neposredno (npr. s stavkom `trenutek.ura = 10`), ampak, kot bomo videli kasneje, samo prek skrbno načrtovanih *metod*.

Zakaj omejujemo dostop do atributov? Prvič zato, ker atributi predstavljajo *notranje stanje* objekta, ki ga je pogosto treba varovati pred nepooblaščenimi posegi in ohranjati v veljavnem in konsistentnem stanju. Na primer, avtomobil ima ogromno »atributov« (nivoji tekočin, prestavna razmerja, maksimalni navor ...), vendar pa jih je velika večina povsem nedostopnih običajnim uporabnikom. Voznik lahko z avtomobilom »komunicira« samo prek skrbno načrtovanih »metod«, kot so volan, prestavna ročica, pedali za plin, zavoro in sklopko, ročna zavora ipd. Tudi v manj kompleksnih primerih bi lahko prost dostop do atributov porušil veljavnost ali konsistentnost stanja objekta. Na primer, če pri razredu `Cas` omogočimo prost dostop do atributov, nimamo nikakršnega mehanizma, da uporabniku preprečimo izdelavo objekta, ki predstavlja neveljaven trenutek (npr. 24:30). Podobno bi lahko pri razredu, ki vsebuje atributa za predstavitev zakonskega partnerja in zakonskega statusa, uporabnik s prostim dostopom do atributov nastavil zakonskega partnerja na obstoječo osebo, zakonski status pa na »samski«.

<sup>4</sup>Ta razred bomo prav tako uporabljali tudi v naslednjih poglavjih.

Drugi razlog za omejevanje dostopa do atributov je zagotavljanje neodvisnosti od *implementacije* — konkretne izvedbe določene operacije ali organizacije določenega nabora podatkov. Če ohranimo glave in pomen posameznih metod, lahko njihovo implementacijo spremenimo (npr. poslužimo se učinkovitejšega nabora atributov), ne da bi bil uporabnik razreda pri tem kakorkoli oškodovan. Za lažje razumevanje si predstavljajmo razred *Pozicija*, katerega objekti predstavljajo posamezne pozicije na šahovnici. Razred ponuja metodo poteze, ki vrne seznam vseh veljavnih potez v podani poziciji, njegov osrednji atribut pa je tabela *razporeditev* tipa `int[][]`, v kateri element  $(i, j)$  podaja vsebino polja na koordinatah  $(i, j)$ . Če dostop do atributov omejimo, si lahko privoščimo, da neučinkovito tabelo *razporeditev* zamenjamo z bistveno učinkovitejšo bitno predstavitevjo, ne da bi se uporabnik razreda pri interakciji z njegovimi objekti moral kakorkoli prilagoditi. Če dostopa do atributov ne bi omejili, tega ne bi mogli storiti, saj bi obstajala možnost, da se uporabniki že poslužujejo tabele *razporeditev*.

V javi lahko dostopnost atributov in drugih elementov razreda nastavljamo s pomočjo *dostopnih določil*. Dostopno določilo navedemo na začetku deklaracije atributa oziroma glave metode. Java pozna sledeča dostopna določila:

- **public**: Do elementa, deklariranega z dostopnim določilom **public**, lahko dostopamo iz kateregakoli razreda.
- **protected**: Če je element razreda *R* deklariran s tem dostopnim določilom, potem lahko do njega dostopamo iz vseh razredov, ki pripadajo istemu paketu kot razred *R*, in iz podrazredov razreda *R*. Pakete smo bežno omenili v razdelku 2.9.2, s podrazredi pa se bomo seznanili v poglavju 7.
- **(brez)**: Če element razreda ni deklariran z nobenim dostopnim določilom, je dostopen iz vseh razredov v istem paketu.
- **private**: Če je element razreda *R* deklariran z dostopnim določilom **private**, je dostopen samo znotraj razreda *R*.

Atribute praviloma deklariramo z dostopnim določilom **private**:

```
public class Cas {
    private int ura;
    private int minuta;
}
```

Atributa *ura* in *minuta* sta odslej dostopna samo znotraj razreda *Cas*. Poskus dostopa v okviru drugega razreda (npr. *TestCas*) sproži napako pri prevajanju:

```
public class TestCas {
    public static void main(String[] args) {
```

```

        Cas trenutek = new Cas();
        trenutek.ura = 10;    // napaka pri prevajanju
        ...
    }
}

```

### 6.3 Izdelava objekta in konstruktor

Če so atributi privatni, jih izven razreda, v katerem so deklarirani, ne moremo neposredno nastavljati, zato potrebujemo drugačen način za inicializacijo objektov. Začetne vrednosti atributov objekta nastavljamo s posebno obliko metode, ki ji pravimo *konstruktor*. Ime konstruktorja je enako imenu razreda, za razliko od običajnih metod pa konstruktor nima izhodnega tipa (niti void). V konstruktorju praviloma nastavimo attribute objekta na privzete ali podane vrednosti.

Dodajmo konstruktor v razred *Cas*. Smiselno mu je dodeliti dva parametra: s prvim (naj bo *h*) podamo vrednost atributa *ura*, z drugim (naj bo *min*) pa vrednost atributa *minuta*. Konstruktor skopira parametra *h* in *min* v atributa *ura* in *minuta* objekta, ki je bil pravkar izdelan. Beseda *this*, kot bomo videli kasneje, se nanaša na pravkar izdelani objekt.

```

// koda 6.3 (razrediInObjekti/cas/Cas.java)
public class Cas {
    private int ura;
    private int minuta;

    public Cas(int h, int min) {    // konstruktor
        this.ura = h;
        this.minuta = min;
    }
}

```

Konstruktor pokličemo hkrati z operatorjem *new*:

```

// koda 6.4 (razrediInObjekti/cas/TestCas.java)
public class TestCas {
    public static void main(String[] args) {
        Cas trenutek = new Cas(10, 35);    // (1)
    }
}

```

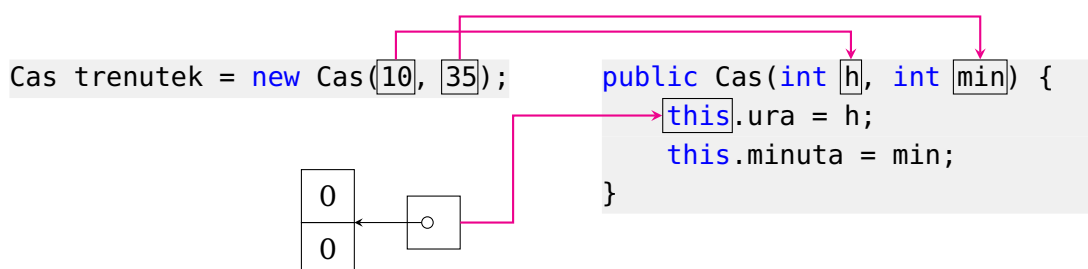
Po izvedbi vrstice (1) spremenljivka *trenutek* kaže na objekt, pri katerem je vrednost atributa *ura* enaka 10, vrednost atributa *minuta* pa 35.

Poimenovanje *konstruktor* je nekoliko zavajajoče. Konstruktor ne izdelava objekta; to naredi operator `new`. Res pa je, da se konstruktor pokliče neposredno po izdelavi objekta in je z njo zato tesno povezan. Oglejmo si zaporedje dogodkov ob izvedbi vrstice (1):

1. Operator `new` izdelava nov objekt v pomnilniku in neimenovan kazalec, ki kaže nanj.
2. Vsi atributi objekta se nastavijo na privzete vrednosti (tabela 5.1). Ker je privzeta vrednost za tip `int` enaka 0, se atributa `ura` in `minuta` nastavita na 0.
3. Pokliče se konstruktor. Ob klicu se argumenti (10 in 35 v našem primeru) skopirajo v parametre (`h` in `min`), poleg tega pa se kazalec na objekt, ki ga je ustvaril operator `new`, skopira v kazalec `this` (slika 6.4). S pomočjo kazalca `this` bo konstruktor lahko dostopal do izdelanega objekta.
4. Konstruktor se izvrši. Stavka `this.ura = h` in `this.minuta = min` skopirata vrednosti parametrov `h` in `min` v atributa `ura` in `minuta` objekta, na katerega kaže kazalec `this`, torej objekta, ki ga je ustvaril operator `new`.
5. Ko konstruktor opravi svoje delo, se zaključi tudi izvajanje operatorja `new`. Rezultat izraza `new R(...)` je kazalec na izdelani objekt razreda `R`. Po izvedbi stavka

```
Cas trenutek = new Cas(10, 35);
```

bo torej na izdelani objekt kazal kazalec `trenutek`.



Slika 6.4 Klic konstruktorja.

**Naloga 6.2** Napišite konstruktorje razredov `Datum`, `PostniNaslov`, `Predmet` in `Student`. Konstruktorji naj imajo toliko parametrov, kot je atributov.

**Naloga 6.3** V razredu `Ulocek` ustrezno popravite dostopni določili atributov, nato pa ga dopolnite s konstruktorjem

```
public Ulomek(int st, int im)
```

tako da bo novoustvarjeni objekt predstavljal okrajšano različico ulomka s števcem *st* in imenovalcem *im*. Ulomek je okrajšan, če sta si absolutni vrednosti števca in imenovalca medsebojno tuji in če je negativen kvečjemu števec. Če je števec enak 0, mora biti imenovalec enak 1. Lahko predpostavite, da je parameter *im* različen od 0.

## 6.4 Metode

### 6.4.1 Statične in nestatične metode

Objekte sedaj znamo izdelovati, prav veliko drugega pa z njimi še ne moremo početi. Če želimo, da bodo naši razredi uporabni, moramo definirati vsaj kakšno *metodo*. Ker smo attribute opremili z dostopnim določilom *private*, potrebujemo metode že za dostop do atributov. Pogosto pa definiramo metode tudi za nastavljanje posameznih atributov, vračanje niza, ki podaja vsebino objekta, izdelavo novega objekta na podlagi obstoječega, primerjavo objektov itd.

Vrnimo se k razredu *Cas* in napišimo metodo, ki podanemu trenutku prišteje *h* ur in *m* minut. Metoda potrebuje tri parametre:

- objekt tipa *Cas*, ki naj mu prišteje podano število ur in minut;
- število *h*;
- število *m*.

Metoda mora biti definirana znotraj razreda *Cas*, saj sicer ne more dostopati do atributov njegovih objektov. Najlažje jo sprogramiramo tako, da podani trenutek in prištevek pretvorimo v število minut od začetka dneva, obe vrednosti seštejemo in rezultat ponovno pretvorimo v ure in minute. Nekoliko sitnosti nam povzroči le dejstvo, da je lahko dobljena vsota minut izven intervala  $[0, 1439]$ . Če vsota ne bi mogla biti negativna, bi problem rešili preprosto z modulom po 1440, ker pa je v javi predznak vrednosti izraza  $a \% b$  enak predznaku števila *a*, bomo dobljenemu rezultatu prišteli 1440 in nato še enkrat izračunali ostanek pri deljenju s 1440.

```
public class Cas {
    ...
    public static void pristejStatic(Cas cas, int h, int min) {
        int noviCas = 60 * (cas.ura + h) + (cas.minuta + min);
        noviCas = (noviCas % 1440 + 1440) % 1440;
        cas.ura = noviCas / 60;
        cas.minuta = noviCas % 60;
    }
}
```

```
}
```

Če metodo pokličemo izven razreda `Cas`, moramo pred njen klic dodati še ime razreda, v katerem je definirana, in piko:

```
public class TestCas {
    public static void main(String[] args) {
        Cas trenutek = new Cas(10, 35);
        Cas.pristejStatic(trenutek, 2, 50); // (1)
    }
}
```

Po izvedbi vrstice (1) objekt `trenutek` predstavlja trenutek 13:25.

Metoda `pristejStatic` se izvrši natanko tako, kot pričakujemo: ob klicu se kazalec `trenutek` skopira v parameter `cas`, vrednost 2 v parameter `h`, vrednost 50 pa v parameter `min`. Kazalec `cas` v metodi `pristejStatic` torej kaže na isti objekt kot kazalec `trenutek` v metodi `main`.

Metoda `pristejStatic` ni napačna, vendar pa lahko metode, ki delujejo nad objektom razreda `R`, v katerem so definirane, napišemo bistveno jedrateje:

- odstranimo določilo `static`;
- odstranimo parameter tipa `R`;
- vse pojavitve odstranjenega parametra tipa `R` nadomestimo z besedo `this`.

Nova različica metode izgleda potemtakem takole ...

```
public void pristej(int h, int min) {
    int noviCas = 60 * (this.ura + h) + (this.minuta + min);
    noviCas = (noviCas % 1440 + 1440) % 1440;
    this.ura = noviCas / 60;
    this.minuta = noviCas % 60;
}
```

... pokličemo pa jo takole:

```
Cas trenutek = new Cas(10, 35);
trenutek.pristej(2, 50);
```

Vidimo, da sta tako definicija kot klic metode bistveno krajša. Metode v razredu `R`, ki delujejo nad vsaj enim objektom razreda `R`, praviloma pišemo kot *nestatične* metode, kar pomeni, da v glavi ne navedemo določila `static`. Določilo `static` uporabljamo za metode, ki ne delujejo nad objekti razreda `R`, ali pa (v redkih primerih) za metode, ki delujejo nad več takimi objekti. Takim metodam pravimo *statične* metode.

Statične metode deklariramo in kličemo tako, kot smo navajeni. Glava take metode v razredu  $R$  je v splošnem ( $D$  je dostopno določilo)

```
 $D$  static  $T$  metoda( $T_1$   $p_1$ ,  $T_2$   $p_2$ , ...,  $T_n$   $p_n$ )
```

pokličemo pa jo kot

```
 $R$ .metoda( $a_1$ ,  $a_2$ , ...,  $a_n$ )
```

pri čemer lahko  $R$  in piko izpustimo, če se klic metode nahaja v istem razredu kot sama metoda (v prejšnjih poglavjih je to vedno veljalo). Ob klicu metode se izvedejo sledeče prireditve:

```
 $p_1$  =  $a_1$   
 $p_2$  =  $a_2$   
...  
 $p_n$  =  $a_n$ 
```

Nestatične metode deklariramo brez določila `static` ...

```
 $D$   $T$  metoda( $T_1$   $p_1$ ,  $T_2$   $p_2$ , ...,  $T_n$   $p_n$ )
```

... pokličemo pa jih tako:

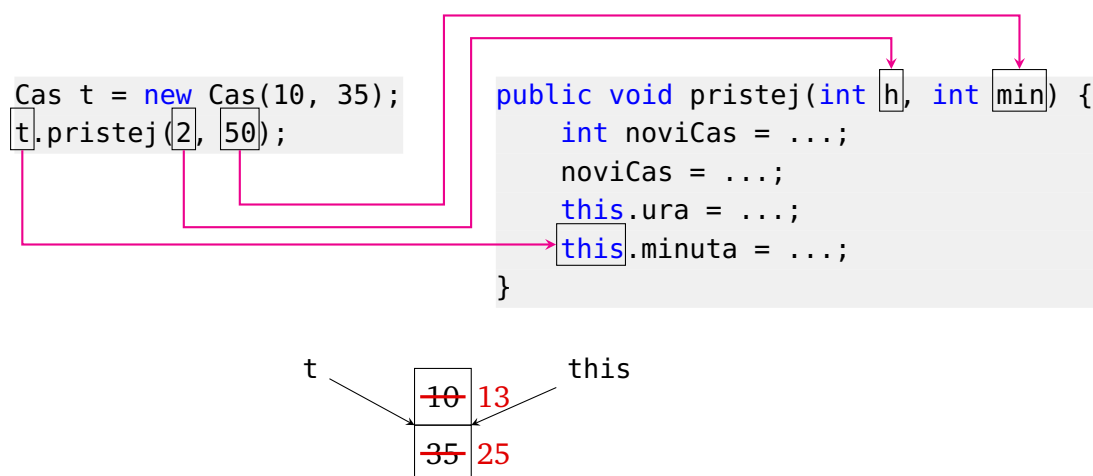
```
objekt.metoda( $a_1$ ,  $a_2$ , ...,  $a_n$ )
```

Ob klicu metode se izvršijo sledeče prireditve:

```
this = objekt  
 $p_1$  =  $a_1$   
 $p_2$  =  $a_2$   
...  
 $p_n$  =  $a_n$ 
```

Kazalec *objekt* torej deluje kot dodaten argument pri klicu metode. Nestatična metoda z  $n$  parametri se obnaša, kot da bi imela  $n + 1$  parametrov: poleg  $n$  argumentov, ki jih ob klicu navedemo v oklepajih za imenom metode, sprejme še »argument«, ki ga navedemo pred piko in imenom metode. Ta »argument« se ob klicu metode skopira v dodaten »parameter« po imenu `this`. To pomeni, da bo kazalec `this` kazal na isti objekt kot kazalec *objekt*. Slika 6.5 prikazuje, kaj se zgodi ob klicu metode pristej.

Čeprav se kazalec *objekt* pri klicu *objekt.metoda*(...) obnaša kot običajen argument, ima nekoliko poseben status, saj določa razred, v katerem bo javin prevajalnik iskal metodo *metoda*. Če je kazalec *objekt* tipa  $R$ , bo prevajalnik poskušal v razredu  $R$  poiskati metodo z ujemajočim imenom in zaporedjem tipov parametrov. Če je ne bo našel, bo javil napako. Na primer, pri zaporedju stavkov

Slika 6.5 Klic metode `pristej`.

```

Cas trenutek = new Cas(10, 35);
trenutek.pristej(2, 50);

```

prevajalnik zaradi deklaracije v prvi vrstici ve, da je kazalec `trenutek` tipa `Cas`, zato v razredu `Cas` poišče metodo z imenom `pristej` in dvema parametroma tipa `int`. Metodo najde, zato tvori ustrezen prevod, ki v času izvajanja pokliče metodo in ji posreduje kazalec `trenutek` ter števili 2 in 50.

Če metodo pokličemo kot `objekt.metoda(...)`, bomo rekli, da metodo izvedemo *nad* objektom *objekt* (oziroma nad objektom, na katerega kaže kazalec *objekt*, če smo natančnejši). Na primer, s klicem `trenutek.pristej(2, 50)` izvršimo metodo `pristej` nad objektom `trenutek`.

**Naloga 6.4** Razred `Student` dopolnite z metodo `public int skupajKT()`, ki vrne skupno število kreditnih točk predmetov študenta `this`.<sup>6</sup>

**Naloga 6.5** Razred

```

public class Stevilo {
    int n;

    public Stevilo(int nn) {
        this.n = nn;
    }
}

```

<sup>6</sup>Zopet smo ohlapni pri terminologiji: »študent `this`« je študent, ki ga predstavlja objekt, na katerega kaže kazalec `this`.



```

    public static boolean med(Stevilo a, Stevilo b, Stevilo c) {
        return a.n >= b.n && a.n <= c.n;
    }
}

```

dopolnite z metodo `vmes`, tako da se bo klic `p.vmes(q, r)` obnašal enako kot klic `Stevilo.med(p, q, r)`.

#### 6.4.2 Metode za vračanje vrednosti atributov (»getterji«)

V razredu `Cas` bomo napisali še več metod. Ker sta atributa `ura` in `minuta` deklarirana kot privatna, izven razreda `Cas` še vedno ne moremo do njiju. V ta namen definiramo metodi, ki vrnete vrednost enega oziroma drugega atributa podanega objekta. Metodi sprejmeta samo kazalec `this` — navidezni parameter tipa `Cas`, ki kaže na objekt, iz katerega naj metodi izluščita vrednost atributa `ura` oz. `minuta`:

```

public class Cas {
    ...
    public int vrniUro() {
        return this.ura;
    }

    public int vrniMinuto() {
        return this.minuta;
    }
}

```

Ker sta metodi nestatični, ju pokličemo kot `objekt.metoda()`:

```

public class TestCas {
    public static void main(String[] args) {
        Cas trenutek = new Cas(10, 35);
        System.out.println(trenutek.vrniUro());    // 10
        System.out.println(trenutek.vrniMinuto()); // 35
    }
}

```

Kazalec `trenutek` se ob klicu metode skopira v kazalec `this`. Znotraj metod `vrniUra` in `vrniMinuto` je torej objekt, nad katerim metodi delujeta, dostopen prek kazalca `this`. V metodah razreda `Cas` lahko dostopamo do privatnih atributov, zato izraza `this.ura` in `this.minuta` ne sprožita napake pri prevajanju.

Metodi `vrniUro` in `vrniMinuto` se v žargonu imenujeta »getterja«. V angleščini se namreč metode, ki vračajo vrednosti posameznih atributov, po nenapisanem pra-

vilu pričnejo z besedo *get* (npr. *getHours*, *getMinutes* itd.).

**Naloga 6.6** Razred *Ulomek* dopolnite z »getterjema«, ki vrneta števec in imenovalec okrajšanega ulomka. Na primer:

```
Ulomek u = new Ulomek(9, -6);
System.out.println(u.vrniStevec());    // -3
System.out.println(u.vrniImenovalec()); // 2
```

### 6.4.3 Metode za nastavljanje vrednosti atributov (»setterji«)

Poleg metod za vračanje vrednosti posameznih atributov lahko v razred dodamo tudi metode za njihovo nastavljanje oziroma — bolje rečeno — spreminjanje, saj se atributi nastavijo že ob klicu konstruktorja.

Metodi za nastavljanje atributov *ura* in *minuta* izgledata tako:

```
public class Cas {
    ...
    public void nastaviUro(int h) {
        this.ura = h;
    }

    public void nastaviMinuto(int min) {
        this.minuta = min;
    }
}
```

Ob klicu podamo objekt in novo vrednost njegovega atributa *ura* oziroma *minuta*:

```
Cas trenutek = new Cas(10, 35);
trenutek.nastaviUro(17);
System.out.println(trenutek.vrniUro()); // 17
```

Po analogiji z »getterji« takim metodam pravimo »setterji«, saj naj bi se njihova imena v angleško govorečih programih pričnjala z besedo *set* (npr. *setHours*, *setMinutes* itd.).

**Naloga 6.7** Sledeči razred dopolnite tako, da bo mogoče ustvarjati šolarje in določati njihove sošolce:

```
public class Solar {
    private String ip;           // ime in priimek šolarja this
    private Solar[] sosolci;     // sošolci šolarja this
}
```

V testnem razredu ustvarite nekaj šolarjev in nekatere med njimi določite za (vzajemne!) sošolce.

#### 6.4.4 (Ne)spremenljivost

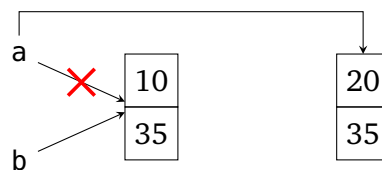
Medtem ko so »getterji« povsem neškodljivi, je treba biti pri »setterjih« nekoliko bolj previden. Kaj izpiše sledeči kos kode?

```
Cas a = new Cas(10, 35);
Cas b = a;
a.nastaviUro(20);
System.out.println(a.vrniUro());
System.out.println(b.vrniUro());
```

Dvakrat 20, seveda! Kazalca *a* in *b* kažeta na isti objekt, temu objektu pa potem nastavimo atribut ura na 20. Takšno obnašanje nas ne bi več smelo presenetiti, kljub temu pa ni zaželeno, saj lahko povzroči številne zahrbtne napake. Zato se, če je le mogoče, *izogibamo metodam, ki spreminjajo attribute objektov*. Razred, ki ne vsebuje takih metod, je *nespremenljiv* (angl. *immutable*). Pri nekaterih razredih nespremenljivost ni mogoča ali smiselna, pri nekaterih pa jo brez težav dosežemo. Namesto »setterjev« in drugih metod, ki spreminjajo attribute, preprosto ustvarimo nov objekt:

```
Cas a = new Cas(10, 35);
Cas b = a; // (1)
a = new Cas(20, a.vrniMinuto());
System.out.println(a.vrniUro()); // 20
System.out.println(b.vrniUro()); // 10
```

V vrstici (1) kazalca *a* in *b* sicer kažeta na isti objekt, toda že v naslednji vrstici kazalec *a* pokaže na nov objekt (slika 6.6).



Slika 6.6 Kazalec *a* pokaže na nov objekt.

Metoda pristoj, ki smo jo napisali v razdelku 6.4.1, objektu, nad katerim deluje, spremeni atributa ura in minuta. Boljša je sledeča metoda, ki obstoječi objekt (objekt, na katerega kaže kazalec *this*) pusti pri miru ter raje ustvari in vrne nov objekt:

```

public Cas plus(int h, int min) {
    int noviCas = 60 * (this.ura + h) + (this.minuta + min);
    noviCas = (noviCas % 1440 + 1440) % 1440;
    int novaUra = noviCas / 60;
    int novaMinuta = noviCas % 60;
    return new Cas(novaUra, novaMinuta);
}

```

Metodo lahko uporabimo takole:

```

Cas trenutek = new Cas(10, 35);
Cas kasneje = trenutek.plus(2, 50);

```

**Naloga 6.8** Razred Ulomek dopolnite s sledečimi metodami:

- `public Ulomek plus(Ulomek u)`  
Vrne vsoto ulomkov `this` in `u`.
- `public Ulomek negacija()`  
Vrne nasprotno vrednost ulomka `this`. Nasprotna vrednost ulomka  $a/b$  je ulomek  $-a/b$ .
- `public Ulomek minus(Ulomek u)`  
Vrne razliko ulomkov `this` in `u`. Pomagajte si z metodama vsota in negacija.

Vse tri metode morajo ustvariti in vrniti nov objekt, ki predstavlja ciljni ulomek. Nobena ne sme spreminjati objekta `this`.

#### 6.4.5 Metode za izpis vsebine in vračanje vsebine v obliki niza

Ker objekt tipa `Cas` predstavlja trenutek v dnevu, je smiselno, da lahko njegovo vsebino izpišemo v obliki `h:mm` (npr. 9:07). V ta namen lahko uporabniku razreda `Cas` ponudimo tako metodo:

```

public void izpisi() {
    System.out.printf("%d:%02d", this.ura, this.minuta);
}

```

Še bolj uporabna pa je metoda, ki niza `h:mm` ne izpiše, ampak ga vrne:

```

public String toString() {
    return String.format("%d:%02d", this.ura, this.minuta);
}

```

Metoda `String.format` deluje enako kot klic `System.out.printf(...)`, le da niza, ki ga zgradi, ne izpiše, ampak ga vrne kot svoj rezultat.

V čem je prednost metode `toString` pred metodo `izpisi`? Če niz izpišemo, ne moremo z njim nič več početi. Če pa ga namesto tega vrnemo, ga lahko uporabnik po želji še vedno izpiše ...

```
System.out.println(trenutek.toString());
```

... lahko pa ga uporabi, denimo, kot sestavni del nekega večjega niza:

```
Cas kosilo = new Cas(12, 0);
String porocilo = String.format(
    "Ob %s smo pojedli kosilo, ob %s pa smo šli teč.",
    kosilo.toString(), kosilo.plus(3, 30).toString()
);
```

Gotovo ste opazili, da smo metodo poimenovali po angleško namesto po slovensko (npr. `vNiz`). Če uporabimo ime `toString`, lahko namreč namesto

```
System.out.println(trenutek.toString());
```

pišemo kar

```
System.out.println(trenutek);
```

saj metoda `println` samodejno pokliče metodo `toString`, če ji kot argument podamo vrednost referenčnega tipa. To velja tudi za metodi `print` in `printf`. Več o metodi `toString` bomo povedali v razdelku 7.4.1.

**Naloga 6.9** Razred `Ulocek` dopolnite z metodo `toString`.

**Naloga 6.10** Napišite razred `Oseba`, tako da bodo njegovi objekti predstavljali osebe s podanim imenom, priimkom, spolom in letom rojstva.<sup>7</sup> Napišite »getterje« za posamezne attribute in metodo `toString` (zgledujte se po nizu Janez Novak (M), 2003).

**Naloga 6.11** Stavek `System.out.println(...)` očitno kliče metodo `println`. S pomočjo javine dokumentacije ugotovite, kateremu razredu pripada ta metoda. Je metoda statična ali nestatična? Kaj je `System.out`?

#### 6.4.6 Metode za primerjanje objektov

Tovrstne metode praviloma sprejmejo dva objekta in ju med seboj primerjajo. Najprej bomo napisali metodo `jeEnakKot`, ki vrne `true` natanko tedaj, ko podana objekta

<sup>7</sup>Še en razred, ki nas bo spremljal tudi v nadaljnjih poglavjih ...

predstavljata isti trenutek. Koliko parametrov naj ima ta metoda? Ker en objekt sprejme prek kazalca `this`, potrebuje le še en »pravi« parameter. Glava metode po temtatem izgleda takole:

```
public boolean jeEnakKot(Cas drugi)
```

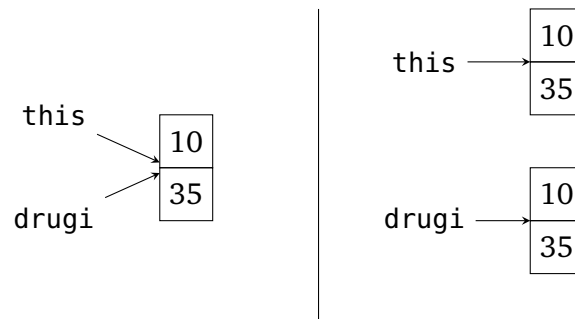
Če metodo pokličemo, denimo, kot `malica.jeEnakKot(kosilo)`, se kazalec `malica` skopira v kazalec `this`, kazalec `kosilo` pa v kazalec `drugi`.

Kako bi napisali metodo za preverjanje enakosti objektov? Poskusimo takole:

```
public boolean jeEnakKot_poskus(Cas drugi) {
    return this == drugi;
}
```

Metoda vrne `true` natanko tedaj, ko kazalca `this` in `drugi` kažeta na isti objekt (slika 6.7, levo). Vendar pa bi radi upoštevali tudi možnost, da kazalca kažeta na dva ločena objekta, ki se med seboj ujemata v obeh atributih (slika 6.7, desno). Sledeča metoda pokriva oba scenarija:

```
public boolean jeEnakKot(Cas drugi) {
    return this.ura == drugi.ura && this.minuta == drugi.minuta;
}
```



**Slika 6.7** Levo: kazalca sta enaka, ker kažeta na isti objekt. Desno: kazalca sta različna, njuna ciljna objekta pa imata enako vsebino.

Ilustrirajmo razliko s primerom. Kazalca `a` in `b` kažeta na isti objekt, kazalec `c` pa na ločen objekt z enako vsebino:

```
Cas a = new Cas(10, 35);
Cas b = a;
Cas c = new Cas(10, 35);
Cas d = new Cas(8, 40);
```

```

System.out.println(a.jeEnakKot_poskus(b)); // true
System.out.println(a.jeEnakKot(b));       // true

System.out.println(a.jeEnakKot_poskus(c)); // false
System.out.println(a.jeEnakKot(c));       // true

System.out.println(a.jeEnakKot_poskus(d)); // false
System.out.println(a.jeEnakKot(d));       // false

```

Pri delu s časovnimi trenutki nam lahko koristi tudi metoda, ki vrne razliko med podanima trenutkoma v minutah:

```

public int razlikaVMin(Cas drugi) {
    return (this.ura - drugi.ura) * 60 + this.minuta - drugi.minuta;
}

```

Na primer, klic

```
new Cas(10, 35).razlikaVMin(new Cas(13, 10))
```

vrne vrednost  $-155$ .

Napišimo še dve metodi za primerjavo objektov tipa `Cas`. Metoda `jeManjsiOd` vrne `true` natanko tedaj, ko je trenutek `this` strogo manjši od trenutka `drugi` (trenutek `this` se zgodi pred trenutkom `drugi`), metoda `jeManjsiAliEnak` pa poleg tega dopušča tudi možnost, da sta trenutka enaka.

```

public boolean jeManjsiOd(Cas drugi) {
    return this.ura < drugi.ura ||
        (this.ura == drugi.ura && this.minuta < drugi.minuta);
}

public boolean jeManjsiAliEnak(Cas drugi) {
    return this.jeManjsiOd(drugi) || this.jeEnakKot(drugi);
}

```

Na primer:

```

Cas predavanja = new Cas(8, 15);
Cas vaje = new Cas(11, 15);
Cas sestanek = new Cas(11, 15);
System.out.println(predavanja.jeManjsiOd(vaje)); // true
System.out.println(vaje.jeManjsiOd(sestanek)); // false
System.out.println(vaje.jeManjsiAliEnak(sestanek)); // true

```

Pri obeh metodah si lahko pomagamo tudi z metodo `razlikaVMin`. Trenutek

`this` je manjši od trenutka `drugi`, ko je razlika med njima (`>this minus drugi`) negativna:

```
public boolean jeManjsiOd(Cas drugi) {
    return this.razlikaVMin(drugi) < 0;
}
```

**Naloga 6.12** Razred `Cas` dopolnite s statično metodo `jePrviManjsi`, tako da bo klic `Cas.jePrviManjsi(p, q)` enakovreden klicu `p.jeManjsiOd(q)`.

**Naloga 6.13** Razred `Ulolek` dopolnite z metodo

```
public int primerjaj(Ulolek drugi)
```

ki vrne vrednost 0, če sta ulomka `this` in `drugi` enaka, poljubno negativno vrednost, če je ulomek `this` manjši od ulomka `drugi`, in poljubno pozitivno vrednost, če je ulomek `this` večji od ulomka `drugi`.

**Naloga 6.14** Razred `Oseba` iz naloge 6.10 dopolnite z metodo

```
public int primerjaj(Oseba druga)
```

ki osebi `this` in `druga` primerja po priimku, osebi z enakim priimkom pa po imenu. Metoda naj vrne negativno vrednost, če oseba `this` po opisanem kriteriju spada pred osebo `druga`, vrednost 0, če se osebi po tem kriteriju med seboj ne razlikujeta, in pozitivno vrednost, če oseba `this` po navedenem kriteriju sodi za osebo `druga`.

#### 6.4.7 Izpuščanje besede `this`

Dostop do nestatičnega atributa *a* ali klic nestatične metode *f* je mogoč samo z navedbo objekta, pri katerem nas zanima atribut *a* oziroma nad katerim naj deluje metoda *f*. Na primer, če delamo z objekti razreda `Cas`, lahko pišemo le `objekt.ura`, `objekt.minuta`, `objekt.vrniUro()` itd., kjer je `objekt` kazalec na objekt tipa `Cas`. Obstaja pa pomembna izjema: namesto `this.atribut` in `this.metoda(...)` lahko pišemo kar `atribut` in `metoda(...)`. Če se znotraj nestatične metode pojavi gol sklic na atribut ali metodo, prevajalnik predenj sam doda besedo `this` in piko.

Metodi `vrniUro` in `jeManjsiAliEnak` bi torej lahko napisali takole:

```
public int vrniUro() {
    return ura;
}

public boolean jeManjsiAliEnak(Cas drugi) {
    return jeManjsiOd(drugi) || jeEnakKot(drugi);
}
```



```
}
```

Čeprav ni nič narobe, če besedo `this` izpustimo, kadar je to mogoče (tako počnejo številni programerji), jo bomo v tej knjigi dosledno pisali, saj menimo, da njena uporaba zmanjša možnost napak, kot je, denimo, nenamerna zamenjava atributa in enako poimenovane lokalne spremenljivke.

## 6.5 Primer uporabe razreda

Čas je, da razred `Cas` uporabimo v praksi. Napišimo program, ki izpiše vozni red avtobusa, če vemo, da ta z začetne postaje prvič odpelje ob času  $h_{zac}:min_{zac}$ , zadnjič najkasneje ob času  $h_{kon}:min_{kon}$ , vmes pa vozi na vsakih  $k$  minut. Na primer, če je  $h_{zac} = 5$ ,  $min_{zac} = 30$ ,  $h_{kon} = 22$ ,  $min_{kon} = 0$  in  $k = 75$ , potem je naš vozni red videti takole:

5:30	6:45	8:00	9:15	10:30	11:45	13:00
14:15	15:30	16:45	18:00	19:15	20:30	21:45

Če ne bi imeli na voljo razreda `Cas`, bi delali z urami in minutami ali pa (enostavneje) samo z minutami, ki bi jih sproti pretvarjali v ure in minute. Nobena rešitev ne bi bila posebej zahtevna, toda razred `Cas` nam omogoča, da se problema lotimo na višji ravni. Namesto da se ukvarjamo z urami in minutami, delamo kar z objekti tipa `Cas`. Namesto da pare ura-minuta med seboj primerjamo z razmeroma kompleksnim logičnim izrazom ali pa s pretvorbo v minute, lahko enostavno uporabimo metodo `jeManjSiAliEnak` in se ne spuščamo v umazane podrobnosti. Razred `Cas` nam torej omogoča, da na časovni trenutek gledamo kot na celoto, ne kot na par ura-minuta.

Še več: s stališča uporabnika razreda `Cas` je povsem vseeno, kako je predstavljen časovni trenutek in kako so implementirane posamezne metode; pomembno je le, kako lahko objekt ustvari, katere metode so mu na voljo in kako lahko te metode uporablja.

Preidimo k rešitvi. Po branju podatkov ustvarimo dva objekta, `cas`, ki predstavlja trenutni čas, in `casKon`, ki predstavlja najpoznejši čas odhoda z začetne postaje. Nato vstopimo v zanko, v kateri s pomočjo metode `plus` izpisujemo in posodabljammo trenutni čas (dejansko vsakokrat ustvarimo nov objekt, ki predstavlja posodobljeni trenutek). Zanko izvajamo, dokler je trenutni čas manjši ali enak ciljnemu.

```
// koda 6.5 (razrediInObjekti/cas/VozniRed.java)
public class VozniRed {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int uraZac = sc.nextInt();
        int minutaZac = sc.nextInt();
```

```

        int uraKon = sc.nextInt();
        int minutaKon = sc.nextInt();
        int interval = sc.nextInt();

        Cas cas = new Cas(uraZac, minutaZac);
        Cas casKon = new Cas(uraKon, minutaKon);
        while (cas.jeManjsiAliEnak(casKon)) {
            System.out.println(cas);
            cas = cas.plus(0, interval);
        }
    }
}

```

**Naloga 6.15** Razred `Ulolek` preizkusite z metodo

```
public static Ulolek harmonicnaVsota(int k)
```

ki vrne vrednost vsote  $\sum_{n=1}^k \frac{1}{n}$ .

**Naloga 6.16** Predelajte razred `Cas` tako, da bodo časovni trenutki predstavljeni samo z minutami namesto z urami in minutami. Na primer, trenutek 10:35 naj bo predstavljen s številom  $10 \cdot 60 + 35 = 635$ . Z vidika uporabnika razreda se razred ne sme spremeniti: glave konstruktorja in metod morajo ostati enake, le njihovo implementacijo morate ustrezno prilagoditi. Program za izpis voznega reda mora torej delovati v nespremenjeni obliki.

## 6.6 Statični atributi

Videli smo, kaj določilo `static` pomeni pri metodah: statična metoda ob klicu prejme natanko toliko vrednosti, kot ima parametrov, nestatični pa se poleg argumentov posreduje še kazalec na objekt, ki stoji pred piko in imenom metode.

Kaj pa beseda `static` označuje pri atributih? Nestatičen atribut pripada *vsakemu objektu posebej*, statičen pa *razredu kot celoti*. Na primer, pri razredu za predstavitev podatkov o osebah bi bili atributi `ime`, `priimek`, `starost` ipd. nestatični, saj ima vsaka oseba svoje ime, svoj priimek, svojo starost itd. Primer statičnega atributa pa bi bil, recimo, števec doslej ustvarjenih objektov. To ni lastnost, ki bi pripadala vsakemu posameznemu objektu (npr. Janezu, Mariji ...) posebej, ampak gre za spremenljivko, ki pripada celotnemu razredu.

Pri razredu `Cas` sta atributa `ura` in `minuta` seveda nestatična, saj ima vsak trenutek svojo uro in svojo minuto: trenutek 10:35 ima drugačni vrednosti teh atributov kot trenutek 15:20. Primer statičnega atributa pa je atribut, imenovali ga bomo

zapis12, ki določa način zapisa časovnega trenutka: če je njegova vrednost `false`, se trenutki izpisujejo v običajni 24-urni obliki (npr. 10:35, 15:20 ...), če je `true`, pa v 12-urni obliki, kot jo uporabljajo v angleško govorečem svetu (npr. 10:35 AM, 3:20 PM ...). Atribut `zapis12` je neodvisen od konkretnega objekta, saj določa lastnosti, ki veljajo za vse objekte razreda `Cas` (oziroma, pravilneje rečeno, za razred `Cas` kot celoto).

Kje inicializiramo statični atribut? Konstruktor ni pravo mesto, saj so statični atributi neodvisni od posameznih objektov. Ponavadi jih inicializiramo kar ob deklaraciji:

```
public class Cas {
    private static boolean zapis12 = false;
    ...
}
```

Če za inicializacijo potrebujemo več kode, uporabimo poseben blok, imenovan *statični inicializator*. V sledečem primeru v statičnem inicializatorju inicializiramo tabelo fakultet števil od 0 do 19:

```
public class ... {
    private static int[] fakulteta;

    static { // statični inicializator
        fakulteta = new int[20];
        fakulteta[0] = 1;
        for (int i = 1; i < fakulteta.length; i++) {
            fakulteta[i] = fakulteta[i - 1] * i;
        }
    }
    ...
}
```

Vrnimo se na razred `Cas`. Če želimo spreminjati vrednost atributa `zapis12`, potrebujemo metodo. Metoda bo seveda statična, saj ne deluje nad nobenim objektom:

```
public class Cas {
    ...
    public static void nastaviZapis12(boolean da) {
        zapis12 = da;
    }
    ...
}
```

Vidimo, da pri dostopu do atributa `zapis12` nismo navedli besede `this`. Izraz `this`.

zapis12 bi bil namreč celo dvakrat napačen: prvič zato, ker kazalec `this` v statičnih metodah sploh ne obstaja, drugič pa zato, ker statični atributi ne pripadajo nobenemu objektu in je zapis `objekt.staticniAtribut` torej nesmiseln.

Popraviti moramo še metodo `toString`. V dvanajsturnem zapisu imajo trenutki od 0:00 do 11:59 pripono AM, trenutki od 12:00 do 23:59 pa pripono PM. Minute ostanejo nedotaknjene, ure pa se pretvorijo po pravilu  $h' = (h + 11) \bmod 12 + 1$ :

```
public String toString() {
    if (zapis12) {
        String pripona = (this.ura < 12) ? ("AM") : ("PM");
        int h = (this.ura + 11) % 12 + 1;
        return String.format("%d:%02d %s", h, this.minuta, pripona);
    }
    return String.format("%d:%02d", this.ura, this.minuta);
}
```

Spodobi se, da najnovejše pridobitve tudi preizkusimo:

```
Cas a = new Cas(10, 35);
Cas b = new Cas(15, 20);
System.out.printf("%s, %s\n", a, b); // 10:35, 15:20
Cas.nastaviZapis12(true);
System.out.printf("%s, %s\n", a, b); // 10:35 AM, 3:20 PM
```

Kot že vemo, se v okviru klica metode iz družine `print*` pri izpisovanju objektov samodejno doda klic metode `toString`. Stavek

```
System.out.printf("%s, %s\n", a, b);
```

je potemtakem enakovreden stavku

```
System.out.printf("%s, %s\n", a.toString(), b.toString());
```

Pomembno se je zavedati, da imajo objekti razreda `Cas` še vedno po dva atributa (`ura` in `minuta`), medtem ko je statični atribut `zapis12` shranjen v posebnem prostoru, ki pripada razredu kot celoti.

**Naloga 6.17** Razred `Ulomek` dopolnite s statičnim atributom, ki šteje ustvarjene ulomke (objekte tipa `Ulomek`). Napišite tudi metodo za vračanje vrednosti tega atributa. Na primer:

```
Ulomek a = new Ulomek(3, 5);
Ulomek b = new Ulomek(4, 7);
Ulomek c = a;
Ulomek d = a.minus(b);
```

```
Ulomek e = a.plus(c);
System.out.println(Ulomek.steviloUstvarjenih()); // 4
```

## 6.7 Več metod z istim imenom

Včasih želimo definirati več metod, ki počnejo (bolj ali manj) isto stvar, a se razlikujejo po parametrih. Smiselno je, da take metode tudi enako poimenujemo. Java dopušča več metod z istim imenom v istem razredu, če se razlikujejo po *podpisih*. Podpis metode je niz oblike

$$ime(T_1, T_2, \dots, T_n)$$

kjer je *ime* ime metode,  $T_1, \dots, T_n$  pa so tipi posameznih parametrov. Na primer, podpis metode

```
public static boolean preveri(int stevilo, String niz)
```

se glasi

```
preveri(int, String)
```

Ker morajo imeti metode v istem razredu različne podpise, se morajo metode z enakim imenom med seboj razlikovati po zaporedju tipov parametrov. Ni dovolj, da spremenimo imena parametrov, izhodni tip ali zaporedje določil.

Ko pokličemo eno od več metod z enakimi imeni, prevajalnik poišče »pravo« metodo na podlagi tipov argumentov. To je nekoliko težje, kot se zdi na prvi pogled, saj tip argumenta ni nujno enak tipu pripadajočega parametra; zadošča, da je prvi tip priredljiv drugemu. Ta lastnost lahko privede do nerazrešljive dvoumnosti. Recimo, da razred vsebuje sledeče metode:

```
public static void f(long a) { ... }
public static void f(double b) { ... }
public static void f(long a, double b) { ... }
public static void f(double a, long b) { ... }
```

Metode se razlikujejo po podpisih, zato se razred prevede. Sedaj pa si oglejmo primere klicev metod:

```
f(3);
f(4.0);
f(3, 4.0);
f(3, 4);
```

Prvi stavek bi lahko poklical metodo `f(long)` ali `f(double)`, vendar pa se pokliče prva metoda, saj je tipu `int` po prevajalnikovih pravilih bližji tip `long` kot tip `double`. Pri drugem in tretjem stavku ni dilem, četrti pa sproži napako pri prevajanju, saj ni mogoče ugotoviti, ali naj se pokliče tretja ali četrta metoda.

## 6.8 Več konstruktorjev

Pri konstruktorjih so pravila podobna kot pri metodah. Razred lahko vsebuje več konstruktorjev, če se ti razlikujejo po zaporedju tipov parametrov. Na primer, razredu `Cas` bi lahko dodali konstruktor s parametrom `h`, ki uro nastavi na `h`, minuto pa pusti na privzeti ničli:

```
public Cas(int h) {
    this.ura = h;
}

public Cas(int h, int min) {
    this.ura = h;
    this.minuta = min;
}
```

Ta vzorec je precej pogost: poleg izhodiščnega konstruktorja, ki vse relevantne attribute nastavi na vrednosti pripadajočih parametrov, definiramo še več konstruktorjev z manjšim številom parametrov, ki nastavijo samo nekatere attribute. V takšnih primerih nam pride prav, če lahko v okviru konstruktorja pokličemo nek drug konstruktor, saj se lahko na ta način izognemo podvajanju kode. Na primer, v konstruktorju `Cas(int)` lahko pokličemo konstruktor `Cas(int, int)` in mu posredujemo argumenta `h` in `0`. Po analogiji s klici metod bi to storili takole:

```
public Cas(int h) {
    Cas(h, 0);
}
```

Žal pa tak poskus povzroči napako pri prevajanju. Drug konstruktor v istem razredu pokličemo s pomočjo besede `this`:

```
public Cas(int h) {
    this(h, 0);
}
```

Bodimo pozorni na troje. Prvič, beseda `this` ima v izrazu `this(...)` drugačno vlogo kot v izrazih `this.atribut` ali `this.metoda`. V drugem primeru je `this` kazalec na objekt, v prvem pa ta beseda nima posebnega pomena, ampak gre zgolj za

klic konstruktorja. Drugič, stavek `this(...)` je mogoč samo znotraj konstruktorja. Tretjič, če je stavek `this(...)` prisoten, mora to biti prvi stavek v konstruktorju.

Če v razredu ne definiramo nobenega konstruktorja, potem javanski prevajalnik samodejno doda t.i. *privzeti konstruktor*. To je konstruktor brez parametrov, ki vsebuje zgolj sledeči stavek:

```
super();
```

Kot bomo videli v poglavju 7, ta stavek pokliče konstruktor nadrazreda. Povedali bomo, da je nadrazred razreda `Cas` razred `Object`, njegov konstruktor pa ne naredi ničesar.

Če definiramo vsaj en lasten konstruktor, potem prevajalnik *ne* doda privzetega konstruktorja. V našem primeru izraz `new Cas()` potemtakem ni mogoč.

## 6.9 Razred String

Razred `String` uporabljamo že od vsega začetka, sedaj pa je napočil čas, da si ga nekoliko pobliže ogledamo. Objekti tipa `String` so, kot vemo, *nizi* — zaporedja znakov. Razred `String` je nekoliko poseben, saj nam poleg običajnega načina izdelave objektov ...

```
String niz = new String("programiranje");
```

... nudi tudi krajšo pot:

```
String niz = "programiranje";
```

Poleg tega lahko nad nizi uporabljamo tudi operatorja `+` in `+=`:

```
String a = "programiranje";
String b = "v javi";
String c = a + " " + b;    // "programiranje v javi"
c += " SE";               // "programiranje v javi SE"
```

Spremenljivka tipa `String` je seveda zgolj kazalec na objekt, ki vsebuje niz, zato nizov ne smemo primerjati z operatorjem `==`, razen če nas izrecno zanima, ali dva kazalca kažeta na isti niz:

```
String a = "programiranje";
String b = a;
String c = new String("programiranje");
System.out.println(a == b);    // true
System.out.println(a == c);    // false
```

Če želimo nize primerjati po vsebini, pokličemo metodo `equals`:

```
String a = "programiranje";
String b = a;
String c = new String("programiranje");
System.out.println(a.equals(b));    // true
System.out.println(a.equals(c));    // true
```

Razred `String` ponuja celo vrsto uporabnih metod. Lahko jih preučite v uradni javini dokumentaciji, na tem mestu pa bomo nekatere od njih zgolj prikazali na primeru:

```
// koda 6.6 (razrediInObjekti/nizi/Nizi.java)
String niz = "programiranje";

// pridobi znak na podanem indeksu
System.out.println(niz.charAt(3));    // g

// leksikografsko primerjaj niza
System.out.println(niz.compareTo("java"));
// > 0, ker niz "programiranje" po abecedi sodi za niz "java"

// preveri, ali se niz prične oz. konča z določenim nizom
System.out.println(niz.startsWith("prog"));    // true
System.out.println(niz.endsWith("ranje"));    // true

// poišči znak ali podniz
System.out.println(niz.indexOf('g'));    // 3
System.out.println(niz.indexOf("mira"));    // 6

// vrni dolžino oz. preveri praznost
System.out.println(niz.length());    // 13
System.out.println(niz.isEmpty());    // false

// pridobi podniz med podanima indeksoma
System.out.println(niz.substring(2, 7));    // ogram

// zamenjaj podniz
System.out.println(niz.replace("gramir", "slavlj"));    // proslavljanje

// odstrani začetne in končne presledke
System.out.println("  abc  ".strip());    // abc

// razbij na komponente, ločene s poljubnim zaporedjem presledkov,
```



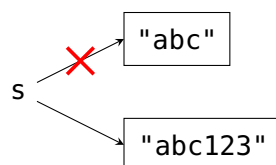
```
// tabulatorjev in prelomov vrstice (\\s predstavlja presledek,
// tabulator ali prelom vrstice, + pa pomeni "eden ali več")
String[] besede = "danes je lep dan".split("\\s+");
for (String beseda: besede) {
    System.out.println(beseda);
}
// danes, je, lep, dan

// to pa že poznamo ...
int a = 3;
int b = 4;
System.out.println(String.format("%d + %d = %d", a, b, a + b));
```

Razred String je nespremenljiv, zato nobena od njegovih metod ne spremeni objekta, nad katerim jo pokličemo. Vse metode ustvarijo nov niz, niz `this` pa pustijo nedotaknjen. To velja tudi za operator `+=`. Na primer, koda

```
String s = "abc";
s += "123";
```

ustvari nov niz `"abc123"`, nato pa s kazalcem `s` pokaže nanj (slika 6.8). Ker na niz `"abc"` ne kaže več noben kazalec, ga bo *smetar* (poseben proces, ki teče v ozadju in za vsak ustvarjen objekt spremlja število kazalcev, ki kažejo nanj) pobrisal, če bo program tekel dovolj dolgo. Z brisanjem objektov se nam v javi torej ni treba ukvarjati, saj se med delovanjem programa izvaja samodejno, ne da bi karkoli opazili.



**Slika 6.8** Delovanje operatorja `+=` pri nizih: spremeni se kazalec, ne niz.

#### Naloga 6.18 Napišite metodo

```
public static int[] frekvenceCrk(String niz)
```

ki vrne tabelo, v kateri element na indeksu *i* pove, kolikokrat se v podanem nizu pojavi črka, ki ima v angleški abecedi indeks *i*. Upoštevajte, da znaka `'A'` in `'a'` oba predstavljata črko A.

#### Naloga 6.19 Napišite metodo

```
public static String[] urejeneBesede(String niz)
```

ki vrne leksikografsko urejeno tabelo besed v podanem nizu. Vsaka beseda naj v izhodni tabeli nastopa samo po enkrat. Lahko predpostavite, da je niz sestavljen zgolj iz malih črk angleške abecede in presledkov.

**Naloga 6.20** S pomočjo javinega razreda `BigInteger` napišite program, ki prebere število  $n \in [0, 10^6]$  in izpiše Fibonaccijevo število  $F_n$ . (Spomnimo se:  $F_0 = F_1 = 1$ ,  $F_n = F_{n-2} + F_{n-1}$  (pri  $n \geq 2$ ).)

## 6.10 Razred za predstavitev vektorja

V tem razdelku bomo napisali razred za predstavitev *vektorja* — »raztegljive«  
tabele. Tak vsebovalnik je na voljo v večini sodobnih programskih jezikov, bodisi kot razred v standardni knjižnici (npr. `ArrayList` v javi ali `vector` v C++) bodisi kot vgrajen tip (npr. `list` v pythonu).

### 6.10.1 Razred `VektorInt`

Naš razred za predstavitev vektorja bo podoben razredu `ArrayList` v paketu `java.util`, vendar pa bo omogočal zgolj hranjenje elementov tipa `int`, zato ga bomo imenovali `VektorInt`. V razdelku 7.6 pa bomo napisali razred za vektor, ki bo omogočal hranjenje elementov poljubnih referenčnih tipov.

Tabele seveda ni mogoče raztezati; ko določimo njeno velikost, ta ostane taka, kot je, dokler tabela obstaja. Lahko pa raztezanje simuliramo tako, da ustvarimo novo, večjo tabelo in vanjo skopiramo elemente trenutne tabele. Tako bomo ravnali tudi mi.

Vsak objekt razreda `VektorInt` bo vseboval dva atributa:

- `int[] elementi`: Tabela, ki hrani elemente vektorja.
- `int stElementov`: Dejansko število elementov v tabeli `elementi`. To število je vedno manjše ali enako kapaciteti tabele (`elementi.length`).

Tabelo `elementi` bomo ustvarili z določeno začetno kapaciteto, dejansko število elementov v njej pa bo na začetku enako 0. Ko bo uporabnik v vektor dodajal elemente, se bo atribut `stElementov` seveda povečeval. Ko bo dosegel kapaciteto in bo uporabnik želel dodati še en element, bomo tabelo »raztegnili«. Uporabnik razreda `VektorInt` ne bo videl, da se je v ozadju ustvarila nova tabela; zanj bo objekt tega tipa deloval kot tabela, ki se lahko poljubno povečuje.

Ob izdelavi objekta, torej v konstruktorju, moramo izdelati tabelo elementov. Tabeli lahko načeloma določimo poljubno začetno kapaciteto, saj jo bomo po potrebi povečevali. Naj bo začetna kapaciteta enaka 10:

```
// koda 6.7 (razrediInObjekti/vektor/VektorInt.java)
public class VektorInt {
    private int[] elementi;
    private int stElementov;

    public VektorInt() {
        this.elementi = new int[10];    // (1)
        this.stElementov = 0;           // (2)
    }
    ...
}
```

Vrstica (2) je dejansko odveč, saj je privzeta vrednost atributa `stElementov` že tako ali tako enaka 0, kljub temu pa poveča jasnost. V vrstici (1) pa bo marsikaterega programerja zmotilo število 10. Z njim ni nič narobe, vendar pa načela lepega programiranja zahtevajo, da se izogibamo konkretnim vrednostim in da namesto tega uporabljamo *poimenovane konstante*. Konstante so opremljene z določilom `final`, ki doseže, da vsak poskus spremembe njihovih vrednosti povzroči napako pri prevajanju. Pristavili bomo tudi določilo `static`, saj je začetna kapaciteta lastnost celotnega razreda, ne vsakega vektorja posebej. Ker KONSTANTE\_PISEMO\_TAKOLE, bomo našo konstanto poimenovali `ZACETNA_KAPACITETA`:

```
public class VektorInt {
    private static final int ZACETNA_KAPACITETA = 10;

    private int[] elementi;
    private int stElementov;

    public VektorInt() {
        this.elementi = new int[ZACETNA_KAPACITETA];
        this.stElementov = 0;
    }
    ...
}
```

S poimenovano konstanto dosežemo dvoje: prvič, program je preglednejši, saj ime `ZACETNA_KAPACITETA` pove bistveno več kot število 10, in drugič, če ista konstanta nastopa na več različnih mestih, jo bistveno lažje spremenimo, če namesto njene vrednosti dosledno uporabljamo njeno ime.

Če definiramo še konstruktor, ki začetno kapaciteto sprejme kot svoj argument, lahko konstruktor brez parametrov poenostavimo:

```
public class VektorInt {
```

```

...
public VektorInt(int zacetnaKapaciteta) {
    this.elementi = new int[zacetnaKapaciteta];
    this.stElementov = 0;
}

public VektorInt() {
    this(ZACETNA_KAPACITETA);
}
...
}

```

Razred `VektorInt` ponuja več metod. Metoda `steviloElementov` vrne dejansko število elementov vektorja `this`, metoda `vrni` vrne element vektorja `this` na podanem indeksu, metoda `nastavi` pa element na podanem indeksu nastavi na podano vrednost:

```

public int steviloElementov() {
    return this.stElementov;
}

public int vrni(int indeks) {
    return this.elementi[indeks];
}

public void nastavi(int indeks, int vrednost) {
    this.elementi[indeks] = vrednost;
}

```

Metoda `nastavi` spremeni vrednost elementa, ki že obstaja, ne moremo pa je uporabiti za dodajanje elementov, saj je z vidika uporabnika začetna dolžina vektorja (tj. dejansko število elementov) enaka 0. V ta namen se poslužimo metode `dodaj`, ki podani element doda na konec vektorja, torej v celico z indeksom `this.stElementov`, in poveča dejansko število elementov. Pred tem preveri, ali je tabelo treba »povečati«, in to po potrebi tudi stori.

```

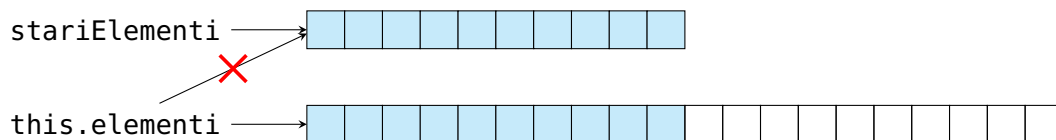
public void dodaj(int vrednost) {
    this.poPotrebiPovecaj();
    this.elementi[this.stElementov] = vrednost;
    this.stElementov++;
}

```

Če je dejansko število elementov v tabeli enako kapaciteti, potem metoda

poPotrebiPovecaj ustvari novo tabelo z dvakratno kapaciteto in vanjo skopira elemente prvotne tabele (slika 6.9). Ker ni mišljeno, da bi uporabnik metodo neposredno klical, jo deklariramo z dostopnim določilom private:

```
private void poPotrebiPovecaj() {
    if (this.stElementov >= this.elementi.length) {
        int[] stariElementi = this.elementi;
        this.elementi = new int[2 * stariElementi.length];
        for (int i = 0; i < this.stElementov; i++) {
            this.elementi[i] = stariElementi[i];
        }
    }
}
```



Slika 6.9 »Raztezanje« tabele.

Običajne tabele ne omogočajo vstavljanja in odstranjevanja (izločanja) elementov, v razredu VektorInt pa ni razloga, da takih metod ne bi ponudili. Element vstavimo tako, da vse elemente desno od mesta vstavljanja prestavimo za eno mesto v desno in zatem v »izpraznjeno« celico vpišemo podani element, pri odstranjevanju pa vse elemente desno od mesta vstavljanja prestavimo za eno mesto v levo. Tako kot pri dodajanju moramo tudi pri vstavljanju tabelo »povečati«, če je trenutno število elementov enako njeni kapaciteti.

```
public void vstavi(int indeks, int vrednost) {
    this.poPotrebiPovecaj();
    for (int i = this.stElementov - 1; i >= indeks; i--) {
        this.elementi[i + 1] = this.elementi[i];
    }
    this.elementi[indeks] = vrednost;
    this.stElementov++;
}

public void odstrani(int indeks) {
    for (int i = indeks; i < this.stElementov - 1; i++) {
        this.elementi[i] = this.elementi[i + 1];
    }
}
```

```

    this.stElementov--;
}

```

Skoraj vsak javanski razred ponuja metodo `toString`.<sup>7</sup> V razredu `VektorInt` bo metoda vrnila vsebino vektorja v obliki niza  $[e_0, e_1, \dots, e_{n-1}]$ .

```

public String toString() {
    String str = "[";
    for (int i = 0; i < this.stElementov; i++) {
        if (i > 0) {
            str += ", ";
        }
        str += Integer.toString(this.elementi[i]);
    }
    str += "]";
    return str;
}

```

Metoda `toString` deluje, vendar pa ni najbolj učinkovita, saj vsak stik nizov, kot smo videli v razdelku 6.9, ustvari nov niz. Ker to negativno vpliva na učinkovitost, si je bolje pomagati z razredom `StringBuilder`, ki predstavlja spremenljiv niz. Razred `StringBuilder` pravzaprav deluje podobno kot vektor znakov. Metoda `append` doda podani argument na konec objekta `this`, metoda `toString` pa vrne navaden niz z enako vsebino. Sledeča različica metode `toString` je zato bistveno učinkovitejša:

```

public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append("[");
    for (int i = 0; i < this.stElementov; i++) {
        if (i > 0) {
            sb.append(", ");
        }
        sb.append(this.elementi[i]);
    }
    sb.append("]");
    return sb.toString();
}

```

### Naloga 6.21 Razred `VektorInt` dopolnite z metodo

```
VektorInt podvektor(int zacetek, int konec)
```

<sup>7</sup>Kot bomo videli v poglavju 7, jo v resnici vsebuje vsak razred.

ki ustvari in vrne vektor, ki vsebuje elemente vektorja `this` z indeksi od začetka do vključno `konec - 1`. Vrnjeni vektor naj ima enako kapaciteto kot vektor `this`.

**Naloga 6.22** Metode razreda `VektorInt` ne preverjajo svojih argumentov. Na primer, sledeča koda izpiše vrednost 42, v resnici pa bi morala sprožiti izjemo, saj novoustvarjeni vektor še nima nobenega elementa in zato tudi nima nobenega veljavnega indeksa:

```
VektorInt vektor = new VektorInt();
vektor.nastavi(5, 42);
System.out.println(vektor.vrni(5));
```

Dopolnite metode tako, da bodo v primeru neveljavnih indeksov sprožile izjemo tipa `IndexOutOfBoundsException`. Izjemo sprožite s stavkom

```
throw new IndexOutOfBoundsException();
```

### 6.10.2 Razred `VektorString`

Objekt razreda `VektorInt` lahko hrani elemente tipa `int`. Z minimalnimi popravki lahko napišemo, denimo, razred `VektorString`. Njegovi objekti lahko hranijo elemente tipa `String`:

```
// koda 6.8 (razrediInObjekti/vektor/VektorString.java)
public class VektorString {
    private static final int ZACETNA_KAPACITETA = 10;

    private String[] elementi;
    private int stElementov;

    public VektorString(int zacetnaKapaciteta) {
        this.elementi = new String[zacetnaKapaciteta];
        this.stElementov = 0;
    }

    public VektorString() {
        this(ZACETNA_KAPACITETA);
    }

    public int steviloElementov() {
        return this.stElementov;
    }
}
```

```

public String vrni(int indeks) {
    return this.elementi[indeks];
}

public void nastavi(int indeks, String vrednost) {
    this.elementi[indeks] = vrednost;
}

public void dodaj(String vrednost) {
    this.poPotrebiPovecaj();
    this.elementi[this.stElementov] = vrednost;
    this.stElementov++;
}

public void vstavi(int indeks, String vrednost) {
    this.poPotrebiPovecaj();
    for (int i = this.stElementov - 1; i >= indeks; i--) {
        this.elementi[i + 1] = this.elementi[i];
    }
    this.elementi[indeks] = vrednost;
    this.stElementov++;
}

public void odstrani(int indeks) {
    for (int i = indeks; i < this.stElementov - 1; i++) {
        this.elementi[i] = this.elementi[i + 1];
    }
    this.stElementov--;
}

private void poPotrebiPovecaj() {
    if (this.stElementov == this.elementi.length) {
        String[] stariElementi = this.elementi;
        this.elementi = new String[2 * stariElementi.length];
        for (int i = 0; i < this.stElementov; i++) {
            this.elementi[i] = stariElementi[i];
        }
    }
}

public String toString() {

```



```

        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (int i = 0; i < this.stElementov; i++) {
            if (i > 0) {
                sb.append(", ");
            }
            sb.append(this.elementi[i]);
        }
        sb.append("]");
        return sb.toString();
    }
}

```

Seveda bi lahko na podoben način izdelali tudi razred za vektorje z elementi tipa `Cas`, razred za vektorje z elementi tipa `Ulomak` itd. Najbrž pa ni treba posebej poudarjati, da je izdelava ločenega razreda za vsak tip elementov precej nesmotrno početje. V poglavju 8 bomo videli, da lahko z drobno spremembo razred predelamo tako, da bomo lahko tip elementov vektorja podali kar kot argument pri njegovi izdelavi.

### 6.10.3 Primer uporabe

Na vektor lahko gledamo kot na tabelo z dodatnimi funkcionalnostmi (»raztezanje«, vstavljanje in odstranjevanje), treba pa se je zavedati, da so te funkcionalnosti potencialno neučinkovite, saj lahko vključujejo kopiranje ali premikanje velikega števila elementov. Preden jih uporabimo, se nam splača premisliti, ali jih sploh potrebujemo. No, v naslednjem primeru nam bodo prišle prav.

**Primer 6.1.** V ravni vrsti stoji  $n$  otrok. Igrajo se izštevanko, dokler ne ostane samo eden. V vsakem krogu izštevanke izpade  $k$ -ti otrok od leve proti desni. Če pridemo do konca vrste, preden naštejemo  $k$  otrok, se vrnemo na začetek vrste in nadaljujemo s štetjem. Na primer, če je  $n = 5$ , bo pri  $k = 4$  (in tudi  $k = 9$ ,  $k = 14$  itd.) izpadel četrti otrok.

Napišimo program, ki prebere števili  $n$  in  $k$  ter imena posameznih otrok (vsako ime je sestavljeno iz ene same besede) in za vsak krog izštevanke izpiše, kdo v njem izpade. Na primer, za vhod

```
5 9 Ana Bojan Cvetka Denis Eva
```

naj program izpiše sledeče:

```

Denis
Ana
Eva

```

## Bojan

*Rešitev.* Najprej preberemo imena otrok, nato pa vstopimo v zanko, ki teče po krogih izštevanke. Za vsak krog izštevanke določimo, kdo izpade, izpišemo njegovo ime in ga izločimo. Imena otrok bomo hranili v vektorju tipa `VektorString`, saj jih bomo tako najlažje izločali.

Za branje nizov, sestavljenih iz ene besede, lahko nad objektom tipa `Scanner` pokličemo metodo `next`. Indeks otroka, ki izpade, najlažje določimo s pomočjo ostanka pri deljenju: če je  $o$  trenutno število otrok v vrsti, potem je indeks izpadlega enak  $(k - 1) \bmod o$ .

```
// koda 6.9 (razrediInObjekti/vektor/Izstevanka.java)
import java.util.Scanner;

public class Izstevanka {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int stOtrok = sc.nextInt();
        int stBesed = sc.nextInt();

        // preberi imena otrok v vektor
        VektorString otroci = new VektorString();
        for (int i = 0; i < stOtrok; i++) {
            otroci.dodaj(sc.next());
        }

        // število krogov izštevanke
        int stKrogov = stOtrok - 1;

        // simuliraj izštevanko
        for (int krog = 1; krog <= stKrogov; krog++) {
            // ugotovi, kdo izpade
            int ixIzpadlega = (stBesed - 1) % stOtrok;
            System.out.println(otroci.vrni(ixIzpadlega));

            // izloči izpadlega
            otroci.odstrani(ixIzpadlega);
            stOtrok--;
        }
    }
}
```



## 6.11 Povzetek

- Objekt je skupek spremenljivk, ki niso nujno istega tipa. Spremenljivke (imenujemo jih atributi) so dostopne prek imen.
- Spremenljivka objektnega tipa ne vsebuje objekta, ampak kazalec nanj.
- Razred je opis zgradbe in obnašanja objektov. Razred določa tip: tip objekta razreda *R* je enak *R*.
- Razred podaja attribute, ki jih vsebujejo posamezni objekti, metode, ki jih lahko nad objekti izvajamo, in konstruktorje, s katerimi lahko inicializiramo pravkar ustvarjene objekte.
- Vsak element razreda (atribut, metodo ali konstruktor) lahko opremimo z dostopnim določilom. Določilo `private` v razredu *R* pove, da je element neposredno (kot `objekt.element` pri nestatičnih oziroma `Razred.element` pri statičnih elementih) dostopen samo v okviru razreda *R*. Element brez določil je neposredno dostopen v razredu *R* in v vseh razredih v istem paketu. Element z določilom `protected` je poleg tega neposredno dostopen še v podrazredih razreda *R*. Določilo `public` pove, da je element neposredno dostopen od koderkoli.
- Objekt ustvarimo z operatorjem `new`. Ta nastavi objektove attribute na privzete vrednosti, nato pa pokliče konstruktor in mu poleg morebitnih argumentov posreduje še kazalec na pravkar ustvarjeni objekt, da bo lahko nastavil vrednosti njegovih atributov. Taisti kazalec je tudi rezultat izraza, ki ga prične operator `new`.
- Če v razredu ne napišemo lastnega konstruktorja, potem prevajalnik vanj doda konstruktor brez parametrov, ki vsebuje zgolj klic konstruktorja nadrazreda.
- Metode so lahko statične ali nestatične. Statične metode ob klicu prejmejo natanko toliko vrednosti, kot imajo parametrov. Pokličemo jih kot `Razred.metoda(...)`. Ob klicu nestatične metode (`objekt.metoda(...)`) se spremenljivka *objekt* (kazalec na objekt) skopira v spremenljivko `this`, ki deluje kot dodaten parameter metode.
- Spremenljivka `this` v nestatičnih metodah kaže na objekt, nad katerim smo metodo poklicali, v konstruktorjih pa na pravkar izdelani objekt. V statičnih metodah beseda `this` nima pomena.
- Tudi atributi so lahko statični ali nestatični. Nestatični atributi pripadajo posameznim objektom (vsak objekt ima svoje), statični pa razredu kot celoti.

- V istem razredu lahko imamo več metod z istim imenom, če se med seboj razlikujejo po podpisu (kombinaciji imena in seznama tipov parametrov). Enako velja za konstruktorje.
- Vektor je vsebovalnik, ki se obnaša kot raztegljiva tabela. »Raztezanje« tabele, ki hrani elemente vektorja, realiziramo preprosto tako, da elemente prepisemo v novo, večjo tabelo.

### ***Iz profesorjevega kabineta***

»Profesor Doberšek, pravkar berem nek članek in ne vem, kaj pomeni *metaprogramiranje*,« svojega nadrejenega spoštljivo ogovori asistent Slapšak.

»Metaprogramiranje ... Kdo bi vedel! No, imel sem teto po imenu Meta. Na vrtu je vedno imela obilico mete, čeprav si z njo ni znala prav dosti pomagati. Da bi tale moja teta Meta na veliko programirala, pa ne bi mogel z gotovostjo trditi. Tudi nje po mojem niso programirali. Je bila vse premalo pokorna za kaj takega.«

»Metaprogram je program, ki obdeluje druge programe. Prevajalnik, denimo,« se izza vogala zasliši glas docentke Javornik.

»Java, mimogrede,« nadaljuje docentka, »nudi izvrstno podporo metaprogramiranju. Zlahka pridobiš, recimo, seznam metod in atributov podanega razreda. Oglej si paket `java.lang.reflect`, in videl boš! Prični z razredom `Class`.«

Asistent je že povsem zbegan. »Kako, prosim?!« mukoma izdavi, a sliši le še nekaj takega kot »močno je, zato nikar ne zlorablaj,« in docentke že ni več.

Napišite in preizkusite metodo

```
public static void izpisiAttribute(String razred)
```

ki izpiše imena, določila in tipe vseh atributov v razredu s podanim imenom. Lahko predpostavite, da so atributi opremljeni samo z določili `public`, `private`, `static` in/ali `final`. Na primer, za razred

```
public class MojRazred {
    private int dolzina;
    public static boolean jeRes;
    private final String naslov;
    private static float[][] podatki;
    public static final int NACIN_IZPISA = 42;

    public MojRazred(String mojNaslov) {
        this.naslov = mojNaslov;
    }
}
```

naj klic `izpisiAttribute("MojRazred")` izpiše sledeče:

```
dolzina <določila = [private], tip = int>  
jeRes <določila = [public, static], tip = boolean>  
naslov <določila = [private, final], tip = java.lang.String>  
podatki <določila = [private, static], tip = float[][]>  
NACIN_IZPISA <določila = [public, static, final], tip = int>
```

Razred `MojRazred` se mora nahajati v istem imeniku kot vaš program (razred, ki vsebuje metodo `izpisiAttribute`), poleg tega pa ga morate pred zagonom programa prevesti.



## 7 Dedovanje

Dedovanje je relacija med razredi, ki izhaja iz relacije »je poseben primer« (npr. izredni študent je poseben primer študenta). Če je razred *B* (npr. *IzredniStudent*) *podrazred* (»dedič«) razreda *A* (npr. *Student*), potem razred *B* podeduje od razreda *A* vse nestatične attribute (npr. ime in priimek, vpisno številko in mesečne stroške bivanja) in vse nestatične metode razen privatnih, lahko pa vsebuje še svoje lastne elemente (npr. atribut, ki podaja šolnino). Razred *B* lahko posamezne metode tudi *redefinira*, saj se lahko določeni postopki (npr. izračun mesečnih stroškov) za objekte podrazreda izvajajo drugače kot za objekte nadrazreda. Če je razred *B* podrazred razreda *A*, potem lahko z objektom tipa *B* počnemo vse tisto, kar lahko počnemo z objektom tipa *A*, zato nam dedovanje omogoča poenoteno obravnavo objektov različnih podrazredov istega razreda.

### 7.1 Uvod

V vsakdanjem življenju pogosto naletimo na situacijo, ko je koncept *B* poseben primer koncepta *A*. Na primer, izredni študent je poseben primer študenta. To pomeni, da imajo izredni študentje vse lastnosti (attribute), ki jih imajo tudi redni (ime, priimek, vpisno številko, predmetnik ...), poleg tega pa imajo morda še kakšne lastne (npr. šolnino). Vrstna hiša je poseben primer hiše, ta pa je poseben primer nepremičnine. Kvadrat je poseben primer pravokotnika, ta je poseben primer štirikotnika, ta pa je poseben primer geometrijskega lika: štirikotnik je geometrijski lik s štirimi stranicami, pravokotnik je štirikotnik, v katerem vsi koti merijo 90 stopinj, kvadrat pa je pravokotnik, v katerem so vse štiri stranice enako dolge. Še več primerov opisane relacije najdemo v naravi: pes je sesalec, sesalec je vretenčar, vretenčar pa žival. Psi imajo vse tisto, kar imajo sesalci, poleg tega pa še svoje posebnosti. Za sesalce velja vse, kar velja za vretenčarje, pa še kaj drugega. In tako naprej.

Relacijo »*B* je poseben primer *A*« v javi izrazimo z relacijo med razredi, ki ji pravimo *dedovanje*. Dedovanje nakažemo z rezervirano besedo *extends*:

```
public class A {  
    ...  
}
```

```
public class B extends A {
    ...
}
```

Razred *B* je *podrazred* razreda *A*. To pomeni, da razred *B* samodejno, ne da bi nam bilo treba še kaj storiti, od razreda *A* podeduje vse nestatične attribute (*tudi privatne*) in vse nestatične metode *razen* *privatnih*. Konstruktorji se *ne* dedujejo. Razred *B* lahko obogatimo še z dodatnimi atributi in metodami, posamezne metode razreda *A* pa lahko tudi *redefiniramo* (ponovno definiramo).

Če je razred *B* podrazred razreda *A*, je razred *A* *nadrazred* razreda *B*. Lahko tudi rečemo, da je razred *B* *izpeljan* iz razreda *A* ali da razred *B* *razširja* razred *A*. Ker vsak razred določa tip, pravimo tudi, da je tip *B* *podtip* tipa *A*, tip *A* pa *nadtip* tipa *B*.

Relacija nadrazred-podrazred oziroma nadtip-podtip je tranzitivna: če je razred *B* podrazred razreda *A*, razred *C* pa podrazred razreda *B*, je tudi razred *C* podrazred razreda *A*. Kadar bomo želeli razlikovati med razredoma *B* in *C*, bomo rekli, da je razred *B* *neposredni*, razred *C* pa *posredni* podrazred razreda *A*.

**Naloga 7.1**    Narišite drevo, ki ponazarja smiselno hierarhijo razredov Avto, Avtobus, PotniskiVlak, PotniskoVozilo, TovorniVlak, Tovornjak, TovornoVozilo in Vozilo. Katere attribute bi lahko imeli posamezni razredi?

## 7.2 Hierarhija študentov

### 7.2.1 Nadrazred

Oglejmo si preprost razred. Njegovi objekti predstavljajo posamezne študente:

```
// koda 7.1 (dedovanje/studentje/Student.java)
public class Student {
    private String ip;
    private String vpisna;
    private int stroskiBivanja;

    public Student(String ip, String vpisna, int stroskiBivanja) {
        this.ip = ip;
        this.vpisna = vpisna;
        this.stroskiBivanja = stroskiBivanja;
    }

    public String vrniIP() {
        return this.ip;
    }
}
```



```

    public int stroski() {
        return this.stroskiBivanja;
    }
}

```

Vsak študent ima svoje ime in priimek (atribut `ip`, npr. Janez Novak), svojo vpisno številko (atribut `vpisna`) in mesečne stroške bivanja (atribut `stroskiBivanja`). Konstruktor nastavi vrednosti vseh treh atributov novoustvarjenega objekta na vrednosti, ki jih dobi prek parametrov.<sup>1</sup> Metoda `vrniIP` je »getter« za atribut `ip`, metoda `stroski` pa vrne skupne študentove mesečne stroške. Ti so enaki kar stroškom bivanja.

### 7.2.2 Podrazred

Iz razreda `Student` izpeljimo razred `IzredniStudent`:

```
public class IzredniStudent extends Student
```

Razred `IzredniStudent` torej od razreda `Student` podeduje attribute `ip`, `vpisna` in `stroskiBivanja` ter metodi `vrniIP` in `stroski`. Poleg podedovanih atributov bomo za vsakega izrednega študenta hranili tudi mesečne stroške šolnine (atribut `solnina`). Definirali bomo tudi konstruktor; ta se namreč ne podeduje. Metoda `stroski` se sicer podeduje, vendar pa jo bomo *redefinirali*, saj se mesečni stroški za izrednega študenta izračunajo drugače kot za splošnega študenta.

```

// koda 7.2 (dedovanje/studentje/IzredniStudent.java)
public class IzredniStudent extends Student {
    // dodatni atribut
    private int solnina;

    // konstruktor
    public IzredniStudent(...) {
        ...
    }

    // redefinirana metoda
    @Override
    public int stroski() {
        ...
    }
}

```

<sup>1</sup>Vidimo, da so parametri konstruktorja enaki imenom atributov. To je precej običajna praksa. Seveda pa si v tem primeru v konstruktorju ne moremo privoščiti, da bi besedo `this` izpustili, saj brez nje ni mogoče razlikovati med atributi in parametri.

```
}
```

### 7.2.3 Konstruktor podrazreda

Konstruktor razreda `IzredniStudent` lahko ima poljubno število parametrov, toda če želimo ob inicializaciji svobodno nastavljati vrednosti vseh atributov, potrebujemo štiri parametre, po enega za vsak atribut. Poskusimo takole:

```
public IzredniStudent(String ip, String vpisna,
                      int stroskiBivanja, int solnina) {
    this.ip = ip;
    this.vpisna = vpisna;
    this.stroskiBivanja = stroskiBivanja;
    this.solnina = solnina;
}
```

Poskus se žal izjalovi, saj nam prevajalnik med drugim sporoči te napake:

```
IzredniStudent.java:8: error: ip has private access in Student
    this.ip = ip;
        ^
IzredniStudent.java:9: error: vpisna has private access in Student
    this.vpisna = vpisna;
        ^
IzredniStudent.java:10: error: stroskiBivanja has private access in
                        Student
    this.stroskiBivanja = stroskiBivanja;
        ^
```

Čeprav so atributi `ip`, `vpisna` in `stroskiBivanja` sestavni del vsakega objekta tipa `IzredniStudent`, do njih znotraj razreda `IzredniStudent` ne moremo dostopati, saj so v razredu `Student` deklarirani kot privatni.

Ena od možnih idej za rešitev nastale zagate je sprememba dostopnih določil atributov, deklariranih v razredu `Student`. Ni treba, da atributom dodelimo določilo `public`; zadošča že določilo `protected`, ki omogoči dostop iz vseh podrazredov. Kljub temu pa takih posegov brez tehtnih razlogov naj ne bi izvajali.

Namesto spreminjanja dostopnih določil je bolje, da si v konstruktorju razreda `IzredniStudent` pomagamo s konstruktorjem razreda `Student`. To storimo s pomočjo stavka `super(...)`. Stavek `super(...)` pokliče konstruktor neposrednega nadrazreda in mu posreduje argumente, nanizane med parom oklepajev. V našem primeru bo konstruktor razreda `Student` inicializiral attribute `ip`, `vpisna` in `stroskiBivanja`, konstruktor razreda `IzredniStudent` pa bo inicializiral še atribut `solnina`.

```
public IzredniStudent(String ip, String vpisna,
                    int stroskiBivanja, int solnina) {
    super(ip, vpisna, stroskiBivanja);
    this.solnina = solnina;
}
```

Prikazana »delitev dela« je pravzaprav edina smiselna možnost. V javi namreč velja pravilo, da mora biti prvi stavek konstruktorja klic `super(...)` ali `this(...)`. Če ta klic izpustimo, prevajalnik sam vstavi klic `super()` brez argumentov. To bi v našem primeru pomenilo, da se pokliče tisti konstruktor v razredu `Student`, ki nima parametrov. Vendar pa takega konstruktorja ni, saj smo ga »povežili« z lastnim konstruktorjem s parametri. Zato prevajalnik pri prvotni (napačni) različici konstruktorja razreda `IzredniStudent` poleg napak v zvezi z dostopom do privatnih atributov sporoči tudi to, da ne najde brezparametrskega konstruktorja razreda `Student`:

```
IzredniStudent.java:7: error: constructor Student in class Student
                        cannot be applied to given types;
    public IzredniStudent(String ip, String vpisna,
                        int stroskiBivanja, int solnina) {
                                                ^
required: String,String,int
found:    no arguments
reason: actual and formal argument lists differ in length
```

**Naloga 7.2** Ker so študentje tudi osebe, definirajte še razred `Oseba`, in to tako, da bo razred `Student` njegov podrazred. Ustrezno prilagodite razred `Student` in po potrebi tudi razred `IzredniStudent`.

**Naloga 7.3** Podana sta razreda `A` in `B`:

```
public class A {
    public A() {
        System.out.println("Dober dan!");
    }
}

public class B extends A {
    public static void main(String[] args) {
        new B();
    }
}
```

Kaj se zgodi, če prevedemo in poženemo razred B? Kaj bi se zgodilo, če bi v razred B dodali konstruktor brez parametrov, ki vsebuje zgolj stavek za izpis niza Dober večer!?

#### 7.2.4 Redefinicija metode

Metodo stroški bomo v razredu IzredniStudent redefinirali. Redefinicije metode se v splošnem lotimo takole:

```
class A {
    določilo T metoda( $T_1 p_1, \dots, T_n p_n$ ) { ... }
}

class B extends A {
    @Override
    določilo'  $T'$  metoda( $T_1 p_1, \dots, T_n p_n$ ) { ... }
}
```

Redefinirana metoda mora v podrazredu imeti enako ime in enako zaporedje tipov parametrov kot v nadrazredu. Izhodni tip metode v podrazredu (tip  $T'$ ) mora biti enak izhodnemu tipu metode v nadrazredu (tipu  $T$ ), novejša različica jave pa dopušča tudi možnost, da je tip  $T'$  podtip tipa  $T$ . Dostopno določilo metode v podrazredu mora biti enako ali ohlapnejše kot v nadrazredu (na primer, lahko je metoda v nadrazredu deklarirana kot `protected`, v podrazredu pa kot `public`). Priporočljivo je, da pred glavo redefinirane metode v podrazredu dodamo oznako `@Override`. Ta oznaka zmanjša možnost napake, saj prevajalniku pove, da gre za redefinirano metodo, zato bo ta preveril, ali v nadrazredu obstaja metoda z ujemajočo glavo, in javil napako, če je ne bo.

Če želimo znotraj redefinirane metode v podrazredu poklicati različico te metode iz nadrazreda, storimo to s klicem `super.metoda(...)`. Za razliko od klica konstruktorja nadrazreda lahko klic `super.metoda(...)` izvedemo poljubno mnogokrat (tudi nobenkrat) in kjerkoli znotraj metode.

Mesečni stroški študija se za izrednega študenta izračunajo tako, da se stroškom, izračunanem po metodi za splošnega študenta, prišteje še šolnina:

```
public class IzredniStudent {
    ...
    @Override
    public int stroški() {
        return super.stroški() + this.solnina;
    }
}
```

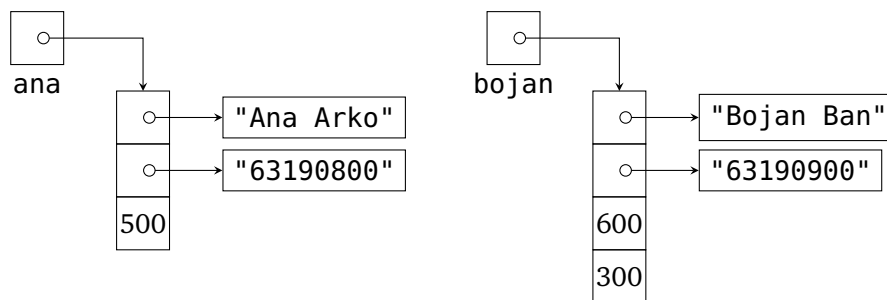
Klic `super.stroski()` pokliče metodo `stroski` iz razreda `Student`. Ta klic se izvede natanko tako kot vsi drugi klici nestatičnih metod: kazalec `this`, ki ga prejme metoda `stroski` v razredu `IzredniStudent`, se ob klicu `super.stroski()` skopira v kazalec `this` znotraj metode `stroski` v razredu `Student`. Metoda `stroski` v nadrazredu zato deluje nad istim objektom kot metoda `stroski` v podrazredu.

Razreda bomo preizkusili tako, da bomo izdelali objekt tipa `Student` in objekt tipa `IzredniStudent` in nad vsakim poklicali metodi `vrniIP` in `stroski`. Kot lahko vidimo, se pri računanju mesečnih stroškov za izrednega študenta upoštevajo tako stroški bivanja kot šolnina:

```
public class TestStudent {
    public static void main(String[] args) {
        Student ana = new Student("Ana Arko", "63190800", 500);
        System.out.println(ana.vrniIP());    // Ana Arko
        System.out.println(ana.stroski());   // 500

        IzredniStudent bojan = new IzredniStudent(
            "Bojan Ban", "63190900", 600, 300);
        System.out.println(bojan.vrniIP());  // Bojan Ban
        System.out.println(bojan.stroski()); // 900
    }
}
```

Slika 7.1 prikazuje objekta, ki smo ju izdelali v metodi `main`.



Slika 7.1 Objekt tipa `Student` in objekt tipa `IzredniStudent`.

**Naloga 7.4** Koliko atributov, konstruktorjev in metod vsebuje razred `B`?

```
public class A {
    private int a;
    protected double b;
    public String c;
```

```

    public A() {}
    public A(int a) {}

    private void f() {}
    protected void g() {}
    public void h() {}
}

public class B extends A {
    private A d;
    protected B e;

    private void k() {}
    @Override public void g() {}
}

```

**Naloga 7.5** Iz razreda *Oseba* z atributoma *ime* in *priimek* izpeljite razred *OsebaZNazivi*, ki ima dodaten atribut *nazivi* (npr. prof. dr.). V razredu *Oseba* definirajte metodo

```
public String toString()
```

in to tako, da bo vrnila niz, sestavljen iz imena osebe *this*, presledka in priimka osebe *this* (npr. Janez Novak). V razredu *OsebaZNazivi* jo redefinirajte tako, da bo pred ime in priimek postavila še nazive (npr. prof. dr. Janez Novak).

**Naloga 7.6** Podani so sledeči razredi:

```

public class A {
    private int p;

    public A(int p) {
        this.p = p;
    }

    public int f(int a) {
        return a * this.p;
    }
}

public class B extends A {

```

```

public B(int p) {
    super(p);
}

@Override
public int f(int a) {
    return super.f(a) * super.f(a);
}
}

public class C extends B {
    public C(int p) {
        super(p);
    }

    @Override
    public int f(int a) {
        return super.f(a) * super.f(a);
    }
}

```

Kaj izpiše sledeči stavek? Kaj bi izpisal, če bi iz razreda C odstranili metodo `f`? Kaj bi izpisal, če bi metodo `f` odstranili iz razreda B, in kaj, če bi jo iz obeh?

```
System.out.println(new C(4).f(3));
```

### 7.2.5 Tip kazalca in tip objekta

Če je razred *B* podrazred razreda *A*, potem je koncept, ki ga predstavlja razred *B*, poseben primer koncepta, ki ga predstavlja razred *A*. Na primer, vsak izredni študent je tudi študent, obratno pa ni nujno. Ta pogled vpliva na priredljivost tipov: vrednost (kazalec) tipa *B* lahko priredimo spremenljivki tipa *A*, obratno pa ni mogoče. Na primer:

```

Student a = new Student(...);
IzredniStudent b = new IzredniStudent(...);
Student p = b;           // OK
IzredniStudent q = a;    // napaka pri prevajanju

```

Pravilo se odraža tudi pri klicih metod. Metodi, ki sprejme parameter tipa *A*, lahko podamo tudi kazalec tipa *B*, ne da bi se prevajalnik pritožil, saj se bo ob klicu metode kazalec tipa *B* lahko skopiral v parameter tipa *A*. Podobno velja pri vračanju: metoda

z izhodnim tipom *A* lahko vrne tudi kazalec tipa *B*.

Sprostitev togega pravila o priredljivosti tipov ima zanimive posledice. Najpomembnejša je ta, da *tip kazalca ni nujno enak tipu objekta, na katerega kaže*. Kazalec tipa *A* (npr. *Student*) lahko namreč kaže tudi na objekt tipa *B* (npr. *IzredniStudent*). Na primer, po izvedbi stavkov

```
// koda 7.3
Student a = new Student(...);
IzredniStudent b = new IzredniStudent(...);
Student p = b;
```

je kazalec (spremenljivka) *p* tipa *Student*, kaže pa na objekt tipa *IzredniStudent*, medtem ko sta tipa kazalcev *a* in *b* enaka tipoma objektov, na katera kažeta.

Ločitev med tipom kazalca in tipom objekta vpliva tudi na obnašanje prevajalnika in izvajalnika:

- *Prevajalnik* pozna zgolj tipe *kazalcev*, saj vidi le *deklaracije* tipov kazalcev. Z njegovega vidika je pomembno samo to, da je kazalec *p* deklariran kot spremenljivka tipa *Student*, tip objekta, na katerega dejansko kaže, pa ga ne zanima. (V resnici ga niti ne more, saj v času prevajanja v splošnem ni jasno, kam kaže določeni kazalec.)
- *Izvajalnik* pozna zgolj tipe *objektov*. Z njegovega vidika tip kazalca *p* ni pomemben; šteje zgolj tip objekta, na katerega kaže. Ker smo v kodi 7.3 objekt, na katerega kaže kazalec *p*, ustvarili z izrazom `new IzredniStudent(...)`, izvajalnik točno ve, da *p* kaže na objekt tipa *IzredniStudent*.

Namesto »tip objekta, na katerega kaže kazalec *p*« bomo pogosto pisali kar »tip objekta *p*«. Kot rečeno, sta tipa kazalca in objekta lahko enaka, lahko pa je tip objekta *p* podtip tipa kazalca *p*. Drugih možnosti ni.

Pomembno se je zavedati tudi naslednjega:

- Tip kazalca je vseskozi enak, saj je določen s (fiksno) deklaracijo. Tipa objekta prav tako ni mogoče spremeniti, saj ga določimo ob njegovi izdelavi (z operatorjem `new` in klicem konstruktorja). Lahko pa isti kazalec v času svojega obstoja kaže na objekte različnih tipov. Na primer, s kazalcem tipa *Student* lahko najprej pokažemo na objekt tipa *IzredniStudent*, nato pa na objekt tipa *Student*.
- Izraz `new R(...)` ustvari objekt tipa *R* in vrne kazalec tipa *R*. Če je razred *B* podrazred razreda *A*, bo prevajalnik stavek

```
A p = new B(...);
```



torej razumel takole: »vrednost (kazalec) tipa *B* priredi spremenljivki tipa *A*«. Ker je tip *B* podtip tipa *A*, bo stavek prevedel brez pritožb. Kaj pa izvajalnik? Ustvaril bo objekt tipa *B* in kazalec nanj shranil v spremenljivko *p*, njen tip pa ga ne bo zanimal.

Če tip kazalca *p* ni enak tipu objekta, na katerega kaže, ni povsem jasno, katera metoda se bo poklicala pri klicu *p.metoda(...)*. Tista iz razreda, ki mu pripada kazalec *p*, ali tista iz razreda, ki mu pripada objekt, na katerega kaže kazalec *p*? Na primer, katera metoda *stroski* se bo poklicala v sledečem primeru? Tista iz razreda *Student* ali tista iz razreda *IzredniStudent*?

```
Student ana = new Student("Ana Arko", "63190800", 500);
IzredniStudent bojan = new IzredniStudent(
    "Bojan Ban", "63190900", 600, 300);
Student s = bojan;
System.out.println(s.stroski());
```

Pravilo je takšno: pri klicu *p.metoda(...)* se pokliče metoda *metoda* iz razreda, ki mu pripada objekt, na katerega kaže kazalec *p*, ne metoda iz razreda, ki mu pripada kazalec *p*. Ker kazalec *s* v trenutku, ko se izvaja klic metode, kaže na objekt tipa *IzredniStudent*, se bo poklicala metoda *stroski* iz razreda *IzredniStudent*, ne tista iz razreda *Student*. Gornji izsek kode bo torej izpisal število 900, ne 600.

**Naloga 7.7** Naj bo razred *Zival* nadrazred razredov *Pes* in *Macka*. Za vsakega od kazalcev *z1*, *z2*, *z3*, *z4* in *z5* podajte njegov tip in tip objekta, na katerega kaže po izvedbi sledeče kode:

```
Zival z1 = new Zival(...);
Pes z2 = new Pes(...);
Zival z3 = new Macka(...);
Zival z4 = z1;
z1 = z2;
Zival z5 = z1;
```

**Naloga 7.8** Razreda *D* in *C* sta podrazreda razreda *B*, ta pa je podrazred razreda *A*. Kazalci *a*, *b*, *c* in *d* so po vrsti tipa *A*, *B*, *C* oziroma *D*. Katera od teoretično dvanajstih možnih netrivialnih medsebojnih prirejanj (*a* = *b*, *a* = *c*, *a* = *d*, *b* = *a*, *b* = *c* itd.) se prevedejo?

**Naloga 7.9** Podana sta sledeča razreda:

```
public class A {
    public void f() {
```

```

        this.g();
        this.h();
    }
    protected void g() {
        System.out.println("A.g");
    }
    private void h() {
        System.out.println("A.h");
    }
}

public class B extends A {
    protected void g() {
        System.out.println("B.g");
    }
    private void h() {
        System.out.println("B.h");
    }
}

```

Kaj izpiše naslednja koda?

```

new A().f();
new B().f();

```

**Naloga 7.10** Pri klicu metode se tip *objekta* upošteva samo pri navideznem argumentu, ki se metodi posreduje kot kazalec `this`, medtem ko pri argumentih, nanizanih znotraj oklepajev, šteje zgolj tip *kazalca*. To pravilo nam sicer lahko meša štrene le pri istoimenskih metodah z enakim številom parametrov, ki so za nameček še sorodnih tipov, kljub temu pa se ga je dobro zavedati, da se izognemo morebitnim presenečenjem.

Podana sta sledeča razreda:

```

public class A {
    public void f(A a) {
        System.out.println("A/A");
    }
    public void f(B b) {
        System.out.println("A/B");
    }
}

```

```
public class B extends A {
    @Override
    public void f(A a) {
        System.out.println("B/A");
    }
    @Override
    public void f(B b) {
        System.out.println("B/B");
    }
}
```

Kaj izpiše naslednja koda?

```
A a1 = new A();
A a2 = new B();
B b = new B();
a1.f(a1);
a1.f(a2);
a1.f(b);
a2.f(a1);
a2.f(a2);
a2.f(b);
b.f(a1);
b.f(a2);
b.f(b);
```

### 7.2.6 Dedovanje in tabele

V poglavju 5 smo povedali, da je tabela zaporedje spremenljivk istega tipa. Ta definicija drži kot pribita — tako za tabele z elementi primitivnih tipov kot za tiste z elementi referenčnih tipov. Vendar pa moramo pri slednjih upoštevati možnost, da kazalci v tabeli kažejo na objekte različnih tipov. Kot vemo, lahko kazalec tipa *R* kaže na objekt tipa *R* ali poljubnega podtipa tipa *R*. Na primer, kazalci v tabeli tipa `Student[]` lahko kažejo na objekte tipa `Student` ali `IzredniStudent`:

```
Student[] studentje = {
    new Student("Ana Arko", "63190800", 500),
    new IzredniStudent("Bojan Ban", "63190900", 600, 300),
    new IzredniStudent("Cvetka Cevc", "63191000", 400, 350),
    new Student("Denis Denk", "63191100", 450)
};
```

Ker smo tabelo izdelali kot tabelo tipa `Student[]`, so vsi njeni elementi (tj. kazalci) tipa `Student`, vendar pa kazalca `studentje[1]` in `studentje[2]` kažeta na objekta tipa `IzredniStudent`, ne `Student`. Kodo bi lahko napisali tudi takole:

```
Student[] studentje = new Student[4];
studentje[0] = new Student("Ana Arko", "63190800", 500);
studentje[1] = new IzredniStudent("Bojan Ban", "63190900", 600, 300);
studentje[2] = new IzredniStudent("Cvetka Cevc", "63191000",
                                400, 350);
studentje[3] = new Student("Denis Denk", "63191100", 450);
```

Kaj se zgodi, če za vsak element tabele pokličemo metodo `stroski`? Ker je izbira ciljne metode določena s tipom objekta, se bo za »navadne« študente (objekte tipa `Student`, torej za Ano in Denisa) poklicala metoda iz razreda `Student`, za izredne študente (objekte tipa `IzredniStudent`, torej za Bojana in Cvetko) pa metoda iz razreda `IzredniStudent`. Zanka

```
for (Student student: studentje) {
    System.out.printf("%s: %d%n", student.vrniIP(),
                    student.stroski());
}
```

torej proizvede sledeči izpis:

```
Ana Arko: 500      // metoda stroski iz razreda Student
Bojan Ban: 900     // metoda stroski iz razreda IzredniStudent
Cvetka Cevc: 750   // metoda stroski iz razreda IzredniStudent
Denis Denk: 450    // metoda stroski iz razreda Student
```

**Naloga 7.11** Podani so naslednji razredi:

```
public class A {
    public int f() {
        return 1;
    }
}

public class B extends A {
    @Override
    public int f() {
        return super.f() + 1;
    }
}
```

```

public class C extends B {
    @Override
    public int f() {
        return super.f() + 1;
    }
}

public class D extends B {
    @Override
    public int f() {
        return super.f() + 2;
    }
}

```

Kaj izpiše sledeča koda?

```

A a = new B();
B b = new C();
B c = b;
D d = new D();
A[] t = {new A(), a, b, c, d};
a = d;
for (A element: t) {
    System.out.println(element.f());
}

```

## 7.3 Hierarhija geometrijskih likov

### 7.3.1 Načrt hierarhije

V tem razdelku bomo napisali nekoliko obsežnejši program. Ukvarjali se bomo z geometrijskimi liki, in sicer s pravokotniki, kvadrati in krogi. Napisali bomo metode za izpis osnovnih podatkov o likih (dolžini stranic pri pravokotniku, dolžina stranice pri kvadratu oziroma polmer pri krogu) ter za izračun ploščine in obsega. Like bi radi obravnavali karseda poenoteno, zato jih bomo predstavili z objekti, kazalce nanje pa hranili v tabeli. Če bomo, na primer, želeli izpisati ploščine vseh likov v tabeli, se bomo po njej sprehodili z zanko in za vsak element poklicali metodo `ploscina`.

Ker imajo liki različne attribute, bomo za vsak tip lika napisali svoj razred. Pravokotnike bomo predstavili z objekti razreda `Pravokotnik`, kvadrate z objekti razreda `Kvadrat`, kroge pa z objekti razreda `Krog`. Ker je kvadrat poseben primer pravoko-

tnika, je razred Kvadrat smiselno definirati kot podrazred razreda Pravokotnik.

Če želimo kazalce na objekte različnih razredov hraniti v isti tabeli, morajo biti vsi razredi podrazredi istega razreda. Ker te vloge ne more prevzeti nobeden od navedenih treh razredov (npr. nima smisla, da bi bil razred Krog nadrazred ali podrazred razreda Pravokotnik), potrebujemo poseben skupni nadrazred. Ta razred se bo po pričakovanjih imenoval Lik. Naša hierarhija je s tem povsem določena:

```
public class Lik {
    ...
}

public class Pravokotnik extends Lik {
    ...
}

public class Kvadrat extends Pravokotnik {
    ...
}

public class Krog extends Lik {
    ...
}
```

V nadaljevanju bomo razrede dopolnili tako, da jih bomo lahko uporabljali po zgledu sledečega primera:

```
// koda 7.4
// izdelamo tabelo različnih likov
Lik[] liki = {new Kvadrat(7.0), new Krog(5.0),
              new Pravokotnik(6.0, 7.5)};

// za vsak lik v tabeli izpišemo osnovne podatke
// ter ploščino in obseg
for (Lik lik: liki) {
    System.out.println(lik.toString()); // System.out.println(lik);
    System.out.printf("p = %.1f, o = %.1f%n",
                      lik.ploscina(), lik.obseg());
    System.out.println();
}
```

Gornja koda bo izpisala sledeče:

```
kvadrat [stranica = 7.0]
```

```

p = 49.0, o = 28.0

krog [polmer = 5.0]
p = 78.5, o = 31.4

pravokotnik [širina = 6.0, višina = 7.5]
p = 45.0, o = 27.0

```

V zanki *for* za vsak lik iz tabele *liki* pokličemo metode *toString*, *ploscina* in *obseg*. Ker lahko kazalci v tabeli *liki* kažejo na objekte tipa *Pravokotnik*, *Kvadrat* ali *Krog*, morajo biti omenjene metode prisotne v vseh treh razredih. Izvajalnik bo na podlagi tipov objektov, na katere kažejo posamezni kazalci v tabeli, izbral ustrezno metodo. Na primer, v prvem obhodu zanke v kodi 7.4 bo izraz *lik.ploscina()* poklical metodo *ploscina* iz razreda *Kvadrat*, saj prvi kazalec v tabeli kaže na objekt tipa *Kvadrat*. V drugem obhodu bo klic *lik.toString()* poklical metodo *toString* iz razreda *Krog*.

### 7.3.2 Razred *Lik*

Tabela *liki* bo vsebovala samo kazalce na objekte podrazredov razreda *Lik*; noben objekt ne bo neposredno pripadal razredu *Lik*. Kljub temu pa morajo biti metode *toString*, *ploscina* in *obseg* prisotne *tudi* v razredu *Lik*. Zakaj, če vemo, da se v tem razredu ne bodo nikoli klicale? Zaradi *prevajalnika*, ki vidi zgolj tipe kazalcev. Ker je tabela *liki* deklarirana kot tabela tipa *Lik[]*, so vsi njeni elementi (kazalci) tipa *Lik*, zato bo prevajalnik ob klicu *liki[i].metoda(...)* preveril, ali se metoda *metoda* nahaja v razredu *Lik*. Če je ne bo, bo javil napako.

Glede na to, da morajo biti metode *toString*, *ploscina* in *obseg* v razredu *Lik* prisotne izključno zaradi zahtev prevajalnika, klicale pa se nikoli ne bodo, saj noben lik ne bo ustvarjen kot objekt razreda *Lik*, je pravzaprav vseeno, kako delujejo. Lahko, recimo, vrnejo vrednost *null* oziroma *0.0*. Bolje pa je, da jih deklariramo kot *abstraktne*. Abstraktna metoda je metoda brez telesa, deklarirana z določilom *abstract*. Tovrstne metode bralcu programa jasno povedo, da obstajajo samo zaradi pričakovanj prevajalnika.

Razred, ki vsebuje vsaj eno abstraktno metodo, mora biti *tudi sam* deklariran kot abstrakten. Če je razred *R* abstrakten, potem ne moremo izdelovati njegovih objektov; klic *new R(...)* sproži napako pri prevajanju, tudi če razred vsebuje ustrezajoč konstruktor. Razred z vsaj eno abstraktno metodo mora torej biti abstrakten, obratno pa ni nujno, saj lahko definiramo abstrakten razred brez abstraktnih metod.

Zaradi dedovanja so abstraktne metode prisotne tudi v podrazredu abstraktnega razreda, seveda pa imamo možnost, da jih tam *definiramo* oziroma *implementiramo*.<sup>2</sup>

<sup>2</sup>Abstraktnih metod ne »redefiniramo«, saj so zgolj deklarirane. Kljub temu bomo pojem *redefinicija* uporabljali kot skupen izraz za »pravo« redefinicijo in implementacijo abstraktnih metod.

Če podrazred ne implementira vseh abstraktnih metod, mora biti tudi sam abstrakten.

Razred `Lik` bo torej abstrakten, takšni pa bosta tudi metodi `ploscina` in `obseg` v njem:

```
public abstract class Lik {
    public abstract double ploscina(); // pozor: podpičje, ne {}!

    public abstract double obseg();
}
```

Kaj pa metoda `toString` v razredu `Lik`? Tudi ta bi lahko bila abstraktna. Ni pa nujno. Če dobro pogledamo izpise, ki jih tvori koda 7.4, vidimo, da so za vse tri tipe likov sestavljeni na enak način:

*vrsta* [podatki]

Rezultati metode `toString` za različne tipe likov se torej razlikujejo le po komponentah *vrsta* (niz pravokotnik, kvadrat ali krog) in *podatki* (seznam osnovnih podatkov v obliki niza), splošna zgradba pa je vseskozi enaka. Zato si prihranimo nekaj dela, če splošno zgradbo zajamemo v metodi `toString`, specifične komponente pa v abstraktnih metodah `vrsta` in `podatki`, ki ju bomo implementirali v posameznih podrazredih:

```
// koda 7.5 (dedovanje/lik/Lik.java)
public abstract class Lik {
    public abstract double ploscina();

    public abstract double obseg();

    public String toString() {
        return String.format("%s [%s]",
                               this.vrsta(), this.podatki());
    }

    public abstract String vrsta();

    public abstract String podatki();
}
```

Zakaj smo metodo `toString` v razredu `Lik` definirali v celoti? Saj se ne bo nikoli klicala, mar ne? Previdno: izraz `lik.toString()` te metode resda ne bo neposredno poklical, saj kazalec *lik* nikoli ne bo kazal na objekt tipa `Lik` (ker je razred `Lik` abstrakten, takega objekta niti ne moremo izdelati). Vendar pa se bo poklicala metoda



`toString` iz enega od podrazredov (odvisno od tipa objekta, na katerega kaže kazalec *lik*), ta pa lahko istoimensko metodo iz razreda `Lik` pokliče s pomočjo izraza `super.toString()`. Dejansko pa se bo metoda `toString` v podrazredih razreda `Lik` enostavno podedovala (ne bomo je redefinirali), zato bo klic `lik.toString()` posredno poklical kar metodo `toString` iz razreda `Lik`.

Metoda `toString` pokliče metodi `vrsta` in `podatki`, a seveda ne njunih abstraktnih različic v razredu `Lik`. Metodi `vrsta` in `podatki` bosta v podrazredih redefinirani, poklicali pa se bosta tisti različici, ki ustrezata tipu objekta, na katerega kaže kazalec `this`. Na primer, če metodo `toString` pokličemo nad kazalcem, ki kaže na objekt tipa `Kvadrat`, se bosta poklicali metodi `vrsta` in `podatki` iz razreda `Kvadrat`.

Kako pa je s konstruktorjem v razredu `Lik`? Ker razred nima atributov, konstruktorja nima smisla dodati, v nasprotnem primeru pa bi to najbrž storili. Ker pa je razred abstrakten, bi se njegov konstruktor lahko poklical zgolj s stavkom `super(...)` v konstruktorju podrazreda. V razredu `Lik` torej ne bomo definirali lastnega konstruktorja, zavedati pa se moramo, da prevajalnik zaradi odsotnosti eksplicitno definiranih konstruktorjev doda privzeti konstruktor:

```
public Lik() {
    super();
}
```

### 7.3.3 Razred Pravokotnik

Z razredom `Lik` smo zaključili. Lotimo se razreda `Pravokotnik`. Ta razred ne sme biti abstrakten, saj bomo izdelovali njegove objekte. To pa pomeni, da ne sme vsebovati abstraktnih metod. Ker podeduje vse metode razreda `Lik`, bomo morali implementirati metode `obseg`, `ploscina`, `vrsta` in `podatki`.

Objekti razreda `Pravokotnik` predstavljajo običajne pravokotnike, zato v razredu potrebujemo še atributa `sirina` in `visina` ter konstruktor za njuno inicializacijo. Metode `toString` nam ni treba redefinirati, saj je tista iz razreda `Lik` dobra tudi za razred `Pravokotnik`, le metodi `vrsta` in `podatki` moramo implementirati tako, da bosta vrnila ustrezna niza. Razred `Pravokotnik` lahko sedaj napišemo v celoti:

```
// koda 7.6 (dedovanje/lik/Pravokotnik.java)
public class Pravokotnik extends Lik {
    private double sirina;
    private double visina;

    public Pravokotnik(double sirina, double visina) {
        this.sirina = sirina;
        this.visina = visina;
    }
}
```

```

@Override
public double ploscina() {
    return this.sirina * this.visina;
}

@Override
public double obseg() {
    return 2 * (this.sirina + this.visina);
}

@Override
public String vrsta() {
    return "pravokotnik";
}

@Override
public String podatki() {
    return String.format("širina = %.1f, višina = %.1f",
        this.sirina, this.visina);
}
}

```

#### 7.3.4 Razred Kvadrat

Razred Kvadrat bi lahko izpeljali iz razreda Lik, a ker so kvadrati pravzaprav pravokotniki, ga bomo izpeljali iz razreda Pravokotnik:

```

public class Kvadrat extends Pravokotnik {
    ...
}

```

Razred Kvadrat torej že vsebuje atributa `sirina` in `visina` ter metode `ploscina`, `obseg`, `vrsta` in `podatki`. Seveda pa moramo definirati konstruktor. Ta sprejme en sam parameter: dolžino stranice kvadrata. Ker je kvadrat pravokotnik z enakima dolžinama stranic, mora konstruktor svoj parameter skopirati v oba atributa:

```

// koda 7.7 (dedovanje/lik/Kvadrat.java)
public class Kvadrat extends Pravokotnik {
    public Kvadrat(double stranica) {
        this.sirina = stranica;
        this.visina = stranica;
    }
}

```

```

    }
    ...
}

```

Sedaj nas ne bi več smelo presenetiti, da se gornja koda ne prevede. Prvič, atributa *sirina* in *visina* sta v razredu *Pravokotnik* deklarirana kot privatna, in drugič, če se konstruktor ne prične s stavkom `super(...)` ali `this(...)`, potem prevajalnik sam vstavi klic `super()`, ta pa se ne more prevesti, saj razred *Pravokotnik* nima konstruktorja brez parametrov. Obema težavama se elegantno ognemo, če nalogo nastavitve atributov *sirina* in *visina* predamo konstruktorju nadrazreda:

```

public class Kvadrat extends Pravokotnik {
    public Kvadrat(double stranica) {
        super(stranica, stranica);
    }
    ...
}

```

Na primer, izraz `new Kvadrat(7.0)` izdelava objekt tipa *Kvadrat* z atributoma *sirina* in *visina*, nato pa pokliče konstruktor razreda *Kvadrat* z argumentom 7.0. Konstruktor razreda *Kvadrat* pokliče konstruktor razreda *Pravokotnik* z argumentoma 7.0 in 7.0, ta pa atributoma *sirina* in *visina* pravkar ustvarjenega objekta priredi število 7.0.

Metod *ploscina* in *obseg* nam ni treba redefinirati, saj se ploščina in obseg kvadrata izračunata na enak način kot ploščina in obseg pravokotnika. Moramo pa redefinirati metodi *vrsta* in *podatki*, saj je izpisni niz za kvadrate drugačen kot za pravokotnike.

```

public class Kvadrat extends Pravokotnik {
    ...
    @Override
    public String vrsta() {
        return "kvadrat";
    }

    @Override
    public String podatki() {
        return String.format("stranica = %.1f", this.sirina);
    }
}

```

Pri metodi *podatki* naletimo na težavo, saj atribut *sirina* v razredu *Kvadrat* ni neposredno dostopen. Težavo lahko rešimo bodisi tako, da njegovo dostopno

določilo spremenimo v `protected`, ali pa tako, da v razred `Pravokotnik` dodamo ustrezen »getter«. Odločili se bomo za drugo možnost:

```
public class Pravokotnik extends Lik {
    ...
    public double vrniSirino() {
        return this.sirina;
    }
}

public class Kvadrat extends Pravokotnik {
    ...
    @Override
    public String podatki() {
        return String.format("stranica = %.1f", this.vrniSirino());
    }
}
```

### 7.3.5 Razred Krog

Razred `Krog` je podoben razredu `Pravokotnik` in ne potrebuje posebnih pojasnil:

```
// koda 7.8 (dedovanje/liki/Krog.java)
public class Krog extends Lik {
    private double polmer;

    public Krog(double polmer) {
        this.polmer = polmer;
    }

    @Override
    public double ploscina() {
        return Math.PI * this.polmer * this.polmer;
    }

    @Override
    public double obseg() {
        return 2.0 * Math.PI * this.polmer;
    }

    @Override
    public String vrsta() {
```

```

        return "krog";
    }

    @Override
    public String podatki() {
        return String.format("polmer = %.1f", this.polmer);
    }
}

```

**Naloga 7.12** V hierarhijo smiselno dodajte atribut barva tipa String ali (še bolje) java.awt.Color. Barva je lastnost, ki pripada vsem tipom likov. Vrednost atributa barva naj postane sestavni del izpisa, ki ga tvori metoda toString.

**Naloga 7.13** V hierarhijo smiselno dodajte še razred Elipsa in ustrezno posodobite razred Krog.

### 7.3.6 Primer klica metode v hierarhiji

Da bomo lažje razumeli, kako se naša hierarhija obnaša pri klicih metod, bomo preučili izvajanje sledeče kode:

```

Lik lik = new Kvadrat(7.0);
System.out.println(lik.toString());

```

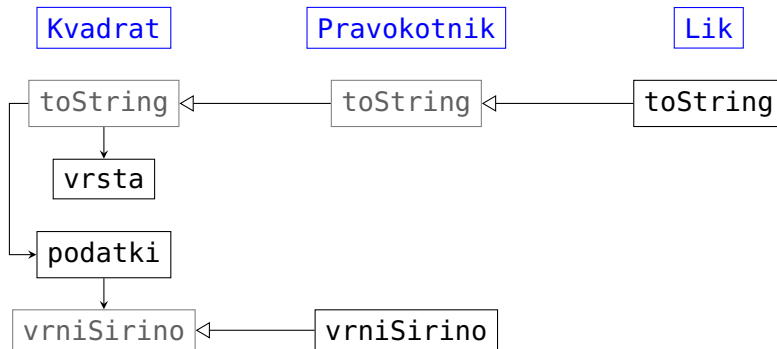
Ker kazalec lik kaže na objekt tipa Kvadrat, se v drugi vrstici pokliče metoda toString iz razreda Kvadrat. Ali ta metoda v razredu Kvadrat sploh obstaja? Seveda, saj se podeduje od razreda Pravokotnik, ta razred pa jo podeduje od razreda Lik. Metoda toString je torej v razredu Kvadrat enaka kot v razredu Lik.

Metoda toString nad objektom this izvede metodi vrsta in podatki. Ker kazalec this kaže na objekt tipa Kvadrat, se pokličeta različici metod vrsta in podatki iz razreda Kvadrat. Prva metoda vrne enostavno niz kvadrat, druga pa si pomaga z metodo vrniSirino iz razreda Kvadrat. Ta metoda, ki se podeduje iz razreda Pravokotnik, vrne atribut sirina objekta this, torej vrednost 7.0.

Odnosi med metodami pri klicu lik.toString(), pri čemer kazalec lik kaže na objekt tipa Kvadrat, so prikazani na sliki 7.2. Puščica s konico v obliki praznega trikotnika predstavlja dedovanje, navadna puščica pa klic metode.

### 7.3.7 Testni razred

S hierarhijo razredov za predstavitev likov smo zaključili, sedaj pa bomo napisali še poseben razred, v katerem jo bomo preizkusili. V razredu Glavni bomo napisali tri metode:



Slika 7.2 Odnosi med metodami ob klicu metode toString nad objektom razreda Kvadrat.

- `public static void izpisiPodatke(Lik[] liki)`  
Za vsak lik v tabeli izpiše osnovne podatke ter ploščino in obseg.
- `public static Lik likZNajvecjoPloscino(Lik[] liki)`  
Vrne kazalec na lik z največjo ploščino oziroma null, če je tabela prazna.
- `public static Pravokotnik pravokotnikZNajvecjoSirino(Lik[] liki)`  
Vrne kazalec na pravokotnik z največjo širino oziroma null, če tabela ne vsebuje nobenega pravokotnika.

V metodi `izpisiPodatke` se enostavno sprehodimo po tabeli `lik` in za vsak lik pokličemo metode `toString`, `ploscina` in `obseg`. Tip ciljnega objekta nam pove, katera različica metode se bo poklicala, zato se bodo za pravokotnike klicale metode iz razreda `Pravokotnik`, za kvadrate in kroge pa metode iz razredov `Kvadrat` oziroma `Krog`.

```
// koda 7.9 (dedovanje/lik/Glavni.java)
public static void izpisiPodatke(Lik[] liki) {
    for (Lik lik: liki) {
        System.out.printf("%s | p = %.1f, o = %.1f%n",
                           lik.toString(), lik.ploscina(), lik.obseg());
    }
}
```

V metodi `likZNajvecjoPloscino` se prav tako sprehodimo po tabeli likov, sproti pa vzdržujemo kazalec na lik z doslej največjo ploščino (`najLik`). Da se izognemo večkratnemu računanju ploščine istega lika, hranimo tudi doslej največjo ploščino (`najPloscina`).

```
public static Lik likZNajvecjoPloscino(Lik[] liki) {
```

```

    Lik najLik = null;
    double najPloscina = 0.0;

    for (Lik lik: liki) {
        double ploscina = lik.ploscina();
        if (najLik == null || ploscina > najPloscina) {
            najPloscina = ploscina;
            najLik = lik;
        }
    }
    return najLik;
}

```

Pri pisanju metode `pravokotnikZNajvecjoSirino` bomo spoznali nekaj novih konceptov, zato si zasluži poseben razdelek.

### 7.3.8 Operator `instanceof` in izrecna pretvorba tipa

V metodi `pravokotnikZNajvecjoSirino` vzdržujemo kazalec na doslej najširši pravokotnik (`naj`). Za vsak lik preverimo, ali je pravokotnik, in če je, pridobimo njegovo širino in po potrebi posodobimo kazalec `naj`.

```

public static Pravokotnik pravokotnikZNajvecjoSirino(Lik[] liki) {
    Pravokotnik naj = null;
    for (Lik lik: liki) {
        if (lik je pravokotnik) {
            pridobi širino in po potrebi posodobi kazalec naj
        }
    }
    return naj;
}

```

Kako ugotovimo, ali kazalec `lik` kaže na objekt tipa `Pravokotnik`? Lahko si pomagamo z metodo `vrsta`. Če vrne niz `pravokotnik`, potem je naš lik seveda pravokotnik. Ker pa so tudi kvadrati pravokotniki, moramo upoštevati še niz `kvadrat`:

```

String vrsta = lik.vrsta();
if (vrsta.equals("pravokotnik") || vrsta.equals("kvadrat")) {
    ...
}

```

S prikazanim pristopom ni v osnovi nič narobe, ima pa to slabost, da moramo metodo `vrsta` dodati prav v vsak razred. Če izvirne kode hierarhije ne moremo spreminjati, je ta način neuporaben.

Za preverjanje tipa objekta, na katerega kaže podani kazalec, lahko v javi uporabimo operator `instanceof`. Izraz

`obj instanceof R`

ima vrednost `true` natanko tedaj, ko kazalec `obj` kaže na objekt razreda `R` ali (pозor!) objekt poljubnega podrazreda razreda `R`. Vrednost izraza `lik instanceof Pravokotnik` bo potemtakem `true` natanko takrat, ko bo spremenljivka `lik` kazala na objekt tipa `Pravokotnik` ali `Kvadrat`.

Metodo `pravokotnikZNajvecjoSirino` lahko sedaj napišemo takole:

```
public static Pravokotnik pravokotnikZNajvecjoSirino(Lik[] liki) {
    Pravokotnik naj = null;
    for (Lik lik: liki) {
        if (lik instanceof Pravokotnik) {
            if (naj == null || lik.vrniSirino() > naj.vrniSirino()) {
                naj = lik;
            }
        }
    }
    return naj;
}
```

Ko metodo poskusimo prevesti, nam ne uspe. Prva napaka se skriva v izrazu `lik.vrniSirino()`. Kot vemo, prevajalnik pozna samo tipe kazalcev. Tip kazalca `lik` je enak `Lik`, zato bo prevajalnik iskal metodo `vrniSirino` v razredu `Lik`. Ker je ne bo našel, bo javil napako. Napačen je tudi stavek `naj = lik`, saj kazalca tipa `R` ne moremo prirediti spremenljivki podtipa tipa `R`.

Obe napaki rešimo z *izrecno pretvorbo tipa*:

```
public static Pravokotnik pravokotnikZNajvecjoSirino(Lik[] liki) {
    Pravokotnik naj = null;
    for (Lik lik: liki) {
        if (lik instanceof Pravokotnik) { // (1)
            Pravokotnik p = (Pravokotnik) lik;
            if (naj == null || p.vrniSirino() > naj.vrniSirino()) {
                naj = p;
            }
        }
    }
    return naj;
}
```

Izraz `(Pravokotnik) lik` vrne nov kazalec tipa `Pravokotnik`, ki kaže na isti objekt



kot kazalec lik. Izraza `p.vrniSirino()` in `naj = p` se bosta potemtakem normalno prevedla. Kaj pa sam izraz `(Pravokotnik) lik`? Ker prevajalnik ne ve, kam kaže kazalec `lik`, ga prevede brez napak ali opozoril, saj dopušča možnost, da kazalec `lik` dejansko kaže na objekt tipa `Pravokotnik` ali `Kvadrat`. Če pa se bo v času izvajanja izrecne pretvorbe tipa izkazalo, da kazalec `lik` dejansko kaže na objekt tipa `Krog`, se bo sprožila izjema tipa `ClassCastException`, saj tip `Pravokotnik` ni podtip tipa `Krog`. V našem primeru se to seveda ne bo zgodilo, saj se izrecna pretvorba tipa zaradi pogojnega stavka v vrstici (1) izvede le tedaj, ko kazalec `lik` kaže na objekt tipa `Pravokotnik` ali njegovega podtipa.

V splošnem velja takole: izraz  $(R) \ p$  se prevede, če je tip kazalca  $p$  enak bodisi  $R$  bodisi  $R'$ , kjer je  $R'$  poljuben podtip ali nadtip tipa  $R$ . V trenutku izvajanja izraza  $(R) \ p$  pa mora kazalec  $p$  kazati na objekt tipa  $R$  ali na objekt poljubnega podtipa (ne pa tudi nadtipa!) tipa  $R$ .

Za lažje razumevanje si privoščimo še en primer. Recimo, da je razred  $A$  nadrazred razredov  $B$  in  $C$ , ta dva pa med sabo nista v hierarhičnem odnosu. Predpostavimo, da ima vsak od teh razredov konstruktor brez parametrov in da definiramo sledeče spremenljivke:

```
A a1 = new A();
A a2 = new B();
A a3 = new C();
B b = new B();
C c = new C();
```

Tabela 7.1 prikazuje primere izrazov, ki se (vsak zase) prevedejo in izvedejo (stolpec  $P \wedge I$ ), izrazov, ki se prevedejo, pri izvajanju pa sprožijo izjemo tipa `ClassCastException` (stolpec  $P \wedge \neg I$ ), in izrazov, ki se niti ne prevedejo (stolpec  $\neg P$ ).

**Tabela 7.1** Primeri izrazov, ki se (ne) prevedejo ( $P$  oz.  $\neg P$ ) oziroma (ne) izvedejo ( $I$  oz.  $\neg I$ ).

$P \wedge I$	$P \wedge \neg I$	$\neg P$
(B) a2	(B) a1	(C) b
(C) a3	(C) a1	(B) c
(A) b	(C) a2	
(A) c	(B) a3	

Operator `instanceof` nam marsikdaj lahko koristi, vendar pa z njim ne gre pretiravati. V navezi z izrecno pretvorbo tipa nam namreč omogoča programiranje v takem slogu:

```
if (lik instanceof Kvadrat) {
```

```

    Kvadrat k = (Kvadrat) lik;
    System.out.printf("kvadrat [stranica = %.1f]%n", k.vrniSirino());
} else if (lik instanceof Pravokotnik) {
    Pravokotnik p = (Pravokotnik) lik;
    System.out.printf("pravokotnik [širina = %.1f, višina = %.1f]%n",
        p.vrniSirino(), p.vrniVisino());
} else {
    Krog k = (Krog) lik;
    System.out.printf("krog [polmer = %.1f]%n", k.vrniPolmer());
}

```

Takšna koda je v nasprotju z načeli lepega programiranja. Načrtovalci programskih jezikov so »iznašli« dedovanje ravno zato, da bi se izognili nepreglednim verigam pogojnih stavkov. Zato rabo operatorja `instanceof` in izrecne pretvorbe tipa omejimo na primere, ko je nek podatek ali izračun smiseln samo za enega od podrazredov — npr. širina ima pomen samo pri pravokotnikih, pri krogih pa ne. Če je podatek ali izračun smiseln za celotno hierarhijo, definiramo ustrezen atribut ali metodo v hierarhično najvišjem razredu, v podrazredih pa ga oz. jo samo podedujemo. Če se izračun za določen podrazred razlikuje od privzetega, potem metodo tam redefiniramo (tak primer je metoda `stroski` pri hierarhiji študentov). Če pa izračun v nadrazredu ni mogoč ali smiseln ali pa če izračuni za različne razrede med seboj nimajo nič skupnega, potem metodo v najvišjem razredu deklariramo kot abstraktno in jo definiramo v podrazredih. Tak primer sta metodi za računanje ploščine in obsega v hierarhiji likov.

#### Naloga 7.14 Razred Glavni dopolnite z metodo

```
public static void vecji(Lik[] liki, Lik meja)
```

ki izpiše podatke o vseh likih, pri katerih je ploščina večja od ploščine lika meja.

#### Naloga 7.15 Razred Glavni dopolnite z metodo

```
public static void uredi(Lik[] liki)
```

ki tabelo likov uredi tako, da bodo v njej najprej nastopali pravokotniki, ki niso tipa Kvadrat, nato kvadrati, nazadnje pa krogi. Vsaka skupina naj bo padajoče urejena po ploščini.

#### Naloga 7.16 Kaj izpiše sledeča koda?

```

Lik p = new Pravokotnik(3.0, 4.0);
Lik q = new Kvadrat(5.0);
Kvadrat r = new Kvadrat(6.0);
System.out.println(p instanceof Lik);

```

```

System.out.println(q instanceof Lik);
System.out.println(r instanceof Lik);
System.out.println(p instanceof Pravokotnik);
System.out.println(q instanceof Pravokotnik);
System.out.println(r instanceof Pravokotnik);
System.out.println(p instanceof Kvadrat);
System.out.println(q instanceof Kvadrat);
System.out.println(r instanceof Kvadrat);

```

**Naloga 7.17** Denimo, da najprej izvršimo te stave:

```

Lik p = new Pravokotnik(3.0, 4.0);
Lik q = new Kvadrat(5.0);
Kvadrat r = new Kvadrat(6.0);

```

Za vsakega od naslednjih izrazov posebej povejte, ali (A) bi se prevedel in izvedel; (B) bi se prevedel, ne pa tudi izvedel; (C) se ne bi niti prevedel.

```

(Pravokotnik) p
(Kvadrat) p
(Krog) p
(Pravokotnik) q
(Kvadrat) q
(Krog) q
(Lik) r
(Pravokotnik) r
(Krog) r

```

### 7.3.9 Kovariantnost

Javanske tabele so *kovariantne*: če je tip *B* podtip tipa *A*, je tudi tip *B[]* podtip tipa *A[]*. To pomeni, da lahko, denimo, kazalec tipa *Kvadrat[]* priredimo spremenljivki tipa *Pravokotnik[]* ali spremenljivki tipa *Lik[]*. Vsi sledeči stavki se bodo torej prevedli ...

```

Pravokotnik[] pr = {new Pravokotnik(3.0, 4.0), new Kvadrat(5.0)};
Lik[] liki = pr;
Pravokotnik[] pr2 = (Pravokotnik[]) liki;
Kvadrat[] kv = (Kvadrat[]) pr;

```

... vendar pa bo zadnji stavek sprožil izjemo tipa *ClassCastException*, saj kazalec *pr* kaže na objekt tipa *Pravokotnik[]*, ne *Kvadrat[]*. Izjema bi se sprožila tudi

v primeru, če bi tabela `pr` vsebovala same kvadrate, saj bi bil tip tabele še vedno `Pravokotnik[]`. Upoštevajmo, da je zapis

```
Pravokotnik[] pr = {...};
```

okrajšava za

```
Pravokotnik[] pr = new Pravokotnik[]{...};
```

Koda

```
Pravokotnik[] pr = {new Pravokotnik(3.0, 4.0), new Kvadrat(5.0)};
Lik[] liki = pr;
liki[0] = new Kvadrat(6.0);
liki[1] = new Krog(7.0);
```

bi se prav tako prevedla. Prevajalnika zadnja vrstica ne zmoti: kazalec `liki[1]` je tipa `Lik` (ker je kazalec `liki` tipa `Lik[]`), tip `Krog` pa je podtip tipa `Lik`. Pri izvajanju pa ta vrstica povzroči izjemo tipa `ArrayStoreException`. Izvajalnik namreč preverja, ali se v tabelo tipa `R[]` shranjujejo le objekti tipa `R` ali podtipa tipa `R`. Če to ne drži, sproži izjemo.

**Naloga 7.18** Denimo, da izdelamo sledeči tabeli:

```
Pravokotnik[] pr1 = {new Kvadrat(3.0)};
Pravokotnik[] pr2 = new Kvadrat[]{new Kvadrat(4.0)};
```

Za vsakega od sledečih stavkov povejte, ali se (vsak zase) prevede in izvede, samo prevede ali niti ne prevede:

```
Lik[] liki1 = pr1;
Lik[] liki2 = pr2;
Kvadrat[] kv1 = pr1;
Kvadrat[] kv2 = (Kvadrat[]) pr1;
Kvadrat[] kv3 = pr2;
Kvadrat[] kv4 = (Kvadrat[]) pr2;
Kvadrat[] kv5 = (Kvadrat[]) (Lik[]) pr2;
Krog[] kr1 = pr1;
Krog[] kr2 = (Lik[]) pr1;
Krog[] kr3 = (Krog[]) pr1;
Krog[] kr4 = (Krog[]) (Lik[]) pr1;
```

**Naloga 7.19** Napišite metodo

```
public static int preveri(Lik[] liki)
```

ki vrne

- 1, če kazalec `l` kaže na tabelo tipa `Pravokotnik[]` ali njegovega podtipa;
- 2, če pogoj iz prejšnje alineje ne velja, vendar pa je vsak element tabele kazalec na objekt tipa `Pravokotnik` ali njegovega podtipa;
- 3, če ne velja nič od tega.

## 7.4 Razred Object

Razred `Object` je edini razred v javi, ki nima svojega nadrazreda. Vsi ostali razredi (tako vgrajeni kot tisti, ki jih napišemo sami) so podrazredi razreda `Object`. Na primer, razred `Cas`, ki smo ga v razdelku 6.2 deklarirali kot

```
public class Cas
```

je v resnici deklariran kot

```
public class Cas extends Object
```

le da lahko pristavek `extends Object` izpustimo.

Tip `Object` je nadtip vseh javanskih referenčnih tipov (razen seveda tipa `Object` samega). To vključuje tudi tabelarične tipe. Na primer, sledeča koda se prevede in izvede:

```
int[] a = {1, 2, 3};
char[][] b = {{'a', 'b'}, {'c'}};
String[] c = {"včeraj", "danes", "jutri"};
Object o = a;
Object[] t1 = {a, b, c};
Object[] t2 = b;
```

Zadnja vrstica deluje zaradi kovariantnosti: ker je tip `char[]` podtip tipa `Object`, je tudi tip `char[][]` podtip tipa `Object[]`.

Razred `Object` nima atributov, premore pa kar nekaj metod. Te metode samodejno podedujejo vsi javanski razredi, po potrebi pa jih lahko redefinirajo. Omejili se bomo na sledeče tri metode:

```
public String toString()
public boolean equals(Object obj)
public int hashCode()
```

### 7.4.1 Metoda toString

Metoda toString v razredu Object vrne niz, ki vsebuje

- oznako tipa objekta, na katerega kaže kazalec this;
- ločilo @;
- število (v šestnajstiškem številskem sistemu), ki ga za objekt this vrne metoda hashCode (razdelek 7.4.3).

Pokličimo metodo toString nad objektom tipa Object:

```
Object obj = new Object();
System.out.println(obj.toString()); // npr. java.lang.Object@b81eda8
```

Če metode toString v svojem razredu ne redefiniramo, a jo kljub temu pokličemo, dobimo podoben izpis, saj se bo zgolj podedovala. Pogosto pa metodo redefiniramo, in to tako, da vrne niz, ki vsebuje ključne podatke o objektu. Na primer, v razredu Cas smo metodo toString napisali tako, da vrne niz, ki podaja uro in minuto v obliki, kot smo je vajeni v vsakdanjem življenju:

```
public class Cas {
    ...
    @Override
    public String toString() {
        return String.format("%d:%02d", this.ura, this.minuta);
    }
}
```

Metoda toString je tesno povezana z izpisovanjem, saj pri vseh treh metodah System.out.print\* obstaja različica, ki sprejme parameter tipa Object in nad tem parametrom pokliče metodo toString. Zato za objekt poljubnega tipa velja, da lahko, denimo, namesto

```
System.out.println(objekt.toString());
```

pišemo kar

```
System.out.println(objekt);
```

### 7.4.2 Metoda equals

Metoda equals je v razredu Object definirana preprosto takole:

```
public boolean equals(Object obj) {
```

```

    return this == obj;
}

```

Če metode `equals` ne redefiniramo, se bo torej obnašala popolnoma enako kot operator `==`: vrnila bo `true` natanko tedaj, ko kazalca `this` in `obj` kažeta na isti objekt. V razredu `Cas` metode `equals` nismo redefinirali, zato metoda za različna objekta z enako vsebino vrne `false`:

```

Cas a = new Cas(15, 40);
Cas b = a;
Cas c = new Cas(15, 40);
System.out.println(a.equals(b)); // true
System.out.println(a.equals(c)); // false

```

Marsikdaj pa se nam ne zdi smiselno, da bi se metoda `equals` obnašala enako kot dvojni enačaj. Zato jo redefiniramo, in to tako, da podana objekta primerja po vrednostih atributov. Potem bo iz `a == b` še vedno sledilo `a.equals(b)`, obratno pa ne bo več nujno res.

Redefinirajmo metodo `equals` v razredu `Cas`. Ta metoda bo podobna že napisani metodi `jeEnakKot`, vendar pa obstaja pomembna razlika: naša lastna metoda `jeEnakKot` sprejme parameter tipa `Cas`, parameter metode `equals` pa mora biti tipa `Object`, saj tako od nas zahtevajo pravila za redefinicijo metode. Seveda pa lahko argument metode `equals` kaže tudi na objekt tipa `Cas`, saj je razred `Cas` podrazred razreda `Object`.

Metodo `equals` bomo v razredu `Cas` napisali takole:

- Na začetku se nam splača preveriti, če kazalca `this` in `obj` kažeta na isti objekt. Če je to res, lahko takoj vrnemo `true`.
- Če objekt, na katerega kaže kazalec `obj`, ne pripada razredu `Cas`, potem vrnemo `false`, saj primerjava objektov različnih tipov vsaj v tem primeru ni smiselna.
- Če kazalec `obj` kaže na objekt tipa `Cas`, potem lahko njegov tip pretvorimo v `Cas` in primerjamo objekta po atributih.

```

// koda 7.10 (dedovanje/cas/Cas.java)
public class Cas {
    ...
    @Override
    public boolean equals(Object drugi) {
        if (this == drugi) {
            return true;
        }
    }
}

```

```

        if (!(drugi instanceof Cas)) {
            return false;
        }
        Cas drugiCas = (Cas) drugi;
        return this.ura == drugiCas.ura &&
            this.minuta == drugiCas.minuta;
    }
}

```

**Naloga 7.20** Napišite metodo `equals` za razred `Ulocek` iz poglavja 6.

**Naloga 7.21** Napišite metodo `equals` za razred `PostniNaslov`, ki je sestavljen iz atributov `postnaStevilka` (tipa `int`), `posta` (tipa `String`) in `ulicaInHisnaStevilka` (tipa `String`). Je nujno, da med seboj primerjate vse tri attribute podanega para objektov, ali lahko katerega od njih izpustite?

**Naloga 7.22** Naši prijatelji iz profesorjevega kabineta so soglasno sklenili, da je za razred `Oseba` (njegovi objekti predstavljajo posamezne osebe) edino pravilno, da metode `equals` *ne* redefinirajo. Zakaj?

### 7.4.3 Metoda `hashCode`

Metoda `hashCode` je v razredu `Object` definirana tako, da vrne celo število, ki predstavlja identiteto objekta, na katerega kaže kazalec `this`. Metoda za isti objekt vedno vrne isti rezultat, nimamo pa jamstva, da vrne različni števili za različna objekta (čeprav v praksi to skoraj vedno velja). Ni predpisano, kako naj metoda `hashCode` izračuna svoj rezultat, ponavadi pa je osnova za izračun pomnilniški naslov objekta.

Kazalca `a` in `c` v sledečem primeru kažeta na isti objekt, kazalec `b` pa na nek drug objekt. To se odraža v rezultatih metode `hashCode` (in tudi `toString`):

```

Object a = new Object();
Object b = new Object();
Object c = a;
System.out.println(a.toString()); // java.lang.Object@26a1ab54
System.out.println(b.toString()); // java.lang.Object@41cf53f9
System.out.println(c.toString()); // java.lang.Object@26a1ab54
System.out.println(a.hashCode()); // 648129364
System.out.println(b.hashCode()); // 1104106489
System.out.println(c.hashCode()); // 648129364

```

Sedaj lahko preverimo, da je šestnajstiško število, ki je del rezultata metode `toString`, enako številu, ki ga vrne metoda `hashCode`. Na primer,  $26A1AB54_{(16)} = 648129364$ .



Če metodo `hashCode` redefiniramo, to storimo običajno tako, da vrne število, ki čimbolj enolično določa objekt. (Zakaj je to smiselno, bomo videli v razdelku 7.7.2.) Metoda `hashCode` naj bi bila poleg tega usklajena z metodo `equals`. To pomeni, da moramo zagotoviti, da za vse pare kazalcev  $a$  in  $b$  velja sledeča implikacija:

$$a.equals(b) \implies (a.hashCode() == b.hashCode())$$

Zaželeno je tudi, da implikacija v čimveč primerih velja tudi v obratni smeri, a to ni obvezno.

Rezultat redefinirane metode `hashCode` ponavadi izračunamo po formuli

$$h = p_1h(a_1) + p_2h(a_2) + \dots + p_nh(a_n),$$

pri čemer so  $p_1, p_2, \dots, p_n$  medsebojno različna praštevila,  $h(a_1), h(a_2), \dots, h(a_n)$  pa rezultati metode `hashCode` za posamezne atribute. Pri atributih referenčnih tipov imamo metodo `hashCode` že tako na voljo (vsaj različico iz razreda `Object`, če že ni redefinirana), pri atributih primitivnih tipov pa si pomagamo s statičnimi metodami `hashCode` iz razredov `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` oziroma `Character`. Na primer, za atribut tipa `int` uporabimo statično metodo `hashCode` iz razreda `Integer`, za atribut tipa `char` pa statično metodo `hashCode` iz razreda `Character`.

Razred `Cas` ima dva atributa tipa `int`, zato bi metodo `hashCode` lahko v njem redefinirali takole:

```
// koda 7.11 (dedovanje/cas/Cas.java)
public class Cas {
    ...
    @Override
    public int hashCode() {
        return 17 * Integer.hashCode(this.ura) +
            31 * Integer.hashCode(this.minuta);
    }
}
```

**Naloga 7.23** Definirajte metodo `hashCode` za razred `Uloomek` iz poglavja 6.

**Naloga 7.24** Definirajte metodo `hashCode` za razred `PostniNaslov` iz naloge 7.21.

**Naloga 7.25** Protagonisti naloge 7.22 so ob vrčku brezalkoholnega piva prišli do naslednje ugotovitve: metodo `hashCode` je smiselno redefinirati natanko tedaj, ko je smiselno redefinirati metodo `equals`. Se z njimi strinjate?

**Naloga 7.26** Ali za metodo `hashCode` iz kode 7.11 velja

```
a.equals(b) ⇔ (a.hashCode() == b.hashCode())
```

— torej ekvivalenca, ne zgolj implikacija?

**Naloga 7.27** Zakaj je smiselno, da so koeficienti  $p_1, p_2, \dots, p_n$  v splošni formuli za računanje rezultata metode `hashCode` medsebojno različna praštevila?

#### 7.4.4 Razred `Object` in tabele

Ker je razred `Object` nadrazred vseh javanskih razredov, lahko vrednost katerega koli referenčnega tipa priredimo spremenljivki tipa `Object`. Od tod sledi, da lahko kazalci v tabeli tipa `Object[]` kažejo na objekte poljubnih tipov. Na primer:

```
Object[] mesanica = {
    new Oseba("Janez", "Novak", 'M', 2003),
    new Cas(10, 35),
    "Dober dan!"
};
```

Vsi trije elementi tabele `mesanica` so kazalci tipa `Object`, vendar pa prvi kaže na objekt tipa `Oseba`, drugi na objekt tipa `Cas`, tretji pa na objekt tipa `String`.

Sledeča zanka se sprehodi po elementih tabele in za vsak element posredno pokliče metodo `toString`:

```
for (Object element: mesanica) {
    System.out.println(element);
}
```

Dobimo tak izpis:

```
Janez Novak (M), 2003
10:35
Dober dan!
```

### 7.5 Ovojni tipi

Javanski primitivni tipi niso del hierarhije referenčnih tipov. Tabela tipa `Object[]` zato ne more hraniti spremenljivk tipov `int`, `char`, `boolean` itd. Lahko pa vsebuje spremenljivke *ovojnih* tipov. Java namreč za vsak primitivni tip ponuja razred, katerega objekti lahko hranijo posamezne vrednosti tega tipa. Na primer, objekt razreda `Integer` lahko hrani število tipa `int`, objekt razreda `Character` lahko vsebuje znak itd. Ovojni tipi, ki pripadajo posameznim primitivnim tipom, so zbrani v tabeli 7.2.

Tabela 7.2 Primitivni in pripadajoči ovojni tipi.

Primitivni tip	Ovojni tip
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

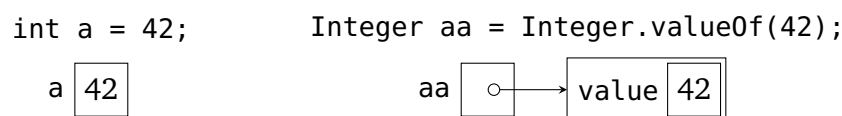
Vsak ovojni razred ponuja statično metodo `valueOf`, ki sprejme vrednost pripadajočega primitivnega tipa in ustvari objekt, ki hrani to vrednost:

```
Integer aa = Integer.valueOf(42);
Boolean bb = Boolean.valueOf(true);
Character cc = Character.valueOf('z');
```

Če želimo pridobiti vrednost, shranjeno v objektu ovojnega tipa, uporabimo metodo `intValue` (za tip `Integer`), `charValue` (za tip `Character`), `booleanValue` (za tip `Boolean`) itd.:

```
int a = aa.intValue();           // 42
boolean b = bb.booleanValue();  // true
char c = cc.charValue();        // 'z'
```

Pomembno se je zavedati, da spremenljivka `a` vsebuje število 42, medtem ko spremenljivka `aa` vsebuje kazalec na objekt, ki hrani število 42 v svojem atributu (slika 7.3).



Slika 7.3 Spremenljivka primitivnega tipa in spremenljivka ovojnega tipa.

Opisano pretvarjanje med spremenljivkami primitivnih in ovojnih tipov ni posebej razburljivo. Vendar pa java ponuja več kot to. Prevajalnik lahko namreč spremenljivke primitivnih tipov *samodejno pretvarja* v spremenljivke pripadajočih ovoj-

nih tipov in obratno.<sup>3</sup> Na primer, po izvedbi stavka

```
Integer aa = 42;
```

bo spremenljivka `aa` kazala na objekt, ki hrani število 42. Prevajalnik bo ugotovil, da je tip desne strani enak `int`, tip leve pa `Integer`, zato bo samodejno dodal klic ustrezne metode `valueOf`:

```
Integer aa = Integer.valueOf(42);
```

Podobno velja za ostale pare tipov:

```
Boolean bb = true;
Character cc = 'z';
```

Samodejno pretvarjanje deluje tudi v obratni smeri:

```
int a = aa;
boolean b = bb;
char c = cc;
```

Javanski prevajalnik v teh primerih vidi, da je na desni strani vrednost ovojnega tipa, na levi pa spremenljivka pripadajočega primitivnega tipa, zato sam doda klic ustrezne metode. Na primer, izraz `aa` samodejno pretvori v izraz `aa.intValue()`.

Prevajalnik je dovolj »inteligenten«, da izvede pretvorbo povsod, kjer je potrebna. Na primer, ker je jasno, da spremenljivke v tabeli tipa `Object[]` lahko pripadajo zgolj referenčnim tipom, lahko počnemo take reči:

```
Object[] mesanica = {
    "Dober dan!",
    42,          // Integer.valueOf(42)
    -3.14,       // Double.valueOf(-3.14)
    true,        // Boolean.valueOf(true)
    'a'          // Character.valueOf('a')
};
```

Prevajalnik bo izraz 42 sam pretvoril v klic `Integer.valueOf(42)` itd. Zanka

```
for (Object element: mesanica) {
    System.out.println(element);
}
```

bo za vsak element poklicala metodo `toString` in izpisala

---

<sup>3</sup>Po angleško se temu reče *autoboxing* (v smeri od primitivnih do ovojnih tipov) oziroma *unboxing* (v obratni smeri).

```

Dober dan!
42
-3.14
true
a

```

Po zaslugi samodejnega pretvarjanja lahko z ovojnimi tipi tudi »računamo«:

```

Integer a = 3;
Integer b = 5;
Integer c = a + b;

```

Prevajalnik gornje stavke pretvori takole:

```

Integer a = Integer.valueOf(3);
Integer b = Integer.valueOf(5);
Integer c = Integer.valueOf(a.intValue() + b.intValue());

```

V ozadju ni nikakršne »magije«, zgolj enostavna pravila. Ko analizira tretjo vrstico, prevajalnik ugotovi, da želimo sešteti dve vrednosti tipa `Integer`. To ni mogoče, zato ju s klicem metode `intValue` pretvori v vrednosti tipa `int`. Rezultat seštevanja, ki je seveda prav tako tipa `int`, mora nato zavoljo ujemanja z levo stranjo pretvoriti v tip `Integer`.

Ker so pretvorbe, ki se izvajajo v ozadju, programerju skrite, se ovojni tipi v večini primerov obnašajo enako kot primitivni tipi. Toda ne v vseh! Ker so ovojni tipi povsem običajni referenčni tipi, lahko imajo njihove spremenljivke vrednost `null`. Operacije nad takimi spremenljivkami sprožijo izjemo tipa `NullPointerException`:

```

Integer a = null;
Integer b = a + 1;    // NullPointerException

```

Izjeme ne sproži samo seštevanje, ampak klic `a.intValue()`, ki ga doda prevajalnik, da bi vrednost `a` pretvoril v tip `int` in s tem omogočil seštevanje.

V sledečem primeru je razlika med primitivnimi in ovojnimi tipi še bolj poudarjena:

```

int[] t = new int[3];           // [0, 0, 0]
System.out.println(t[0] + 1);   // 1

Integer[] u = new Integer[3];   // [null, null, null]
System.out.println(u[0] + 1);   // NullPointerException

```

Za primitivne tipe je namreč privzeta vrednost enaka 0, 0.0, `'\0'` oziroma `false`, za referenčne pa `null`. V podobno past se lahko ujamemo, če spremenljivka tipa `Integer` nastopa kot atribut objekta.

**Naloga 7.28** Kako je videti sledeča koda, potem ko prevajalnik vanjo doda klice vseh »implicitnih« metod?

```
Double a = 2.0;
double b = 3.0;
Integer c = 4;
Double d = a + b * c;
System.out.println(d);
```

Svojo rešitev lahko deloma preverite s pomočjo programa javap, ki v človeku prijaznejši obliki prikaže vsebino datoteke s končnico .class. Na primer, vsebino datoteke *Razred.class* izpišemo takole:

```
javap -c Razred
```

Če bi vsebino želeli razumeti, bi se morali poglobiti v (ne posebej zapleten) jezik javanskega navideznega stroja, klicane metode pa so jasno navedene.

**Naloga 7.29** Napišite metodo

```
public static Number maksimum(Number[] stevila)
```

ki vrne kazalec na tisti objekt v podani tabeli, ki hrani največje število. Razred *Number* je abstraktni nadrazred številskih ovojnih razredov. Upoštevajte, da lahko vsak objekt hrani vrednost kateregakoli od šestih številskih tipov. Kazalci v tabeli lahko imajo tudi vrednost *null*. (Če so vsi kazalci enaki *null*, naj metoda vrne *null*.)

**Naloga 7.30** Napišite metodo

```
public static String vNiz(Object[] obj)
```

ki vrne vsebino podane tabele v obliki niza. Upoštevajte, da lahko tabela vsebuje vrednosti *null* in kazalce na druge tabele (ki lahko prav tako vsebujejo kazalce na tabele itd.).

Metoda *vNiz* mora vrniti natanko tak niz kot metoda *Arrays.deepToString*. Na primer, koda

```
Object[] t = {
    "java",
    new int[]{-5, 6},
    new Object[0],
    null,
    42,
    new Cas[]{new Cas(10, 35), new Cas(14, 20)},
```

```

    new Float[][]{{1.0f, null, 3.0f}, {-1.0f, -4.0f}},
    new boolean[][]{{true, false}, null, {false}}
};
System.out.println(vNiz(t));

```

naj izpiše (v eni vrstici, prelome vrstic smo naknadno vstavili) sledeče:

```

[java, [-5, 6], [], null, 42, [10:35, 14:20],
[[1.0, null, 3.0], [-1.0, -4.0]],
[[true, false], null, [false]]]

```

Namig: pomagajte si z operatorjem `instanceof` in metodo `Arrays.toString`. Upoštevajte tudi dejstvo, da lahko kazalce na tabele z elementi referenčnih tipov (to vključuje tudi, recimo, tabelo tipa `boolean[][]`, saj je `boolean[]` referenčni tip) pretvorite v tip `Object[]` ali `Object`, kazalce na tabele z elementi primitivnih tipov pa zgolj v tip `Object`.

## 7.6 Vektor z elementi poljubnih referenčnih tipov

V razdelku 6.10 smo napisali razreda `VektorInt` in `VektorString`, ki predstavljata vektor (»raztegljivo« tabelo) z elementi tipa `int` oziroma `String`. Z minimalnimi spremembami ju bomo prilagodili tako, da bodo objekti nastalega razreda lahko hranili elemente poljubnih referenčnih tipov. Atribut elementi deklariramo kot tabelo tipa `Object[]` (namesto `int[]` oziroma `String[]`) in ustrezno prilagodimo glave metod, to pa je že skoraj vse.

```

// koda 7.12 (dedovanje/vektor/Vektor.java)
public class Vektor {
    private static final int ZACETNA_KAPACITETA = 10;

    private Object[] elementi;
    private int stElementov;

    public Vektor() {
        this(ZACETNA_KAPACITETA);
    }

    public Vektor(int kapaciteta) {
        this.elementi = new Object[kapaciteta];
        this.stElementov = 0;
    }
}

```

```

public int steviloElementov() {
    return this.stElementov;
}

public Object vrni(int indeks) {
    return this.elementi[indeks];
}

public void nastavi(int indeks, Object vrednost) {
    this.elementi[indeks] = vrednost;
}

public void dodaj(Object vrednost) {
    this.poPotrebiPovecaj();
    this.elementi[this.stElementov] = vrednost;
    this.stElementov++;
}

public void vstavi(int indeks, Object vrednost) {
    this.poPotrebiPovecaj();
    for (int i = this.stElementov - 1; i >= indeks; i--) {
        this.elementi[i + 1] = this.elementi[i];
    }
    this.elementi[indeks] = vrednost;
    this.stElementov++;
}

public void odstrani(int indeks) {
    for (int i = indeks; i < this.stElementov - 1; i++) {
        this.elementi[i] = this.elementi[i + 1];
    }
    this.stElementov--;
}

private void poPotrebiPovecaj() {
    if (this.stElementov >= this.elementi.length) {
        Object[] stariElementi = this.elementi;
        this.elementi = new Object[2 * stariElementi.length];
        for (int i = 0; i < this.stElementov; i++) {
            this.elementi[i] = stariElementi[i];
        }
    }
}

```



```

    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (int i = 0; i < this.stElementov; i++) {
            if (i > 0) {
                sb.append(", ");
            }
            sb.append(this.elementi[i]);
        }
        sb.append("]");
        return sb.toString();
    }
}

```

V vektor lahko sedaj dodajamo elemente poljubnih referenčnih tipov:

```

// koda 7.13 (dedovanje/vektor/TestVektor.java)
Vektor vektor = new Vektor();
vektor.dodaj("Dober dan!");
vektor.dodaj(new Cas(10, 35));
vektor.dodaj(42);
Object obj0 = vektor.vrni(0);
Object obj1 = vektor.vrni(1);
Object obj2 = vektor.vrni(2);

```

Kazalec `obj0` pripada tipu `Object`, objekt, na katerega kaže, pa tipu `String`. Zato ga lahko pretvorimo v tip `String` in tako pridobimo dostop do metod tega razreda. Podobno seveda velja za kazalca `obj1` in `obj2`.

```

String s = (String) obj0;
Cas c = (Cas) obj1;
Integer n = (Integer) obj2;

```

Sedaj lahko pišemo, recimo, `c.vrniUro()`, medtem ko izraz `obj1.vrniUro()` povzroči napako pri prevajanju, saj razred `Object` nima metode `vrniUro`.

Če želimo nad kazalcem, ki ga vrne metoda `vrni`, uporabiti izrecno pretvorbo tipa, moramo poznati tip objekta, na katerega kaže ta kazalec. V primeru napačne pretvorbe se sproži izjema tipa `ClassCastException`:

```

Vektor vektor = new Vektor();
vektor.dodaj("Dober dan!");

```

```
Integer n = (Integer) vektor.vrni(0); // ClassCastException
```

Razred Vektor sedaj ponuja vse tisto, kar ponujajo običajne tabele, poleg tega pa še precej več. Kljub temu to še ni naša zadnja različica razreda za predstavitev vektorja. V poglavju 8 bomo predstavili t.i. generični razred Vektor<T>, ki nam bo omogočal, da tip elementov vektorja podamo kot argument razreda. Na primer, objekt tipa Vektor<Integer> bo hranil elemente tipa Integer, objekt tipa Vektor<Cas> bo hranil elemente tipa Cas itd.

**Naloga 7.31** Razred Glavni v hierarhiji likov dopolnite z metodo

```
public static Vektor vecji(Lik[] liki, Lik meja)
```

ki vrne vektor likov, ki imajo večjo ploščino od lika meja.

## 7.7 Slovar

Pri tabeli in vektorju so elementi dostopni prek indeksov. Včasih pa je bolj smiselno dostop prek vrednosti drugega tipa. Na primer, pri telefonskem imeniku nas ponavadi zanima telefonska številka osebe s podanim imenom in priimkom, ne številka (denimo) petstosedeminštiridesete osebe. Kombinacija imena in priimka je *ključ* za dostop do iskane *vrednosti* — telefonske številke, ki pripada temu ključu. Podobno velja za jezikovne slovarje: do vsebine (prevoda ali razlage), povezane z določeno besedo, dostopamo neposredno z besedo, ne prek njenega indeksa v slovarju. Ključ je potemtakem beseda, pripadajoča vrednost pa njen prevod ali razlaga.

Vsebovalniku, v katerem do vsebine dostopamo prek ključa poljubnega tipa, pravimo *slovar*. Slovar potemtakem — v takšni ali drugačni obliki — hrani *pare ključ-vrednost*. V naši implementaciji bodo tako ključi kot vrednosti objekti poljubnega tipa, zato bomo oboje predstavili s spremenljivko tipa `Object`. Slovar bomo predstavili kot objekt razreda `Slovar`, ki bo v začetni verziji ponujal samo konstruktor in dve metodi:

- `public Slovar()`

Ustvari prazen slovar.

- `public void shrani(Object kljuc, Object vrednost)`

Če slovar še ne vsebuje podanega ključa (oziroma, natančneje, para ključ-vrednost, kjer je ključ enak objektu, na katerega kaže kazalec kljuc), potem metoda vanj shrani nov par ključ-vrednost, sicer pa zamenja vrednost, povezano s podanim ključem.

- `public Object vrni(Object kljuc)`

Vrne vrednost, ki pripada podanemu ključu, oziroma `null`, če slovar ne vsebuje podanega ključa.

Preden se lotimo programiranja naštetih operacij, si oglejmo preprost primer uporabe. Izdelajmo slovar, v katerem bo ključ ime države, vrednost pa število njenih sosed, in vanj dodajmo nekaj parov ključ-vrednost:

```
// koda 7.14 (dedovanje/slovar/TestSlovar.java)
Slovar drzavaVSosede = new Slovar();
drzavaVSosede.shrani("Slovenija", 4);
drzavaVSosede.shrani("Avstrija", 8);
drzavaVSosede.shrani("Češka", 4);
drzavaVSosede.shrani("Francija", 8);
drzavaVSosede.shrani("Italija", 6);
drzavaVSosede.shrani("Slovaška", 5);
drzavaVSosede.shrani("Švica", 5);
```

Povprašajmo naš slovar, koliko sosed imata Avstrija in Nemčija:

```
System.out.println(drzavaVSosede.vrni("Avstrija")); // 8
System.out.println(drzavaVSosede.vrni("Nemčija")); // null
```

Ker ključ Nemčija v slovarju ne obstaja, je rezultat drugega klica metode vrni enak `null`.

Recimo, da bi ob avstrijskih mejah nastala še ena država.<sup>4</sup> V slovarju bi to spremembo zabeležili preprosto tako, da bi ključu Avstrija priredili novo vrednost (torej 9):

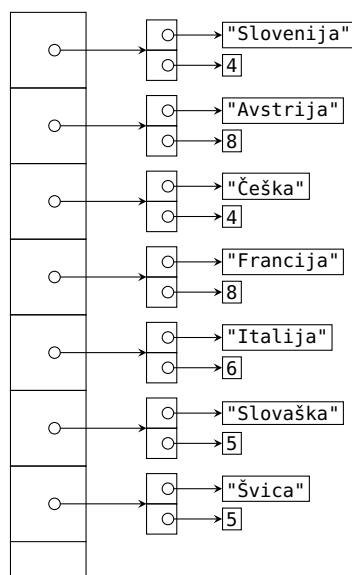
```
drzavaVSosede.shrani("Avstrija", 9);
System.out.println(drzavaVSosede.vrni("Avstrija")); // 9
```

### 7.7.1 Naivna implementacija slovarja

S kakšno podatkovno strukturo bi predstavili (*implementirali*) slovar? Ena od možnosti bi bila tabela parov ključ-vrednost. Pare bi predstavili kot objekte z dvema atributoma (ključ in vrednost, oba tipa `Object`), tabela pa bi seveda hranila kazalce na take objekte. Podobno kot vektor bi se tabela po potrebi »raztezala« (ko bi se napolnila, bi njene elemente skopirali v novo, večjo tabelo). Slika 7.4 prikazuje vsebino tabele po izvedbi kode 7.14.

Če hranimo trenutno število parov ključ-vrednost, potem opisana tabela omogoča učinkovito dodajanje novih parov. Ti se preprosto dodajo na konec tabele. Iskanje

<sup>4</sup>V času pisanja te knjige se to zdi malo verjetno, četudi se semtertja na Štajerskem sliši kak separatistični glas ...



Slika 7.4 Naivna implementacija slovarja.

parov s podanim ključem pa je bistveno počasnejše, saj se moramo v povprečju sprehoditi čez polovico tabele (če je ključ prisoten) oziroma čez celotno tabelo (če ključa v tabeli ni). V naslednjem razdelku bomo zato spoznali podatkovno strukturo, pri kateri sta obe operaciji učinkoviti.

**Naloga 7.32** Napišite razred `Slovar`, ki slovar predstavi z opisano tabelo parov.

**Naloga 7.33** V opisani implementaciji slovarja je dodajanje na konec tabele parov učinkovito, iskanje pa neučinkovito. Kako bi dosegli, da bi bilo dodajanje neučinkovito, iskanje pa učinkovito?

### 7.7.2 Implementacija slovarja z zgoščeno tabelo

Medtem ko je sprehod po tabeli počasen, je dostop do elementa na podanem indeksu izjemno učinkovit, saj gre za operacijo, ki jo strojna oprema neposredno podpira. Bi lahko to dejstvo izkoristili tudi tedaj, ko do vrednosti v tabeli dostopamo prek ključev poljubnega tipa namesto prek indeksov? Lahko, če ključ pretvorimo v indeks. Temu je namenjena metoda `hashCode`, ki za poljuben objekt izračuna celo število. Tega števila sicer ne moremo neposredno uporabiti kot indeks v tabelo, saj se lahko nahaja kjerkoli znotraj intervala tipa `int`, vendar pa ga lahko s pomočjo ostanka pri deljenju pretvorimo v število med  $0$  in  $n - 1$ , kjer je  $n$  dolžina tabele.

Opisano idejo bomo uresničili z razredom `Slovar`. V razredu bomo deklarirali atribut `tabela`, ki bo kazal na tabelo vnaprej določene (in fiksne) dolžine. S sledečo metodo bomo podani ključ pretvorili v indeks v tabelo `tabela`:

```
private int indeks(Object kljuc) {
    int n = this.tabela.length;
    return ((kljuc.hashCode() % n) + n) % n;
}
```

Zakaj ne moremo indeksa izračunati preprosto po formuli `kljuc.hashCode() % n`? Zato, ker je v javi ostanek pri deljenju negativnega števila s pozitivnim številom negativen. No, če ostanku prištejemo  $n$  in nato še enkrat izračunamo ostanek pri deljenju z  $n$ , pa zanesljivo pristanemo v intervalu  $[0, n - 1]$ .

Recimo, da je dolžina tabele `tabela` enaka 7. Za ključ (niz) `Slovenija` je rezultat metode `hashCode` enak `-1541319689`. Po deljenju s 7 dobimo ostanek 0, tak rezultat pa dobimo tudi po prištevanju števila 7 in ponovnem računanju ostanka. Ključ `Slovenija` se potemtakem preslika v indeks 0. Ključ `Avstrija` se preslika v indeks 4 (`-1927211996` → `-3` → 4), ključ `Češka` pa v indeks 2. Tabela 7.3 prikazuje indekse, v katere se preslikajo posamezni ključi iz kode 7.14.

**Tabela 7.3** Indeksi posameznih ključev pri tabeli dolžine 7.

Ključ	Indeks
Slovenija	0
Avstrija	4
Češka	2
Francija	2
Italija	3
Slovaška	2
Švica	4

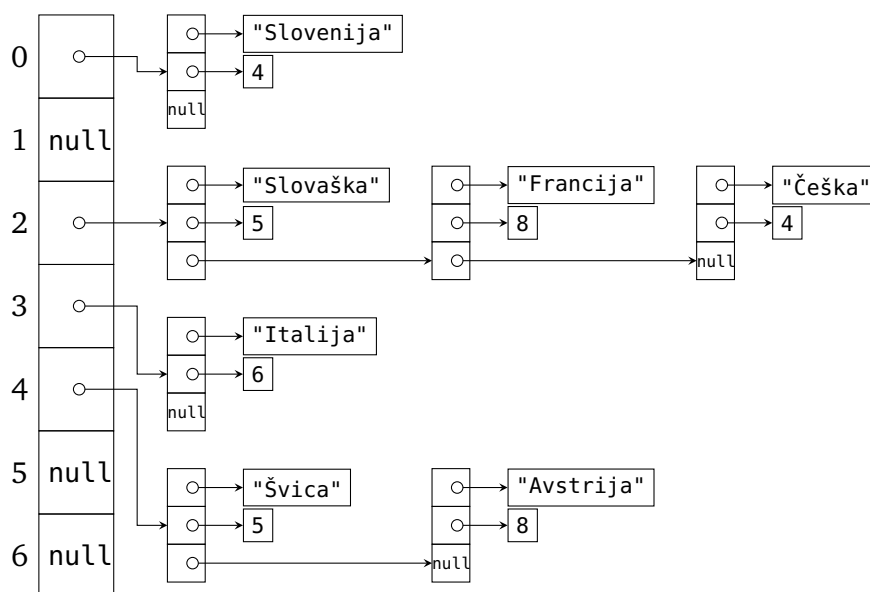
Kaj storimo, če se več ključev preslika v isti indeks? Vidimo, da se to zgodi tudi v našem primeru. Če bi bila tabela večja, bi bil ta problem manj verjeten, ne bi pa povsem izginil, saj je število potencialnih ključev (možnih nizov) bistveno večje od števila celic še tako velike tabele. Problem bomo rešili tako, da bomo v vsaki celici tabele `tabela` hranili kazalec na začetek *povezanega seznama* vseh parov ključ-vrednost, pri katerih se ključi preslikajo v indeks te celice. V našem primeru bo, denimo, celica z indeksom 2 vsebovala kazalec na začetek povezanega seznama s pari (Češka, 4), (Francija, 8) in (Slovaška, 5), saj metoda `indeks` vrne 2 za ključe Češka, Francija in Slovaška.

Kaj je povezan seznam? Gre za podatkovno strukturo, ki — podobno kot tabela — hrani zaporedje elementov. V nasprotju s tabelo, kjer so elementi shranjeni na zaporednih pomnilniških naslovih, neposredno dostopnih prek indeksov, so pri povezanem seznamu elementi (oziroma *vozlišča*, tj. elementi s pripadajočimi pove-

zovalnimi kazalci) med seboj povezani s kazalci, zato ni nujno, da so shranjeni na zaporednih naslovih.

V našem primeru bo vsaka celica tabele *tabela* hranila kazalec na prvo vozlišče povezanega seznama, vsako vozlišče seznama pa bo hranilo enega od parov ključ-vrednost (tj. dva kazalca tipa `Object`) in kazalec na naslednje vozlišče povezanega seznama. Pri zadnjem vozlišču povezanega seznama bo ta kazalec enak `null`.

Funkcija, ki poljuben objekt preslika na omejen celoštevilski interval, se imenuje *zgoščevalna funkcija* (angl. hash function), tabela povezanih seznamov, katerih indekse računamo s pomočjo zgoščevalne funkcije, pa *zgoščena tabela* (angl. hash table). Slika 7.5 prikazuje celotno zgoščeno tabelo za primer v kodi 7.14.



Slika 7.5 Zgoščena tabela (dolžine 7) po izvedbi kode 7.14.

Vozlišča povezanega seznama bodo predstavljena kot objekti tipa `Vozlisce`. Razred `Vozlisce` potemtakem vsebuje dva atributa tipa `Object`, ki predstavljata ključ oziroma pripadajočo vrednost, ter atribut tipa `Vozlisce`, ki hrani kazalec na naslednje vozlišče povezanega seznama. Razred `Vozlisce` bomo definirali kot *statični notranji razred* v razredu `Slovar`. Statični notranji razredi se obnašajo popolnoma enako kot običajni razredi, le da so definirani znotraj nekega drugega razreda; če jih deklariramo z atributom `private`, so za druge razrede nedostopni, sicer pa izven zunanjega razreda do njih dostopamo z zapisom `Zunanji.Notranji`. Razred `Vozlisce` bo `privaten`, saj sodi med implementacijske podrobnosti razreda `Slovar`.

```
// koda 7.15 (dedovanje/slovar/Slovar.java)
public class Slovar {
    private static class Vozlisce {
```

```

    Object kljuc;
    Object vrednost;
    Vozlisce naslednje;

    Vozlisce(Object kljuc, Object vrednost, Vozlisce naslednje) {
        this.kljuc = kljuc;
        this.vrednost = vrednost;
        this.naslednje = naslednje;
    }
}
...
}

```

Ker bo razred Vozlisce dostopen samo znotraj razreda Slovar, njegovih atributov nima smisla opremljati z dostopnimi določili. Konstruktor prav tako ne bi bil nujen, iz praktičnih razlogov pa smo ga vseeno napisali.

Atribut tabela v razredu Slovar bo po novem tipa Vozlisce[], saj zgoščena tabela vsebuje kazalce na začetna vozlišča povezanih seznamov. V razredu Slovar definiramo dva konstruktorja, ki izdelata zgoščeno tabelo privzete oziroma podane velikosti. O problemu izbire velikosti zgoščene tabele bomo govorili malo kasneje.

```

public class Slovar {
    private static class Vozlisce {
        ...
    }

    private static final int VELIKOST_TABELE = 97;
    private Vozlisce[] tabela;

    public Slovar() {
        this(VELIKOST_TABELE);
    }

    public Slovar(int velikostTabele) {
        this.tabela = new Vozlisce[velikostTabele];
    }
    ...
    private int indeks(Object kljuc) {
        int n = this.tabela.length;
        return ((kljuc.hashCode() % n) + n) % n;
    }
}

```

Metodi shrani in vrni si pomagata s privatno metodo poisci, ki vrne kazalec na vozlišče, ki vsebuje podani ključ, oziroma null, če takega ključa ni. Metoda poisci najprej izračuna indeks, ki pripada podanemu ključu, nato pa se sprehodi po povezanem seznamu, ki se prične v celici tabele na dobljenem indeksu. Za vsako vozlišče seznama preveri, ali je objekt, na katerega kaže atribut kljuc v vozlišču, enak (metoda equals) podanemu ključu. Če je, smo iskano vozlišče našli, sicer pa poskusimo z naslednjim vozliščem seznama (če obstaja).

```
// koda 7.16
private Vozlisce poisci(Object kljuc) {
    int indeks = this.indeks(kljuc);
    Vozlisce vozlisce = this.tabela[indeks];
    while (vozlisce != null && !vozlisce.kljuc.equals(kljuc)) {
        vozlisce = vozlisce.naslednje;    // (1)
    }
    return vozlisce;
}
```

Po povezanem seznamu se sprehodimo s pomožnim kazalcem vozlisce. Ta kazalec na začetku kaže na prvo vozlišče seznama, v vsaki iteraciji zanke pa ga prestavimo na naslednje vozlišče (vrstica (1)).

Metoda vrni je sedaj mačji kašelj:

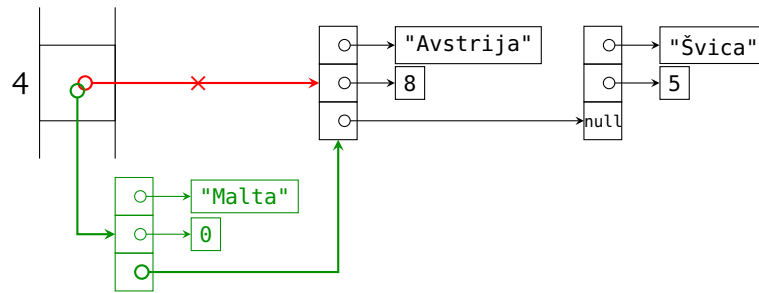
```
public Object vrni(Object kljuc) {
    Vozlisce vozlisce = this.poisci(kljuc);
    if (vozlisce == null) {
        return null;
    }
    return vozlisce.vrednost;
}
```

Metoda shrani preveri, ali vozlišče s podanim ključem že obstaja. Če obstaja, posodobi njegov atribut vrednost, sicer pa mora v povezani seznam na ustreznem indeksu vstaviti nov par ključ-vrednost. V povezani seznam je novo vozlišče najlažje vstaviti na začetek: trenutno začetno vozlišče seznama postane naslednik novega vozlišča, to pa postane novo začetno vozlišče (slika 7.6). Opisani postopek deluje tudi tedaj, ko je povezani seznam prazen.

```
public void shrani(Object kljuc, Object vrednost) {
    Vozlisce vozlisce = this.poisci(kljuc);
    if (vozlisce != null) {    // vozlišče že obstaja
        vozlisce.vrednost = vrednost;
    } else {    // vozlišče še ne obstaja

```





**Slika 7.6** Vstavljanje novega vozlišča na začetek seznama. (Niz Malta se preslika v indeks 4.)

```
int indeks = this.indeks(kljuc);
vozlisce = new Vozlisce(kljuc, vrednost, this.tabela[indeks]);
this.tabela[indeks] = vozlisce;
}
}
```

Sedaj razumemo, zakaj mora biti metoda `hashCode` usklajena z metodo `equals`: če bi lahko metoda `hashCode` za dva enaka ključa vrnila različna rezultata, bi lahko ključ iskali v napačnem povezanem seznamu. Zato je nujno, da v vsakem razredu, za katerega želimo imeti možnost, da njegovi objekti nastopajo kot ključi v slovarju, zagotovimo skladnost metod `hashCode` in `equals`. (To lahko storimo tudi tako, da metod ne redefiniramo, seveda pa to pomeni, da bodo potem tudi ločeni objekti z enako vsebino med seboj obravnavani kot različni — npr. objekta `new String("Slovenija")` in `new String("Slovenija")`.)

Kot smo že povedali, ni nujno, da imajo različni objekti različne vrednosti metode `hashCode` (tega priporočila ponavadi niti ni mogoče v celoti udejanjiti), kljub temu pa je smiselno, da to lastnost zagotovimo za čimveč parov objektov, saj bo verjetnost sovpadanja indeksov tako manjša.

Kakšna naj bo dolžina tabele `tabela` (torej število povezanih seznamov)? Če je tabela premajhna, se bo veliko ključev preslikalo v isto celico, zato bo iskanje manj učinkovito. Če je tabela prevelika, bo verjetnost sovpadanja indeksov manjša, bo pa zato več neizkoriščenega prostora.<sup>5</sup> Po nenapisanem pravilu naj bi bila dolžina tabele za približno 30% večja od pričakovanega števila elementov v njej, poleg tega pa naj bi bila praštevilo, saj se ključi na ta način enakomerneje porazdelijo po indeksih.

Problem izbire velikosti tabele lahko rešimo s podobnim trikom kot pri vektorju — pričnemo z razmeroma majhno tabelo in jo kasneje »povečujemo«, tako da vsakokrat, ko razmerje med številom shranjenih parov in dolžino tabele preseže določen

<sup>5</sup>To je še en primer »zakona o ohranitvi težav«. V računalništvu takih dilem (angl. *tradeoff*) kar mrgoli.

prag, izdelamo novo, večjo tabelo. Seveda pa moramo ob vsaki taki operaciji za vsak shranjen par ključ-vrednost na novo izračunati indeks in ga umestiti v ustrezni povezani seznam.

**Naloga 7.34** Kolikšno je teoretično število različnih nizov v javi, če lahko imamo na vsakem mestu v nizu  $2^{16}$  različnih znakov, največja možna dolžina niza pa znaša  $2^{31} - 1$ ?

**Naloga 7.35** V katerih primerih bi izraz `!vozlisce.kljuc.equals(kljuc)` v pogojnem stavku v kodi 7.16 lahko nadomestili z izrazom `vozlisce.kljuc != kljuc`?

**Naloga 7.36** Razred `Slovar` dopolnite z metodo

```
public void odstrani(Object kljuc)
```

ki iz slovarja odstrani par ključ-vrednost, ki pripada podanemu ključu. Če ključa ni, naj se ne zgodi nič.

**Naloga 7.37** Razred `Slovar` dopolnite z metodo

```
private void povecaj()
```

ki tabelo `tabela` »raztegne« za faktor 2, ko trenutno število parov ključ-vrednost v slovarju preseže tri četrtine trenutne dolžine tabele `tabela`.

### 7.7.3 Primer uporabe slovarja

**Primer 7.1.** Napišimo program, ki deluje kot preprost telefonski imenik. Program povpraša uporabnika po imenu, nato pa preveri, ali za vneseno ime že obstaja shranjena telefonska številka. Če obstaja, jo izpiše, sicer pa povpraša uporabnika po telefonski številki in jo shrani. Opisani postopek se ponavlja, dokler uporabnik ne vnese praznega imena. Na primer:

```
Vnesite ime: Janez
Osebe Janez še ni v imeniku.
Vnesite telefonsko številko: 041 234 567

Vnesite ime: Mojca
Osebe Mojca še ni v imeniku.
Vnesite telefonsko številko: 09 876 543

Vnesite ime: Janez
Telefonska številka: 041 234 567
```

```
Vnesite ime: Mojca
Telefonska številka: 09 876 543

Vnesite ime: (prazno)
```

*Rešitev.* Program na začetku izdela slovar za preslikavo vnesenih imen v telefonske številke. Nato vstopi v zanko za komunikacijo z uporabnikom. Po vnosu imena preveri, ali ime že obstaja kot ključ v slovarju. Če obstaja, izpiše pripadajočo vrednost (telefonsko številko), sicer pa omogoči uporabniku vnos telefonske številke. Telefonsko številko shrani v slovar kot vrednost, ki pripada vnesenemu imenu.

```
// koda 7.17 (dedovanje/slovar/TelefonskiImenik.java)
import java.util.Scanner;

public class TelefonskiImenik {
    public static void main(String[] args) {
        Slovar ime2stevilka = new Slovar();
        Scanner sc = new Scanner(System.in);
        System.out.print("Vnesite ime: ");
        String ime = sc.nextLine();

        while (ime.length() > 0) {
            String stevilka = (String) ime2stevilka.vrni(ime); // (1)
            if (stevilka == null) {
                System.out.printf("Osebe %s ni v imeniku.%n", ime);
                System.out.print("Vnesite telefonsko številko: ");
                stevilka = sc.nextLine();
                ime2stevilka.shrani(ime, stevilka);
            } else {
                System.out.printf("Telefonska številka: %s%n",
                                   stevilka);
            }
            System.out.println();
            System.out.print("Vnesite ime: ");
            ime = sc.nextLine();
        }
    }
}
```

Če želimo rezultat klica metode vrni (vrstica (1)) prirediti spremenljivki tipa `String`, moramo uporabiti izrecno pretvorbo tipa. Ker telefonske številke, ki jih shranjujemo

v slovar, izdelujemo kot objekte razreda `String`, pretvorba ne bo sprožila izjeme tipa `ClassCastException`. □

**Naloga 7.38** Bi se program še vedno pravilno prevedel in izvedel, če bi vrstico (1) zamenjali s sledečo vrstico?

```
Object stevilka = ime2stevilka.vrni(ime);
```

**Naloga 7.39** Napišite program, ki v zanki bere besede, dokler uporabnik ne vnese besede konec. Po vsakem uporabnikovem vnosu naj program izpiše neznana, če je uporabnik do tedaj še ni vnesel, v nasprotnem primeru pa naj izpiše, pred koliko vnosi jo je uporabnik nazadnje vnesel.

## 7.8 Povzetek

- Dedovanje je relacija med razredi. Če sta razreda  $B$  in  $A$  v takšni relaciji (če je razred  $B$  »dedič« razreda  $A$ ), je razred  $B$  podrazred razreda  $A$ , razred  $A$  pa nadrazred razreda  $B$ . Rečemo tudi, da je razred  $B$  izpeljan iz razreda  $A$  oziroma da razred  $B$  razširja razred  $A$ . Namesto o podrazredih in nadrazredih lahko govorimo tudi o podtipih in nadtipih. Relacija podrazred/nadrazred oziroma podtip/nadtip je tranzitivna.
- Če razred  $B$  definiramo kot podrazred razreda  $A$ , potem razred  $B$  od razreda  $A$  podeduje vse nestatične attribute (tudi privatne) in vse nestatične metode razen privatnih. Konstruktorji se ne dedujejo.
- Podedovane metode lahko redefiniramo. Metoda, ki jo redefiniramo, mora imeti v podrazredu isti podpis, isti izhodni tip in isto ali ohlapnejše dostopno določilo.
- Vsak konstruktor se prične bodisi s stavkom `this(...)`, ki pokliče nek drug konstruktor v istem razredu, bodisi s stavkom `super(...)`, ki pokliče konstruktor nadrazreda. Če nobenega od teh stavkov ne napišemo sami, potem prevajalnik sam doda klic `super()`.
- Vrednost referenčnega tipa  $B$  lahko priredimo spremenljivki tipa  $A$  natanko tedaj, ko sta tipa  $A$  in  $B$  enaka ali pa ko je tip  $B$  podtip tipa  $A$ .
- Tip kazalca (spremenljivke referenčnega tipa) je določen z njegovo deklaracijo, tip objekta, na katerega kaže kazalec, pa je določen z razredom, ki mu pripada objekt. Tip kazalca ni nujno enak tipu objekta; lahko je tip objekta tudi podtip tipa kazalca.
- Prevajalnik upošteva le tipe kazalcev, izvajalnik pa le tipe ciljnih objektov.

- Izraz ali stavek `p.metoda(...)` pokliče metodo v razredu, ki mu pripada objekt, na katerega v trenutku izvajanja klica kaže kazalec `p`. Izbira klicane metode je torej določena s tipom ciljnega objekta, ne kazalca.
- Abstraktna metoda nima telesa. Tovrstne metode potrebujemo samo zaradi strogih prevajalnikovih pravil. Prevajalnik lahko namreč klic `p.metoda(...)` sprejme samo pod pogojem, da je metoda `metoda` prisotna v razredu, ki ga določa tip kazalca `p`, tudi če se bo v času izvajanja vedno poklicala ena od redefinicij metode (ker bo kazalec `p` vedno kazal na objekte podtipov svojega tipa).
- Razred, ki vsebuje vsaj eno abstraktno metodo, mora biti tudi sam deklariran kot abstrakten. Za takšne razrede velja, da ni mogoče ustvarjati njihovih objektov. Kljub temu lahko vsebujejo konstruktorje, vendar pa jih je mogoče klicati le iz konstruktorjev podrazredov.
- Z operatorjem `instanceof` lahko preverimo tip objekta, na katerega kaže kazalec.
- Če je tip kazalca `A`, tip njegovega ciljnega objekta pa `B` (pri čemer je tip `B` podtip tipa `A`), in če želimo poklicati metodo, ki obstaja v razredu `B`, v razredu `A` pa ne, si lahko pomagamo z izrecno pretvorbo tipa.
- Izrecna pretvorba `(T) p` se uspešno prevede natanko tedaj, ko je tip `T` nadtip ali podtip tipa kazalca `p` ali pa ko sta tipa enaka. Če se v času izvajanja pretvorbe izkaže, da kazalec `p` ne kaže na objekt tipa `T` ali podtipa tipa `T`, se sproži izjema tipa `ClassCastException`.
- Če je tip `B` podtip tipa `A`, je tudi tip `B[]` podtip tipa `A[]`.
- Razred `Object` je nadrazred vseh javanskih razredov. Razred med drugim vsebuje metode `toString`, `equals` in `hashCode`. Te metode pogosto redefiniramo, v razredu `Object` pa so definirane takole. Metoda `toString` vrne niz, ki podaja tip objekta `this` in število, ki ga za objekt `this` vrne metoda `hashCode`. Metoda `equals` vrne `true` natanko tedaj, ko kazalec `this` in kazalec, ki ga podamo kot argument, kažeta na isti objekt. Metoda `hashCode` vrne število, ki predstavlja identiteto objekta `this`.
- Za vsak primitivni tip imamo v javi na voljo pripadajoči referenčni tip (ime-nujemo ga ovojni tip). Objekt ovojnega tipa hrani vrednost pripadajočega referenčnega tipa. Zaradi samodejnega pretvarjanja lahko primitivne in ovojne tipe skoraj poljubno mešamo. Upoštevati moramo le možnost, da spremenljivka ovojnega tipa vsebuje vrednost `null`.

- Vsebovalnik z elementi tipa `Object` lahko hrani elemente poljubnih referenčnih tipov, vendar pa moramo pri dostopu do elementov sami poskrbeti za ustrezne pretvorbe tipov.
- Slovar je vsebovalnik, v katerem so vrednosti dostopne prek ključev poljubnega tipa. Implementiramo ga lahko z zgoščeno tabelo, ki temelji na preslikovanju ključev v indekse. Preslikovanje poteka s pomočjo metode `hashCode` in operacije ostanka pri deljenju. Zgoščena tabela je tabela kazalcev na začetke povezanih seznamov, ki hranijo pare ključ-vrednost, pri čemer za povezani seznam na indeksu  $i$  velja, da se ključi v njem preslikajo v indeks  $i$ .

### *Iz profesorjevega kabineta*

»Tale moj Jože,« nagovori docentko Javornik profesor Doberšek, »saj ne rečem, fant od fare, ampak včasih pa ga malo lomi. Ondan ga slišim, kako študentom na vajah dopoveduje, naj nizov nikar ne primerjajo z dvojnimi enačanjem, pač pa samo z metodo `equals`. Sam dobro vem, da to ni res, saj nize že od malih nog primerjam z dvojnimi enačjem. Izzval sem ga, naj požene `jshell` in odtipka

```
"test" == "test"
```

Kako je zardel, ko je dobil odgovor `true`! Seveda ni imel nobene prave razlage; zajeceljal je le nekaj v smislu »včasih to očitno kljub vsemu deluje«. Kje pa! To deluje vedno! Imam prav, Genovefa?«

»Žal samo napol, gospod profesor. Javanski izvajalnik vzdržuje množico nizov, ki nastopajo v programu kot konstante. Če se nek niz ponovi, ne izdelava njegove kopije, ampak zgolj z novim kazalcem pokaže na obstoječi objekt. Zato bo izraz `"test" == "test"` vedno imel vrednost `true`. Sedaj pa poskusite tole:

```
new String("test") == new String("test")
```

V tem primeru izvajalnik prisilimo v izdelavo dveh ločenih objektov, zato je rezultat izraza enak `false`. Enako se zgodi, če nize beremo s standardnega vhoda. Jože je ravnal čisto pravilno, ko je posvaril študente, seveda pa bi lahko poznal tudi izjemo, s katero ste mu jo zagodli.«

Popravite razred `Cas` tako, da bo v poljubnem programu za vse pare spremenljivk  $p$  in  $q$  tipa `Cas` veljalo

```
 $p == q \iff p.equals(q)$ 
```

Namig: ker vedno velja `new Cas(h, m) != new Cas(h, m)` (operator `new` vedno ustvari *nov* objekt), morate preprečiti izdelavo objektov s pomočjo operatorja `new`. To storite tako, da konstruktor opremita z dostopnim določilom `private` in napišete metodo

```
public static Cas vrniObjekt(int h, int min)
```

ki ustvari nov ali vrne obstoječi objekt.





## 8 Generiki

V poglavju 7 smo napisali razreda za predstavitev vektorjev in slovarjev, ki hranijo kazalce tipa `Object`. Ker je razred `Object` nadrazred vseh javanskih razredov, lahko taki kazalci kažejo na objekte poljubnih tipov (tudi v okviru istega vsebovalnika). V praksi pa veliko pogosteje uporabljamo vsebovalnike, pri katerih so vsi objekti istega, toda poljubnega tipa. Primer je vektor objektov tipa `Cas` ali pa slovar, ki ključne tipa `String` preslikuje v vrednosti tipa `Integer`. V takih primerih si lahko pomagamo s t.i. *generiki*, mehanizmom, ki nam omogoča, da razrede parametriziramo s tipi. Na primer, razred `Vektor` lahko parametriziramo kot `Vektor<T>`, pri čemer parameter `T` označuje tip elementov vektorja. Ob izdelavi objekta razreda `Vektor` lahko kot argument za parameter `T` podamo poljuben referenčni tip in tako dobimo vektor z elementi tipa `Integer`, vektor z elementi tipa `Cas` itd.

### 8.1 Uvod

Sledeči razred ni posebej razburljiv, za ilustracijo generikov pa nam bo dobro služil:

```
// koda 8.1 (generiki/ovojnikKlasicno/Ovojnik.java)
public class Ovojnik {
    private Object a;

    public Ovojnik(Object a) {
        this.a = a;
    }

    public Object vrni() {
        return this.a;
    }
}
```

Objekt razreda `Ovojnik` hrani (»ovija«) kazalec tipa `Object`. Ta kazalec lahko, kot vemo, kaže na objekt poljubnega tipa:

```
// koda 8.2 (generiki/ovojnikKlasicno/TestOvojnik.java)
```

```
Ovojn timer p = new Ovojn timer("dober dan");           // String
Ovojn timer q = new Ovojn timer(42);                     // Integer
Ovojn timer r = new Ovojn timer(new Cas(10, 35));        // Cas
```

Razred `Ovojn timer` je povsem splošen, ima pa dve slabosti. Prvič, če želimo kazalec, ki je shranjen v objektu tipa `Ovojn timer`, pridobiti kot vrednost »pravega« tipa, moramo ta tip poznati, saj lahko le tako uporabimo ustrezno izrecno pretvorbo:

```
// koda 8.3 (generiki/ovojnikKlasicno/TestOvojn timer.java)
String s = (String) p.vrni();
Integer n = (Integer) q.vrni();
Cas c = (Cas) r.vrni();
```

Če uporabimo napačno pretvorbo tipa, se koda prevede brez napak in opozoril, med izvajanjem pa se sproži izjema tipa `ClassCastException`. To je druga slabost razreda `Ovojn timer`: zaželeno je namreč, da morebitna neskladja tipov ujamemo že v času prevajanja.

Obe slabosti lahko odpravimo s pomočjo *generikov* — parametriziranih tipov. Pri razredu `Ovojn timer` se nam splača parametrizirati tip atributa `a`. To storimo tako, da tip `Object`, ki mu pripada atribut `a`, zamenjamo s parametrom. Ta parameter (rekli mu bomo *tipni parameter*) moramo deklarirati v glavi razreda znotraj lomljenih oklepajev:

```
// koda 8.4 (generiki/ovojnikGenericno/Ovojn timer.java)
public class Ovojn timer<T> {
    private T a;

    public Ovojn timer(T a) {    // tukaj ne pišemo Ovojn timer<T>
        this.a = a;
    }

    public T vrni() {
        return this.a;
    }
}
```

Odslej moramo pri sklicevanju na razred `Ovojn timer` dosledno navajati *tipni argument*, tj. referenčni tip, s katerim želimo nadomestiti parameter `T`. Pri kodi 8.2 ravnamo takole:

```
// koda 8.5 (generiki/ovojnikGenericno/TestOvojn timer.java)
Ovojn timer<String> p = new Ovojn timer<String>("dober dan");
Ovojn timer<Integer> q = new Ovojn timer<Integer>(42);
Ovojn timer<Cas> r = new Ovojn timer<Cas>(new Cas(10, 35));
```

Nekoliko večja količina tipkanja se nam obrestuje pri pridobivanju elementov, shranjenih v posameznih objektih tipa `Ovojniki`. Ker smo s stavkom

```
Ovojniki<String> p = new Ovojniki<String>("dober dan");
```

ustvarili objekt razreda `Ovojniki`, v katerem je tipni parameter `T` nadomeščen s tipom `String`, si lahko predstavljamo, kot da se glava metode `vrni()` pri klicu `p.vrni()` glasi

```
public String vrni()
```

Prevajalnik tako ve, da klic `p.vrni()` vrne vrednost tipa `String`, zato izrecne pretvorbe tipa ne potrebujemo več. Podobno seveda velja za klica `q.vrni()` in `r.vrni()`:

```
// koda 8.6 (generiki/ovojnikGenericno/TestOvojniki.java)
String s = p.vrni();
Integer n = q.vrni();
Cas c = r.vrni();
```

Generiki obstajajo *izključno v času prevajanja*! Izvajalnik kode, povezane z generiki, sploh ne vidi, saj jo prevajalnik v procesu *brisanja tipov* (angl. *type erasure*) samodejno pretvori: kodo 8.5 spremeni v kodo 8.2, kodo 8.6 pa v kodo 8.3. Prevajalnik torej tipni parameter `T` spremeni v tip `Object`, tipne argumente preprosto odstrani, pri klicih metode `vrni` pa doda izrecno pretvorbo v ciljni tip.

Samodejno dodana izrecna pretvorba povzroči, da prevajalnik pri obravnavi kode 8.6 preverja skladnost tipov na levi in desni strani prireditvenih stavkov. Na primer, ker je kazalec `p` tipa `Ovojniki<String>`, bi prevajalnik kodo

```
Integer n = p.vrni();
```

v procesu brisanja tipov spremenil v kodo

```
Integer n = (String) p.vrni();
```

in bi nam sporočil, da gre za napako nezdržljivosti tipov, saj kazalca tipa `String` ni mogoče prirediti spremenljivki tipa `Integer`. Neskladnost tipov bi tako zaznali že v času prevajanja, ne šele v času izvajanja.

Kaj pa primitivni tipi? Žal ne pridejo v poštev. Ker prevajalnik vse tipne parametre spremeni v tip `Object`, lahko v vlogi tipnih argumentov nastopajo zgolj referenčni tipi. Tip `Ovojniki<int>` potemtakem ni mogoč. Seveda pa lahko uporabljamo pripadajoče ovojne tipe (npr. `Integer`). Zaradi samodejnih pretvorb med primitivnimi in ovojnimi tipi to ni posebej huda ovira.

## 8.2 Generični razredi

*Generični razred* je razred, opremljen z enim ali več tipnimi parametri. Tipne parametre navedemo v glavi razreda, in sicer v lomljenih oklepajih neposredno za imenom razreda:

```
public class R<T1, T2, ..., Tn>
```

Sedaj lahko v telesu razreda po mili volji uporabljamo parametre  $T_1, T_2, \dots, T_n$ . No, ne čisto po mili volji. Pri statičnih atributih in v statičnih metodah jih ne moremo, saj so tipni parametri vezani na posamezne objekte, ne na razred kot celoto. Na primer, vsak objekt tipa `Ovojniki<T>` ima svoje konkretno nadomestilo (tipni argument) za parameter `T`.

Tipne parametre praviloma označujemo z velikimi črkami. Na primer:

```
public class Primer<T, U, V>
```

Ko želimo izdelati objekt generičnega razreda, moramo podati argumente za vse tipne parametre. Na primer:

```
Primer<String, Integer, Cas> p = new Primer<String, Integer, Cas>();
```

V tem primeru se bo parameter `T` znotraj telesa razreda `Primer` nadomestil s tipom `String`, parameter `U` s tipom `Integer`, parameter `V` pa s tipom `Cas`. Vnovič pa naj poudarimo, da te zamenjave veljajo samo v času prevajanja. V prevedeni kodi namesto vseh tipnih parametrov nastopa tip `Object`, tip `Primer<String, Integer, Cas>` pa se v procesu brisanja tipov spremeni v neparametrizirani tip `Primer`.

Mimogrede, na desni strani prireditvenega stavka, ki izdelava objekt generičnega razreda, lahko seznam tipnih argumentov nadomestimo z zapisom `<>` in si tako prihranimo nekaj mukotrpnega tipkanja. Na primer, namesto

```
Primer<String, Integer, Cas> p = new Primer<String, Integer, Cas>();
```

lahko pišemo

```
Primer<String, Integer, Cas> p = new Primer<>();
```

**Naloga 8.1** Napišite razred `Par<U, V>`, tako da bodo njegovi objekti predstavljali dvojice elementov poljubnih referenčnih tipov.<sup>1</sup> Razred naj ponuja konstruktor s parametrom tipa `U` (prva komponenta para) in parametrom tipa `V` (druga komponenta para), »getterja« za prvo oz. drugo komponento para `this`, metodo `toString` (par naj se izpiše v obliki *(prva, druga)*), metodo `equals` (para sta

<sup>1</sup>Razred `Par` bomo uporabljali tudi v nadaljevanju.

enaka natanko v primeru, če sta obe prvi in obe drugi komponenti med seboj enaki) in metodo `hashCode`, ki naj bo seveda usklajena z metodo `equals`.

Pri metodi `public boolean equals(Object obj)` boste najbrž napisali stavek

```
Par<U, V> p = (Par<U, V>) obj;
```

Ta stavek se bo sicer prevedel, vendar pa bo prevajalnik izpisal opozorilo, ki si ga boste lahko podrobneje ogledali s stikalom `-Xlint:unchecked`. Prevajalnik, kot dobro vemo, vidi zgolj tip kazalca `obj` (tj. `Object`), šele izvajalnik pa lahko preveri, ali kazalec `obj` kaže na objekt tipa `Par`. V času izvajanja pa so tipni parametri brez pomena, saj jih prevajalnik zamenja s tipom `Object`. Tipa `U` in `V` v prikazani vrstici potemtakem nimata nobenega smisla. Zato ju zamenjamo z oznako `?`:

```
Par<?, ?> p = (Par<?, ?>) obj;
```

Oznako `?` preberemo kot »poljuben (referenčni) tip«.

Podobno se zgodi pri operatorju `instanceof`. Namesto izraza

```
obj instanceof Par<U, V>
```

ki povzroči celo napako pri prevajanju, pišemo

```
obj instanceof Par<?, ?>
```

**Naloga 8.2** Naj bo »izrojeni par« `par`, ki ima obe komponenti med seboj enaki. Iz razreda `Par<U, V>` izpeljite razred `IzrojeniPar<T>`. (Napisati boste morali zgolj konstruktor!)

## 8.3 Generične metode

S tipi lahko parametriziramo tudi metodo. Na primer, če želimo napisati statično metodo `enakaVsebinsa`, ki med seboj primerja vsebino dveh objektov tipa `Ovojniki<T>`, potem moramo tudi samo metodo parametrizirati s tipom `T`:

```
// koda 8.7 (generiki/ovojnikGenericno/TestOvojniki.java)
public static <T> boolean enakaVsebinsa(Ovojniki<T> p, Ovojniki<T> q) {
    return p.vrni().equals(q.vrni());
}
```

Če bi zapis `<T>` za določilom `static` izpustili, bi prevajalnik javil napako, saj ne bi vedel, od kod se je znašel tip `T`.

Seznam tipnih parametrov podamo tik pred izhodnim tipom metode, torej za dostopnimi določili in morebitnim določilom `static`.

Pri klicu generične metode nam ponavadi ni treba podajati argumentov za tipne parametre, saj jih prevajalnik sam izlušči. Na primer, če metodo `enakaVsebina` pokličemo kot

```
// koda 8.8 (generiki/ovojnikGenericno/TestOvojn timer.java)
Ovojn timer<String> a = new Ovojn timer<>("abc");
Ovojn timer<String> b = new Ovojn timer<>("de");
System.out.println(enakaVsebina(a, b));
```

potem prevajalnik sam ugotovi, da se tipni parameter `T` nadomesti s tipom `String`. Če bi želeli eksplicitno navesti tipni argument (v praksi je to zelo redko potrebno), pa bi to storili z zapisom

```
System.out.println(R.<String>enakaVsebina(a, b));
```

pri čemer je `R` razred, v katerem se nahaja metoda `enakaVsebina`. (Razred `R` moramo navesti tudi v primeru, če metodo pokličemo znotraj razreda `R`, saj je klic `<String>enakaVsebina(a, b)` sintaktično napačen.)

### Naloga 8.3 Napišite metodo

```
public static <U, V> String zlij(U a, V b)
```

ki vrne niz oblike  $(p, q)$ , pri čemer sta  $p$  in  $q$  niza, ki ju metoda `toString` vrne za objekta `a` oziroma `b`.

**Naloga 8.4** Metodo `enakaVsebina` prepisite kot nestatično metodo v razredu `Ovojn timer`. Ali metoda v tem primeru še potrebuje zapis `<T>`?

**Naloga 8.5** Razred `Par<U, V>` dopolnite z metodo

```
public <W> Par<U, W> veriga(Par<V, W> par)
```

ki ustvari in vrne `par`, sestavljen iz prve komponente `para` `this` in druge komponente `para` `par`, če je druga komponenta `para` `this` enaka prvi komponenti `para` `par`. Če to ne drži, naj metoda vrne `null`.

## 8.4 Zgornja meja tipnega parametra

Pri uporabi generičnega razreda ali metode lahko tipni parameter `T` nadomestimo s poljubnim referenčnim tipom. Z zapisom

```
T extends R
```

pa lahko zahtevamo, da je tipni parameter `T` mogoče nadomestiti le s tipom `R` ali njegovim podtipom. Pravimo, da je tipni parameter `T` *navzgor omejen* s tipom `R`.

Kot vemo, prevajalnik vse pojavitve neomejenih tipnih parametrov zamenja s tipom `Object`. Če je tipni parameter navzgor omejen s tipom `R`, pa njegove pojavitve zamenja s tipom `R`. To pomeni, da bomo lahko nad objekti takšnega tipa `T` klicali metode razreda `R`.

Oglejmo si primer. Sledeči razred je podoben razredu `Ovojniki`, le da lahko »ovije« zgolj objekt številskega ovojnega tipa.

```
// koda 8.9 (generiki/ovojnikGenericno/StevilskiOvojniki.java)
public class StevilskiOvojniki<T extends Number> {
    private T a;

    public StevilskiOvojniki(T a) {
        this.a = a;
    }

    public T vrni() {
        return this.a;
    }
}
```

Ker vemo, da je tip `T` mogoče nadomestiti samo s tipom `Number` ali njegovim podtipom (npr. `Integer` ali `Double`), lahko na objektih tipa `T` uporabljamo metode razreda `Number` in jih, denimo, primerjamo po številskih vrednostih:

```
public class StevilskiOvojniki<T extends Number> {
    ...
    public <U extends Number> boolean jeVecjiKot(
        StevilskiOvojniki<U> drugi) {
        return this.a.doubleValue() > drugi.a.doubleValue();
    }
}
```

Metodo `jeVecjiKot` bi lahko zapisali tudi kot

```
public boolean jeVecjiKot(StevilskiOvojniki<T> drugi) {
    return this.a.doubleValue() > drugi.a.doubleValue();
}
```

vendar pa bi si s tem omejili možnosti, saj bi zahtevali, da je tipni argument pri objektu `drugi` enak kot pri objektu `this`. Sedaj pa lahko, recimo, med seboj primerjamo objekt tipa `StevilskiOvojniki<Integer>` in objekt tipa `StevilskiOvojniki<Double>`.

**Naloga 8.6** V sledečih metodah, ki primerjajo like (razdelek 7.3) po različnih kriterijih, podajte manjkajoče deklaracije tipnih parametrov, tako da bodo metode karseda splošne.

```
public static <...> int poVrsti(T a, T b) {
    return a.vrsta().compareTo(b.vrsta());
}

public static <...> int poPloscini(T a, T b) {
    return Double.compare(a.ploscina(), b.ploscina());
}

public static <...> int poSirini(T a, T b) {
    return Double.compare(a.vrniSirino(), b.vrniSirino());
}
```

## 8.5 Omejitve pri uporabi generikov

Z generiki lahko počnemo marsikaj, ne pa čisto vsega. Na primer, če je `T` tipni parameter, potem izraz

```
new T(...)
```

sproži napako pri prevajanju. Ni težko ugotoviti, zakaj. Spomnimo se, da izvajalnik namesto parametra `T` vidi tip `Object`, ne glede na tipni argument, s katerim nadomestimo parameter `T`.

Tabelo z elementi tipa `T` lahko ustvarimo le po ovinkih. Namesto stavka

```
T[] t = new T[10];
```

ki povzroči napako pri prevajanju, lahko pišemo zgolj takole:

```
T[] t = (T[]) new R[10];
```

pri čemer je `R` tip, s katerim je tip `T` navzgor omejen (to je `Object`, če z besedo `extends` ne določimo drugače). Prevajalnik nam v tem primeru izpiše opozorilo (`unchecked cast`), ki se ga žal lahko znebimo samo tako, da pred metodo, ki vsebuje »problematično« kodo, postavimo posebno oznako:

```
@SuppressWarnings("unchecked")
... metoda(...) {
    ...
    T[] t = (T[]) new R[10];
}
```



```
...
}
```

## 8.6 Generični razred Vektor

Sedaj smo opremljeni z vsem, kar potrebujemo, da različico razreda Vektor iz razdelka 7.6 nadgradimo v generično različico. Glavo razreda spremenimo v Vektor<T>, tip Object pa zamenjamo s parametrom T. Kot smo videli v prejšnjem razdelku, moramo paziti le pri izdelavi tabele.

```
// koda 8.10 (generiki/vektor/Vektor.java)
public class Vektor<T> {
    private static final int ZACETNA_KAPACITETA = 10;

    private T[] elementi;
    private int stElementov;

    public Vektor() {
        this(ZACETNA_KAPACITETA);
    }

    @SuppressWarnings("unchecked")
    public Vektor(int kapaciteta) {
        this.elementi = (T[]) new Object[kapaciteta];
        this.stElementov = 0;
    }

    public int steviloElementov() {
        return this.stElementov;
    }

    public T vrni(int indeks) {
        return this.elementi[indeks];
    }

    public void nastavi(int indeks, T vrednost) {
        this.elementi[indeks] = vrednost;
    }

    public void dodaj(T vrednost) {
        this.poPotrebiPovecaj();
    }
}
```

```

        this.elementi[this.stElementov] = vrednost;
        this.stElementov++;
    }

    public void vstavi(int indeks, T vrednost) {
        this.poPotrebiPovecaj();
        for (int i = this.stElementov - 1; i >= indeks; i--) {
            this.elementi[i + 1] = this.elementi[i];
        }
        this.elementi[indeks] = vrednost;
        this.stElementov++;
    }

    public void odstrani(int indeks) {
        for (int i = indeks; i < this.stElementov - 1; i++) {
            this.elementi[i] = this.elementi[i + 1];
        }
        this.stElementov--;
    }

    @SuppressWarnings("unchecked")
    private void poPotrebiPovecaj() {
        if (this.stElementov >= this.elementi.length) {
            T[] stari = this.elementi;
            this.elementi = (T[]) new Object[2 * stari.length];
            for (int i = 0; i < this.stElementov; i++) {
                this.elementi[i] = stari[i];
            }
        }
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (int i = 0; i < this.stElementov; i++) {
            if (i > 0) {
                sb.append(", ");
            }
            sb.append(this.elementi[i]);
        }
        sb.append("]");
    }

```

```

        return sb.toString();
    }
}

```

Razred Vektor lahko sedaj uporabljamo brez nadležnih izrecnih pretvorb tipa:

```

// koda 8.11 (generiki/vektor/TestVektor.java)
Vektor<String> besede = new Vektor<>();
Vektor<Integer> stevila = new Vektor<>();
...
String s = besede.vrni(...);
int n = stevila.vrni(...);

```

Neskladja tipov povzročijo napako že v času prevajanja, ne šele v času izvajanja:

```

besede.dodaj(42);           // napaka pri prevajanju
stevila.dodaj("abc");      // napaka pri prevajanju
...
int n = besede.vrni(...);   // napaka pri prevajanju
String s = stevila.vrni(...); // napaka pri prevajanju

```

**Naloga 8.7** Nalogo Izštevanka iz razdelka 6.10.3 rešite s pomočjo generične različice razreda Vektor.

**Naloga 8.8** Razred VektorString iz razdelka 6.10 prepisite tako, da bo deklariran kot podrazred razreda Vektor<T>.

**Naloga 8.9** V razredu Vektor redefinirajte metodi equals in hashCode. Vektorja sta enaka, če imata enako število elementov in enake istoležne elemente. Kapaciteti nista nujno enaki.

**Naloga 8.10** Napišite metodo

```
public static <T> Vektor<T> obrat(Vektor<T> vektor)
```

ki izdela in vrne nov vektor, ki vsebuje elemente podanega vektorja v obratnem vrstnem redu.

**Naloga 8.11** Napišite metodo

```
public static <T> Vektor<Vektor<T>> transpozicija(
    Vektor<Vektor<T>> matrika)
```

ki izdela in vrne transponirano različico podane pravokotne matrike (lahko predpostavite, da imajo vsi sekundarni vektorji enako število elementov). Transpozi-

cija pomeni prepis vrstic v stolpce: element na koordinatah  $(i, j)$  v izvorni matriki se v transponirani matriki preseli na koordinati  $(j, i)$ .

## 8.7 Generični razred Slovar

Pri razredu Slovar postopamo podobno kot pri razredu Vektor, le da imamo dva tipna parametra: tip ključa (K) in tip pripadajoče vrednosti (V). Tudi tokrat imamo nekaj sitnosti pri izdelavi tabele.

Ker je razred Vozlisce statičen in s tem neodvisen od objektov oklepajočega razreda Slovar, se tipna parametra K in V ne preneseta do njega. To pomeni, da moramo parametrizirati tudi razred Vozlisce. Njegova tipna parametra imata sicer enaki imeni kot tipna parametra v razredu Slovar, vendar pa sta povsem neodvisna od njiju.

```
// koda 8.12 (generiki/slovar/Slovar.java)
public class Slovar<K, V> {
    private static class Vozlisce<K, V> {
        K kljuc;
        V vrednost;
        Vozlisce<K, V> naslednje;

        Vozlisce(K kljuc, V vrednost, Vozlisce<K, V> naslednje) {
            this.kljuc = kljuc;
            this.vrednost = vrednost;
            this.naslednje = naslednje;
        }
    }

    private static final int VELIKOST_TABELE = 97;
    private Vozlisce<K, V>[] podatki;

    public Slovar() {
        this(VELIKOST_TABELE);
    }

    @SuppressWarnings("unchecked")
    public Slovar(int velTab) {
        this.podatki = (Vozlisce<K, V>[]) new Vozlisce<?, ?>[velTab];
    }

    public V vrni(K kljuc) {
```

```

        Vozlisce<K, V> vozlisce = this.poisci(kljuc);
        if (vozlisce == null) {
            return null;
        }
        return vozlisce.vrednost;
    }

    public void shrani(K kljuc, V vrednost) {
        Vozlisce<K, V> vozlisce = this.poisci(kljuc);
        if (vozlisce != null) {
            vozlisce.vrednost = vrednost;
        } else {
            int indeks = this.indeks(kljuc);
            vozlisce = new Vozlisce<>(kljuc, vrednost,
                                      this.podatki[indeks]);
            this.podatki[indeks] = vozlisce;
        }
    }

    private Vozlisce<K, V> poisci(K kljuc) {
        int indeks = this.indeks(kljuc);
        Vozlisce<K, V> vozlisce = this.podatki[indeks];
        while (vozlisce != null && !vozlisce.kljuc.equals(kljuc)) {
            vozlisce = vozlisce.naslednje;
        }
        return vozlisce;
    }

    private int indeks(K kljuc) {
        int n = this.podatki.length;
        return ((kljuc.hashCode() % n) + n) % n;
    }
}

```

Tako kot pri razredu Vektor se tudi pri razredu Slovar s parametrizacijo znebimo potrebe po izrecnih pretvorbah tipa:

```

// koda 8.13 (generiki/slovar/TestSlovar.java)
Slovar<String, Integer> s2i = new Slovar<>();
s2i.shrani("ena", 1);
s2i.shrani("dve", 2);
s2i.shrani("tri", 3);

```

```
int n = s2i.vrni("tri");
System.out.println(n);    // 3

Slovar<Cas, String> urnik = new Slovar<>();
urnik.shrani(new Cas(6, 30), "zajtrk");
urnik.shrani(new Cas(12, 30), "kosilo");
urnik.shrani(new Cas(19, 0), "večerja");
String obrok = urnik.vrni(new Cas(12, 30));
System.out.println(obrok);    // kosilo
```

**Naloga 8.12** Nalogo iz razdelka 7.7.3 rešite s pomočjo generičnega razreda Slovar.

**Naloga 8.13** Napišite metodo

```
public static <K, V> Slovar<K, V> izVektorjaParov(
    Vektor<Par<K, V>> vektor)
```

ki vrne slovar, izdelan na podlagi podanega vektorja parov, pri čemer prve komponente parov določajo ključe, druge pa vrednosti. Razred Par ste napisali v okviru naloge 8.1. Lahko predpostavite, da nobena dvojica parov v podanem vektorju nima enakih prvih komponent.

**Naloga 8.14** Napišite metodo

```
public static <K, V> Vektor<K> kljuciZaVrednost(
    Slovar<K, V> slovar, V vrednost)
```

ki vrne vektor ključev, ki jih podani slovar preslika v podano vrednost.

**Naloga 8.15** Napišite metodo

```
public static <K, V> Slovar<V, Vektor<K>> obrat(Slovar<K, V> slov)
```

ki vrne nov slovar, ki vsako vrednost  $v$  iz podanega slovarja preslika v vektor ključev, ki se v podanem slovarju preslikajo v vrednost  $v$ .

## 8.8 Nadomestni znak

V razdelku 7.3.9 smo spoznali, da so tabele kovariantne: če je tip  $Q$  podtip tipa  $P$ , je tip  $Q[]$  prav tako podtip tipa  $P[]$ . Pri generikih pa to ne velja: tipa  $Vektor<P>$  in  $Vektor<Q>$  v času prevajanja nimata nič skupnega. (V času izvajanja gre za en in isti tip, saj prevajalnik v procesu brisanja tipov oba tipa zamenja s tipom  $Vektor$ .)

Pojasnimo, zakaj generiki niso kovariantni. Če bi bil, denimo, tip `Vektor<Integer>` podtip tipa `Vektor<Object>`, bi se sledeča koda prevedla:

```
Vektor<Integer> vi = new Vektor<>();
vi.dodaj(3);
Vektor<Object> vo = vi;    // ta stavek se ne prevede
vo.dodaj("Dober dan!");
```

Ker kazalca `vo` in `vi` kažeta na isti objekt, bi na ta način v vektor tipa `Vektor<Integer>` dodali element tipa `String`. Načrtovalci jave so takšne obvide želeli preprečiti, zato so se pri generikih odpovedali kovariantnosti.

Vse lepo in prav, toda — bi nam kovariantnost sploh kdaj koristila? Da, dejansko bi v nekaterih primerih prišla prav. Recimo, sledeča metoda sprejme samo vektor tipa `Vektor<Number>`, če bi bili generiki kovariantni, pa bi sprejela tudi vektorje tipa `Vektor<Integer>`, `Vektor<Double>` itd.:

```
// koda 8.14 (generiki/nadomestniZnak/Povprecje.java)
public static double povprecje(Vektor<Number> vektor) {
    double vsota = 0;
    int stElementov = vektor.steviloElementov();
    for (int i = 0; i < stElementov; i++) {
        vsota += vektor.vrni(i).doubleValue();
    }
    return vsota / stElementov;
}
```

K sreči pa obstaja rešitev: nadomestni znak. Naj bo  $P$  generični tip (npr. `Vektor`),  $Q$  pa poljuben referenčni tip. Nadomestni znak (znak `?`) lahko uporabimo na tri načine:

- Zapis  $P<?>$  določa, da je tip  $P$  lahko parametriziran s katerimkoli referenčnim tipom. Za poljuben referenčni tip  $R$  velja, da je tip  $P<R>$  podtip tipa  $P<?>$ .
- Zapis  $P<? \text{ extends } Q>$  določa, da je tip  $P$  lahko parametriziran s tipom  $Q$  ali njegovim podtipom. Tip  $P<R>$  je podtip tipa  $P<? \text{ extends } Q>$  natanko v primeru, če je  $R = Q$  ali če je tip  $R$  podtip tipa  $Q$ .
- Zapis  $P<? \text{ super } Q>$  določa, da je tip  $P$  lahko parametriziran s tipom  $Q$  ali njegovim nadtipom. Tip  $P<R>$  je podtip tipa  $P<? \text{ super } Q>$  natanko v primeru, če je  $R = Q$  ali če je tip  $R$  nadtip tipa  $Q$ .

Tip `Vektor<Number>` je potemtakem podtip vseh naslednjih tipov:

- `Vektor<?>`

- `Vektor<? extends Object>`
- `Vektor<? extends Number>`
- `Vektor<? super Number>`
- `Vektor<? super Integer>`, `Vektor<? super Double>` itd.

Vsi naslednji stavki se zato prevedejo:

```
Vektor<Number> v = new Vektor<>();
Vektor<?> v1 = v;
Vektor<? extends Object> v2 = v;
Vektor<? extends Number> v3 = v;
Vektor<? super Number> v4 = v;
Vektor<? super Integer> v5 = v;
```

Če bi gornjo kodo podaljšali z naslednjimi stavki, pa bi prav vsak od njih povzročil napako pri prevajanju:

```
Vektor<? extends Integer> v6 = v;
Vektor<? super Object> v7 = v;
Vektor<Object> v8 = v;
Vektor<Number> v9 = v3;
```

Popravimo metodo povprecje tako, da ji bomo lahko podtaknili vektor tipa `Vektor<Number>`, `Vektor<Integer>`, `Vektor<Double>` itd.:

```
public static double povprecje(Vektor<? extends Number> vektor) {
    double vsota = 0;
    int stElementov = vektor.steviloElementov();
    for (int i = 0; i < stElementov; i++) {
        vsota += vektor.vrni(i).doubleValue();
    }
    return vsota / stElementov;
}
```

Oglejmo si še en primer. Recimo, da izvršimo sledečo kodo:

```
Vektor<Number> v = new Vektor<>();
v.dodaj(10);
Vektor<? extends Number> ve = v;
Vektor<? super Number> vs = v;
```

Sedaj se bo sledeča koda prevedla (in izvedla) ...



```
Number stevilo = ve.vrni(0);
vs.dodaj(20);
```

... oba naslednja stavka pa bosta povzročila napako pri prevajanju:

```
Number stevilo = vs.vrni(0);
ve.dodaj(20);
```

Zakaj? Razlog je v tem, da bi lahko kazalec tipa `Vektor<? extends Number>` kazal na, recimo, objekt tipa `Vektor<Double>`, v tak vektor pa seveda ne bi mogli dodajati elementov tipa `Integer`. Podobno velja, da bi lahko kazalec tipa `Vektor<? super Number>` kazal na, recimo, objekt tipa `Vektor<Object>`, kazalca tipa `Object`, ki bi ga v tem primeru vrnila metoda `vrni`, pa ne bi mogli prirediti spremenljivki tipa `Number`.

V vektor, na katerega kaže kazalec tipa `Vektor<? extends R>`, lahko shranimo le element tipa `null`, lahko pa katerikoli njegov element pridobimo kot element (kazalec) tipa `R` ali nadtipa tipa `R`.<sup>2</sup> V vektor, na katerega kaže kazalec tipa `Vektor<? super R>`, pa lahko shranimo poljuben element tipa `R` ali podtipa tipa `R`, njegove elemente pa lahko pridobivamo le kot elemente (kazalce) tipa `Object`. Opisano pravilo imenujemo *načelo pridobivanja in shranjevanja* (angl. get-and-put principle).

#### Naloga 8.16 Napišite metodo

```
public static void premakni(
    Vektor<? extends Number> izvor, Vektor<? super Number> cilj)
```

ki elemente vektorja `izvor` premakne v vektor `cilj`. Metodo preizkusite z argumenti različnih tipov. Z uporabo načela pridobivanja in shranjevanja pojasnite uporabo besed `extends` in `super` pri tipih parametrov.

**Naloga 8.17** Pri klicu metode iz prejšnje naloge je lahko drugi argument tipa `Vektor<Number>` ali `Vektor<Object>`, ne more pa biti tipa `Vektor<Integer>`. Metodo predelajte tako, da bo prvi argument lahko tipa `Vektor<P>`, drugi pa tipa `Vektor<Q>`, pri čemer je vsak od tipov `P` in `Q` lahko bodisi tip `Number` bodisi podtip tipa `Number`, tip `Q` pa je lahko bodisi nadtip tipa `P` ali pa enak tipu `P`.

## 8.9 Povzetek

- Generični razred je razred, ki je parametriziran z enim ali več tipnimi parametri. Pri uporabi takega razreda moramo vsakokrat navesti tipne argumente —

<sup>2</sup>Ker lahko kazalec z vrednostjo `null` priredimo katerikoli spremenljivki referenčnega tipa, si lahko predstavljamo, kot da vrednost `null` pripada posebnemu tipu, ki je podtip vseh referenčnih tipov.

konkretne tipe, s katerimi se bodo nadomestili pripadajoči tipni parametri.

- Generična metoda je metoda, ki je parametrizirana z enim ali več tipnimi parametri. Pri uporabi takih metod je le redko treba navesti tipne argumente, saj jih prevajalnik praviloma izlušči sam.
- Tipni parametri so lahko neomejeni, lahko pa so navzgor omejeni s podanim konkretnim tipom.
- Tipni parametri so vidni le prevajalniku, izvajalniku pa ne, saj jih prevajalnik v procesu brisanja tipov zamenja s konkretnimi tipi. Neomejen tipni parameter se nadomesti s tipom `Object`, tipni parameter, ki je navzgor omejen s tipom `R`, pa s tipom `R`. Tipni argumenti se v procesu brisanja tipov preprosto odstranijo: tip `Vektor<Integer>`, denimo, se nadomesti s tipom `Vektor`. Prevajalnik doda še ustrezne izrecne pretvorbe tipov.
- Pri generikih moramo upoštevati določene omejitve. Če je `T` tipni parameter, potem objekta tipa `T` ni mogoče ustvariti. Tabela tipa `T[]` sicer lahko izdelamo, vendar pa smo prisiljeni v »obvoz«.
- Generiki nam omogočajo, da podamo tip elementov, ki jih bo vsebovalnik hranil, ne da bi se nam bilo treba pri dostopu do elementov ukvarjati z izrecnimi pretvorbami tipov.
- Generiki v nasprotju s tabelami niso kovariantni, kljub temu pa lahko določeno mero splošnosti pri priredljivosti tipov dosežemo z uporabo nadomestnega znaka.

### ***Iz profesorjevega kabineta***

»Tle hipiji oziroma hiperji, al' kako se jim že reče ...«

»Hipsterji, profesor Doberšek ...«

»Že spet me prekinjaš! Te niso učili, da se starejšega človeka že tako ali tako ne sme prekinjati?! Jaz pa ti za povrh še rečem, kar priznaj, precej debel kruh!«

»O-o-proстите, gospod profesor, saj veste, nisem mislil ... Kruh pa res ni slab, čisto prav imate!«

»Že dobro. Skratka, tile hipsarji tlačijo rekurzijo že v vsak drek. Danes mi je nek študent pod nos pomolil nekaj takega:«

```
public abstract class Lik<T extends Lik<T>>
```

»Kaj za vraga naj bi tole pomenilo?«

»Razred `Lik`, ki je odvisen od tipa `T`, ki razširja razred `Lik`, ki je odvisen od tipa `T`, ki razširja razred `Lik`, ki je odvisen od tipa `T`, ki razširja razred `Lik`, ki je odvisen od ...«

»Genovefa! Prinesi mi vedro vode, hitro, prosim! Temule mojemu Jožetu se je zmešalo. Tako fejst fant je, ne bi ga rad izgubil ...«

»Bo že v redu, profesor, ujel se je v neskončno rekurzijo, umirjeno pristopi docentka Javornik. Do kosila se bo že zbrihtal, saj ni stroj ... Aha, kaj pa imate tu? Rekurzivni tipni parametri, zanimivo ... Zadeva je res čudaška, ni pa neuporabna.«

»Na primer?«

»Recimo, da bi v razred `Lik` želeli dodati metodo `primerjaj`, ki po ploščini primerja lik `this` in `lik`, ki ga podamo kot argument:

```
public abstract class Lik {
    ...
    public int primerjaj(Lik drugi) {
        return Double.compare(this.ploscina(), drugi.ploscina());
    }
    ...
}
```

Ta metoda nam omogoča, da med seboj primerjamo pravokotnike in kroge. Včasih si tega res želimo, če si ne, pa ... Dobro, obstaja metoda `getClass`, s katero lahko rešimo problem na nivoju izvajalnika. Če pa želimo, da se bo v primeru neskladja tipov pritožil že prevajalnik, ne šele izvajalnik, moramo poseči po rekurzivnih tipnih parametrih. Skratka, če razred `Lik` in metodo `primerjaj` definiramo takole ...

```
public abstract class Lik<T extends Lik<T>> {
    ...
    public int primerjaj(T drugi) {
        return Double.compare(this.ploscina(), drugi.ploscina());
    }
}
```

... nam prevajalnik prepreči, da bi med seboj primerjali like različnih tipov. Razreda `Pravokotnik` in `Krog` moramo v tem primeru deklarirati takole:«

```
private static class Pravokotnik extends Lik<Pravokotnik>
private static class Krog extends Lik<Krog>
```

Dokončajte docentkin primer, nato pa ga preizkusite in poskusite doumeti v popolnosti!



## 9 Vmesniki in notranji razredi

Vmesnik si lahko predstavljamo kot okrnjen abstrakten razred, ki v osnovi sestoji zgolj iz abstraktnih metod, čeprav lahko vsebuje tudi nekatere druge elemente. Razred *implementira* vmesnik, če podaja definicije njegovih abstraktnih metod. Medtem ko je vsak razred lahko podrazred kvečjemu enega razreda, lahko implementira poljubno mnogo vmesnikov. Objektov vmesnika seveda ne moremo neposredno izdelovati, lahko pa deklariramo spremenljivko pripadajočega tipa.

V javini standardni knjižnici so definirani številni vmesniki. V tem poglavju si bomo ogledali vmesnike *Comparable*, *Comparator*, *Iterator* in *Iterable*. Prvo dvojico uporabimo, ko želimo definirati urejenost objektov določenega razreda, druga pa nam pride prav pri razredih, pri katerih so objekti sestavljeni iz poljubnega števila elementov (npr. razred *Vektor*). Z vmesnikoma *Iterator* in *Iterable* lahko določimo, kako se po elementih takih objektov »sprehajamo«.

### 9.1 Opredelitev in uporaba

V nasprotju z nekaterimi drugimi jeziki, kot sta npr. C++ in python, je lahko vsak javanski razred neposredni podrazred kvečjemu enega razreda. Ta poenostavitev je namerna, saj *večkratno dedovanje* (kot pravimo situaciji, ko je nek razred neposredni podrazred več razredov) prinaša s seboj določene izzive. Na primer, če je razred C neposredni podrazred razredov A in B in če A in B oba redefinirata metodo *equals* (vsak na svoj način), razred C pa tega ne stori, katera različica metode *equals* naj se potem pokliče nad objektom razreda C? Kljub temu pa večkratno dedovanje včasih pride prav. Zato so se načrtovalci jave odločili, da bodo večkratno dedovanje omogočili samo pri okrnjenih abstraktnih razredih, ki lahko vsebujejo zgolj abstraktne metode. Takšni »razredi« se imenujejo *vmesniki*.

Razred *implementira* vmesnik, če implementira vse njegove abstraktne metode. Če katere od njih ne implementira, mora biti deklariran kot abstrakten. Ta logika je povsem enaka kot pri abstraktnih razredih, razlika je le v tem, da lahko razred implementira poljubno mnogo vmesnikov.

Vmesnik lahko *razširja* poljubno mnogo drugih vmesnikov. To pomeni, da poleg svojih metod vključuje še metode vseh vmesnikov, ki jih razširja. Če vmesnik B razširja vmesnik A, je vmesnik B *podvmesnik* vmesnika A, vmesnik A pa *nadvme-*

*snik* vmesnika *B*. Tako kot relacija nadrazred-podrazred je tudi relacija nadvmesnik-podvmesnik tranzitivna. Ker je vmesnik (tako kot razred) definicija tipa, lahko namesto o nadvmesnikih in podvmesnikih govorimo o nadtipih in podtipih. Pojem nadtipa oz. podtipa lahko razširimo na implementacijo vmesnika: če razred *B* implementira vmesnik *A*, lahko rečemo, da je tip *B* podtip tipa *A*.

V začetnih različicah jave so lahko vmesniki poleg abstraktnih metod vsebovali le še statične konstante. Danes lahko vsebujejo tudi statične metode in t.i. privzete metode (te so dejansko polnopravne metode). Z uvedbo privzetih metod so načrtovalci jave želeli zagotoviti možnost obogatitve vmesnika z dodatnimi metodami, ne da bi zato bilo treba dopolniti vse implementacijske razrede, vendar pa so se tako soočili s težavami, na las podobnimi tistim, ki so se jim želeli izogniti z okrnitvijo večkratnega dedovanja.

Vmesnik definiramo na enak način kot razred, le da namesto `class` pišemo `interface`. Vsi elementi vmesnika so javno dostopni (določili `private` in `protected` nista mogoči), vse metode, ki niso niti statične niti privzete, pa so abstraktne. Določili `public` in `abstract` lahko dejansko kar izpustimo, a zavoljo večje jasnosti tega ne bomo počeli. Implementacijo vmesnika najavimo z besedo `implements`, njegovo razširitev v nekem drugem vmesniku pa z že znano besedo `extends`.

Oglejmo si primer:<sup>1</sup>

```
// koda 9.1 (vmesniki/narisljiv/*.java)
public interface Narisljiv {
    public abstract void narisi(Platno platno);
}

public interface Izracunljiv {
    public abstract double ploscina();
    public abstract double obseg();
}

public class Pravokotnik implements Narisljiv, Izracunljiv {
    ...
    @Override
    public void narisi(Platno platno) {
        ...
    }

    @Override
    public double ploscina() {
        ...
    }
}
```

<sup>1</sup>Imena vmesnikov se pogosto končajo na *-ljiv* v slovenščini oziroma *-able* v angleščini, saj dosti-krat opredeljujejo lastnost, ki jo imajo objekti razredov, ki implementirajo vmesnik.

```

    }

    @Override
    public double obseg() {
        ...
    }
}

```

V razredu Pravokotnik moramo implementirati vse abstraktne metode vmesnikov Narisljiv in Izracunljiv. Če tega ne storimo, moramo razred Pravokotnik proglasiti za abstrakten.

**Naloga 9.1** Preučite kodo v imeniku vmesniki/narisljiv in jo dopolnite z razredom Daljica. Daljica je »narisljiva«, ni pa »izračunljiva«, saj kot enodimenziionalni objekt nima ploščine in obsega.

**Naloga 9.2** V razred Glavni v imeniku vmesniki/narisljiv dodajte sledeči metodi:

- `public static void narisi(Vektor<Narisljiv> v, Platno platno)`  
Na podano platno nariše vse objekte iz podanega vektorja.
- `public static Izracunljiv najvecji(Vektor<Izracunljiv> v)`  
Vrne tisti objekt iz podanega vektorja, ki ima največjo ploščino.

**Naloga 9.3** Sledeči vmesnik predstavlja funkcijo enega parametra:

```

public interface Funkcija<V, I> {
    public abstract I f(V vhod);
}

```

Napišite metodo

```

public static <V, I> Slovar<I, Vektor<V>> grupiraj(
    Funkcija<V, I> funkcija, Vektor<V> vhodi)

```

ki na vsakem elementu podanega vektorja vhodov uporabi podano funkcijo in vrne slovar, ki vsakega od dobljenih rezultatov funkcije preslika v vektor elementov, ki so dali ta rezultat. Na primer, pri funkciji, ki vrne kvadrat svojega argumenta, in pri vektorju vhodov  $\langle -2, -1, 0, 1, 2 \rangle$  bi se rezultat 4 preslikal v vektor  $\langle -2, 2 \rangle$ , rezultat 1 v vektor  $\langle -1, 1 \rangle$ , rezultat 0 pa v vektor  $\langle 0 \rangle$ .

**Naloga 9.4** Objekta vmesnika ni mogoče neposredno izdelati, lahko pa izdelamo razred, ki implementira vmesnik, nato pa ustvarimo njegov objekt. Napišite metodo

```
public static <V, I, J> Funkcija<V, J> kompozitum(
    Funkcija<V, I> f1, Funkcija<I, J> f2)
```

ki vrne funkcijo, ki na svojem vhodu najprej uporabi funkcijo *f1*, na dobljenem rezultatu pa še funkcijo *f2*.

## 9.2 Vmesnik Comparable in naravna urejenost

Vmesnik Comparable ima eno samo abstraktno metodo:

```
public interface Comparable<T> {
    public abstract int compareTo(T drugi);
}
```

Če razred *R* implementira vmesnik Comparable<*R*>, potem je njegove objekte mogoče med seboj *primerjati* (angl. *comparable* = primerljiv). To pomeni, da lahko za vsak par objektov *p* in *q* razreda *R* povemo, ali *p* sodi pred *q*, *q* sodi pred *p* ali pa sta *p* in *q* po primerjalnem kriteriju enakovredna. Objekte je pogosto mogoče primerjati po različnih kriterijih; nize, denimo, bi lahko primerjali leksikografsko (Andrej < Barbara < Cene), inverzno leksikografsko (Cene < Barbara < Andrej) ali pa, zakaj pa ne, po dolžini (Cene < Andrej < Barbara). Nenapisano pravilo je, da z metodo *compareTo* opredelimo najbolj običajno urejenost, ki ji pravimo *naravna urejenost*. Pri nizih je to gotovo leksikografska urejenost, cela števila naravno primerjamo glede na matematično relacijo je-manjši-od, objekte tipa *Cas* primerjamo kronološko, objekte tipa *Oseba* primerjamo leksikografsko po priimkih, v primeru enakih priimkov pa po imenih itd.

Metodo *compareTo* implementiramo tako, da vrne

- negativno število, če objekt *this* po kriteriju naravne urejenosti sodi pred objekt *drugi*;
- pozitivno število, če objekt *this* po kriteriju naravne urejenosti sodi za objekt *drugi*;
- število 0, če sta objekta *this* in *drugi* po kriteriju naravne urejenosti enakovredna.

V razredih *Cas* in *Oseba* bi razred Comparable implementirali na sledeči način:

```
// koda 9.2 (vmesniki/cas/Cas.java)
public class Cas implements Comparable<Cas> {
    ...
    @Override
    public int compareTo(Cas drugi) {
```



```

        return 60 * (this.ura - drugi.ura) +
            (this.minuta - drugi.minuta);
    }
}

```

```

// koda 9.3 (vmesniki/oseba/Oseba.java)
public class Oseba implements Comparable<Oseba> {
    private String ime;
    private String priimek;
    private char spol;
    private int letoRojstva;

    public Oseba(String ime, String priimek, char spol, int lr) {
        this.ime = ime;
        this.priimek = priimek;
        this.spol = spol;
        this.letoRojstva = lr;
    }

    @Override
    public String toString() {
        return String.format("%s %s (%c), %d",
            this.ime, this.priimek, this.spol, this.letoRojstva);
    }

    @Override
    public int compareTo(Oseba druga) {
        if (this.priimek.equals(druga.priimek)) {
            return this.ime.compareTo(druga.ime);
        }
        return this.priimek.compareTo(druga.priimek);
    }
}

```

Bodimo pozorni na tipne argumente: razred `Cas` implementira vmesnik `Comparable<Cas>`, metoda `compareTo` pa zato sprejme parameter tipa `Cas`. Podobno velja za razred `Oseba`.

Tudi razred `String` implementira vmesnik `Comparable` (s tipnim argumentom `String`, seveda) in s tem metodo `compareTo`. Ta metoda določa leksikografsko urejenost, saj vrne nekaj negativnega, če je niz `this` leksikografsko pred drugim nizom, nekaj pozitivnega, če je niz `this` leksikografsko za drugim nizom, in 0, če sta niza enaka. V kodi 9.3 smo metodo uporabili za primerjanje imen in priimkov oseb.

Vmesnik Comparable nam pride prav pri urejanju tabele ali vektorja objektov. Sledeča generična metoda s pomočjo algoritma navadnega vstavljanja naravno uredi podani vektor. Določilo `T extends Comparable<T>` zahteva, da elementi podanega vektorja pripadajo podtipu tipa `Comparable<T>` (torej razredu, ki implementira vmesnik `Comparable<T>`).

```
// koda 9.4 (vmesniki/primerjalniki/Urejanje.java)
public static <T extends Comparable<T>> void uredi(Vektor<T> vektor){
    int stElementov = vektor.steviloElementov();

    for (int i = 1; i < stElementov; i++) {
        T element = vektor.vrni(i);
        int j = i - 1;

        while (j >= 0 && vektor.vrni(j).compareTo(element) > 0) {
            vektor.nastavi(j + 1, vektor.vrni(j));
            j--;
        }
        vektor.nastavi(j + 1, element);
    }
}
```

Metoda deluje na enak način kot metoda `uredi` v razdelku 5.8 (koda 5.13), razlike so le v podrobnostih. Do elementov vektorja namesto z oglatimi oklepaji dostopamo z metodama `vrni` in `nastavi`, izraz

```
t[j] > el
```

v glavi zanke `while` pa smo nadomestili z izrazom

```
vektor.vrni(j).compareTo(element) > 0
```

ki element vektorja na indeksu `j` in element `element` primerja po naravni urejenosti, določeni z metodo `compareTo` v razredu `T`.

Kljub razmeroma majhnim razlikam je gornja metoda uredi precej splošnejša od njene predhodnice iz razdelka 5.8. Prvič, elementi, ki jih urejamo, lahko pripadajo poljubnemu razredu, ki implementira vmesnik `Comparable` (torej `Integer`, `String` itd.), pri prvotni različici pa smo bili omejeni zgolj na tip `int`. Drugič, operacija primerjanja je bila pri prvotni različici fiksna, tukaj pa je odvisna od implementacije vmesnika `Comparable`.

Metodo `uredi` lahko preizkusimo, recimo, takole:

```
public static void main(String[] args) {
    Vektor<Oseba> osebe = new Vektor<>();
}
```

```

osebe.dodaj(new Oseba("Milena", "Novak", 'Z', 1957));
osebe.dodaj(new Oseba("Jure", "Dolenc", 'M', 1972));
osebe.dodaj(new Oseba("Jure", "Novak", 'M', 1978));
osebe.dodaj(new Oseba("Maja", "Vovk", 'Z', 2001));
osebe.dodaj(new Oseba("Marko", "Dolenc", 'M', 1966));
osebe.dodaj(new Oseba("Eva", "Bregar", 'Z', 2007));
osebe.dodaj(new Oseba("Ana", "Mihevc", 'Z', 1998));
osebe.dodaj(new Oseba("Mojca", "Furlan", 'Z', 1986));
osebe.dodaj(new Oseba("Peter", "Bregar", 'M', 1975));
osebe.dodaj(new Oseba("Ivan", "Vovk", 'M', 1947));
System.out.println(osebe);

uredi(osebe);
System.out.println(osebe);
}

```

**Naloga 9.5** Dopolnite razred `Ulopek` iz poglavja 6 tako, da bo implementiral vmesnik `Comparable<Ulopek>`.

**Naloga 9.6** Objekt sledečega razreda hrani število tipa `int`, poleg tega pa ga lahko primerjamo z drugim takim objektom:

```

public class Stevilo implements Comparable<Stevilo> {
    private int n;

    public Stevilo(int n) {
        this.n = n;
    }

    @Override
    public int compareTo(Stevilo drugo) {
        return this.n - drugo.n;
    }
}

```

Zakaj bi bilo boljše, da bi metodo `compareTo` napisali takole?

```

@Override
public int compareTo(Stevilo drugo) {
    return Integer.compare(this.n, drugo.n);
}

```

### 9.3 Vmesnik Comparator

Vmesnik Comparator je na prvi pogled zelo podoben vmesniku Comparable, saj prav tako vsebuje eno samo abstraktno metodo, ta pa ima podobno glavo kot metoda compareTo:

```
public interface Comparator<T> {
    public abstract int compare(T prvi, T drugi);
}
```

V nasprotju z metodo compareTo iz vmesnika Comparable sprejme metoda compare poleg obveznega kazalca this še dva parametra. Metodo implementiramo tako, da vrne poljubno negativno vrednost, če objekt prvi sodi pred objekt drugi, 0, če sta objekta glede na primerjalni kriterij enakovredna, in poljubno pozitivno vrednost, če objekt drugi sodi pred objekt prvi.

Medtem ko vmesnik Comparable implementiramo neposredno z razredom, ki mu želimo določiti naravno urejenost, vmesnik Comparator implementiramo z ločenim razredom. V njem definiramo eno od (v splošnem več možnih) *alternativnih urejenosti*.

Oglejmo si primer. Razredu Oseba smo naravno urejenost že določili, sedaj pa bomo definirali še dve alternativni urejenosti: prva bo osebe primerjala samo po priimku, druga pa primarno po spolu (najprej dame, potem gospodje), sekundarno pa po padajoči starosti. V obeh primerih bomo napisali metodo, ki vrne objekt tipa Comparator. Ta objekt pripada statičnemu notranjemu razredu, ki implementira vmesnik Comparator.

```
// koda 9.5 (vmesniki/primerjalniki/Oseba.java)
public class Oseba {
    ...
    private static class PrimerjalnikPoPriimku
        implements Comparator<Oseba> {
        @Override
        public int compare(Oseba prva, Oseba druga) {
            return prva.priimek.compareTo(druga.priimek);
        }
    }

    private static class PrimerjalnikPoSpoluInStarosti
        implements Comparator<Oseba> {
        @Override
        public int compare(Oseba prva, Oseba druga) {
            if (prva.spol != druga.spol) {
                return druga.spol - prva.spol;
            }
        }
    }
}
```



```

    int stElementov = vektor.steviloElementov();

    for (int i = 1; i < stElementov; i++) {
        T element = vektor.vrni(i);
        int j = i - 1;

        while (j >= 0 && primerjalnik.compare(
            vektor.vrni(j), element) > 0) {    // (1)
            vektor.nastavi(j + 1, vektor.vrni(j));
            j--;
        }
        vektor.nastavi(j + 1, element);
    }
}

```

Ta metoda uredi podani vektor po kriteriju, ki ga določa podani primerjalnik, sicer pa se od metode v kodi 9.4 razlikuje le v glavi in v vrstici (1), kjer smo klic  $e_1.compareTo(e_2)$  nadomestili s klicem `primerjalnik.compare( $e_1$ ,  $e_2$ )`. Metodo lahko preizkusimo takole:

```

public static void main(String[] args) {
    Vektor<Oseba> osebe = new Vektor<>();
    // v vektor dodaj objekte tipa Oseba
    // ...

    uredi(osebe, Oseba.primerjalnikPoPriimku());
    System.out.println(osebe);

    uredi(osebe, Oseba.primerjalnikPoSpoluInStarosti());
    System.out.println(osebe);
}

```

### Naloga 9.7 Razred Ulomek dopolnite z metodo

```
public Comparator<Ulomek> poVsoti()
```

ki vrne primerjalnik, ki podana (okrajšana) ulomka primarno primerja po vsoti števca in imenovalca, sekundarno pa po števcu. Na primer, ulomek  $5/2$  sodi pred ulomek  $1/7$ , ta pa pred ulomek  $3/5$ .

### Naloga 9.8 Napišite metodo

```

public static <T extends Comparable<T>>
    Comparator<Vektor<T>> poElementu(int indeks)

```

ki vrne primerjalnik, ki pri podanih vektorjih primerja elementa na indeksu indeks.

Nikar se ne ustrašite! Primerjalnik tipa `Comparator<Vektor<T>>` med seboj primerja vektorje z elementi tipa `T`, sam tip `T` pa mora implementirati vmesnik `Comparable<T>`, saj drugače ne boste mogli med seboj primerjati elementov podanih vektorjev (neomejeni tip `T` ne ponuja metode `compareTo`, saj je razred `Object` nima).

#### Naloga 9.9 Napišite metodo

```
public static <T> Comparator<T> obrat(Comparator<T> primerjalnik)
```

ki vrne primerjalnik, ki deluje ravno obratno od podanega primerjalnika: če objekt *a* po podanem primerjalniku sodi pred objekt *b*, naj po vrnjenem primerjalniku sodi za njega (in obratno). Če sta objekta po podanem primerjalniku enakovredna, pa naj ju tako obravnava tudi vrnjeni primerjalnik.

#### Naloga 9.10 Napišite metodo

```
public static <T> Comparator<T> stik(
    Comparator<T> prim1, Comparator<T> prim2)
```

ki vrne primerjalnik, ki podana objekta primerja s primerjalnikom `prim1`, če sta po tem kriteriju enakovredna, pa ju primerja še s primerjalnikom `prim2`. S to metodo lahko, denimo, na podlagi primerjalnika, ki osebi primerja po spolu, in primerjalnika, ki osebi primerja po starosti, pridobimo primerjalnik, ki osebi primerja po spolu, osebe istega spola pa po starosti.

## 9.4 Vmesnik Iterator

Z implementacijo vmesnika `Iterator` povemo, kako se po objektu določenega razreda sprehodimo. Sprehod praviloma definiramo za *vsebovalnike*, objekte, namenjene hranjenju elementov. Tipičen primer vsebovalnika je vektor.

Recimo, da razred, ki definira vsebovalnik nad elementi tipa `T` (npr. `Vektor<T>`), vsebuje metodo

```
Iterator<T> iterator()
```

ki vrne iterator po objektu `this`. Iterator uporabimo takole (*R* predstavlja tipni argument):

```
Iterator<R> it = vsebovalnik.iterator();
while (it.hasNext()) {
    R element = it.next();
```

```

    // obdelaj element
    // ...
}

```

Kot vidimo, nad iteratorjem izmenično kličemo metodi `hasNext` in `next` in se tako sprehodimo po vsebovalniku. Metoda `hasNext` vrne `true`, če obstaja še kak element vsebovalnika, ki ga iterator ni obiskal, metoda `next` pa vrne enega od še ne obiskanih elementov. Pri vsebovalnikih, v katerih so elementi nanizani v določenem vrstnem redu, iterator elemente običajno obišče po vrsti od prvega do zadnjega; prvi klic metode `next` vrne prvi element, drugi klic vrne drugi element itd. V splošnem pa lahko iterator obiskuje elemente v poljubnem vrstnem redu.

Vmesnik `Iterator` torej ponuja dve abstraktni metodi:

```

public interface Iterator<T> {
    public abstract boolean hasNext();
    public abstract T next();
}

```

#### 9.4.1 Iterator nad vektorjem

Napišimo iterator, ki se po vrsti sprehodi po elementih vektorja. Zdaj že vemo, da se mora iterator odzivati na klice metod `hasNext` in `next`. Metoda `next` bo ob prvem klicu vrnila prvi element vektorja, ob drugem drugi element itd., metoda `hasNext` pa bo vse do konca sprehoda vračala `true`, na koncu pa bo vrnila `false`. Da bomo lahko metodi pravilno sprogramirali, si moramo zapomniti, na katerem elementu se iterator trenutno »nahaja« (kateri element bo vrnil naslednji klic metode `next`). V ta namen bomo vzdrževali spremenljivko indeks, ki jo bomo na začetku postavili na 0 (torej na indeks prvega elementa vektorja), ob vsakem klicu metode `next` pa jo bomo povečali za 1.

Kot vemo, objekta vmesnika ni mogoče ustvariti, zato bomo iteratorje izdelovali kot objekte razreda, ki implementira vmesnik `Iterator`. Ta razred bomo imenovali `IteratorCezVektor`. Razred bi lahko izdelali kot samostojni razred, a ker z vidika uporabnika iteratorjev ne ponuja ničesar, česar ne bi ponujal že vmesnik `Iterator`, ga nima smisla izpostavljati. Definirali ga bomo kot privatni statični notranji razred v razredu `Vektor`.

Iterator po vektorju bomo izdelali z metodo `iterator` v razredu `Vektor`:

```

// koda 9.7 (vmesniki/iteratorji/Vektor.java)
public class Vektor<T> {
    ...
    public Iterator<T> iterator() {
        return new IteratorCezVektor<T>(this);
    }
}

```



```
    }
}
```

Zakaj smo ob izdelavi objekta razreda `IteratorCezVektor` podali tipni parameter `T` in kazalec `this`? Tip `T` posredujemo zato, ker se statični notranji razred obnaša podobno kot samostojni razred in tipa `T` ne prevzame od oklepajočega razreda. Kazalec `this` pa podamo zato, ker bo objekt razreda `IteratorCezVektor` moral vedeti, po katerem objektu (konkretnem vektorju) se bo sprehodil. To bo tisti objekt, nad katerim kličemo metodo `iterator`. Na primer, če metodo pokličemo takole ...

```
// koda 9.8 (programi/vmesniki/iteratorji/TestVektor.java)
```

```
Vektor<Integer> stevila = new Vektor<>();
stevila.dodaj(10);
stevila.dodaj(20);
stevila.dodaj(30);
Iterator<Integer> it = stevila.iterator();
```

... potem bo objekt razreda `IteratorCezVektor` prejel kazalec na vektor `stevila`. Ta kazalec pa je v metodi `iterator` viden kot `this`.

V razredu `IteratorCezVektor` bomo poleg obveznih metod `hasNext` in `next` definirali tudi dva atributa in konstruktor. Atribut vektor kaže na vektor, po katerem se bo iterator `this` sprehodil, atribut indeks pa, kot smo že povedali, hrani indeks elementa, ki ga bo vrnil naslednji klic metode `next`. V metodi `hasNext` preverimo, ali je ta indeks manjši od števila elementov vektorja. To posredno storimo tudi v metodi `next`; če smo sprehod že zaključili, moramo namreč v skladu s protokolom, navedenim v dokumentaciji vmesnika `Iterator`, sprožiti izjemo tipa `NoSuchElementException`. Če še nismo obiskali vseh elementov, pa v metodi `next` vrnemo element na trenutnem indeksu, obenem pa indeks povečamo za 1 in se tako pripravimo na naslednji krog klicev metod `hasNext` in `next`.

```
public class Vektor<T> {
    ...
    private static class IteratorCezVektor<T> implements Iterator<T>{
        private Vektor<T> vektor;
        private int indeks;

        public IteratorCezVektor(Vektor<T> vektor) {
            this.vektor = vektor;
            this.indeks = 0;
        }

        @Override
        public boolean hasNext() {
```

```

        return this.indeks < this.vektor.steviloElementov();
    }

    @Override
    public T next() {
        if (!this.hasNext()) {
            throw new NoSuchElementException();
        }
        return this.vektor.vrni(this.indeks++);
    }
}

```

Nadaljujmo kodo 9.8 tako, da s pomočjo iteratorja izpišemo vse elemente vektorja. Spotoma si bomo pomagali z javino samodejno pretvorbo tipa `Integer` v tip `int`:

```

// koda 9.9
while (it.hasNext()) {
    int element = it.next();
    System.out.println(element);
}

```

**Naloga 9.11** Razred `Vektor<T>` dopolnite z metodo

```
public Iterator<T> skakalec(int zacetek, int korak)
```

ki vrne iterator, ki prične na elementu z indeksom `zacetek`, vsak klic metode `next` pa ga prestavi za korak elementov naprej.

**Naloga 9.12** Napišite metodo

```
public static <T> Iterator<T> vzratniIterator(Vektor<T> vektor)
```

ki vrne iterator, ki svoj sprehod prične na zadnjem elementu podanega vektorja, nato pa se premika vzvratno proti prvemu elementu, kjer se njegova pot zaključi.

**Naloga 9.13** Napišite metodo

```
public static <T> Iterator<T> poTabeli(T[] tabela)
```

ki vrne iterator, ki se sprehodi po podani tabeli.

### 9.4.2 Iterator nad slovarjem

Iterator nad slovarjem je nekoliko težji zalogaj, saj se moramo sprehoditi po zgoščeni tabeli, ki je dejansko dvodimenzionalna struktura. Najprej se sprehodimo po prvem povezanem seznamu, nato po drugem itd. Povezani seznam, po katerem se trenutno sprehajamo, bomo imenovali *trenutni povezani seznam*.

Za izvedbo takega sprehoda moramo vzdrževati vsaj dva podatka:

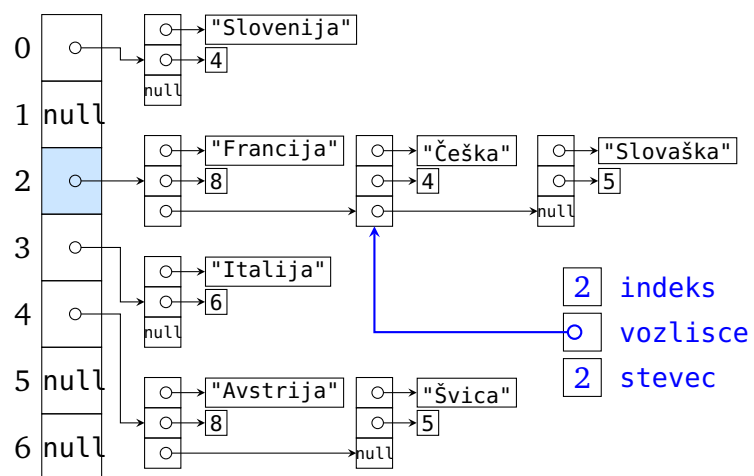
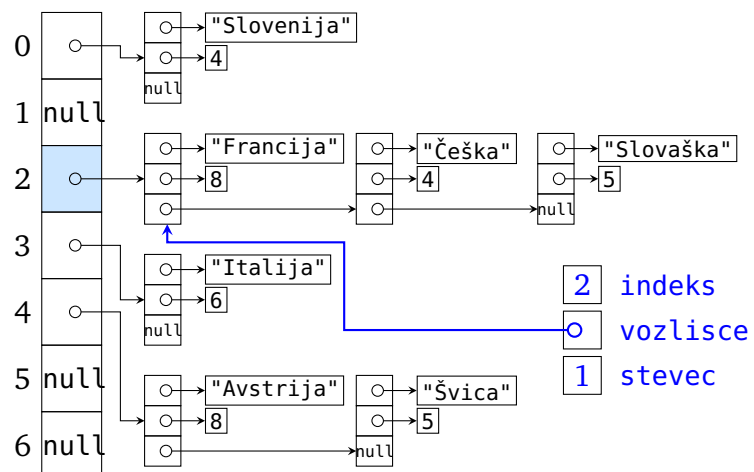
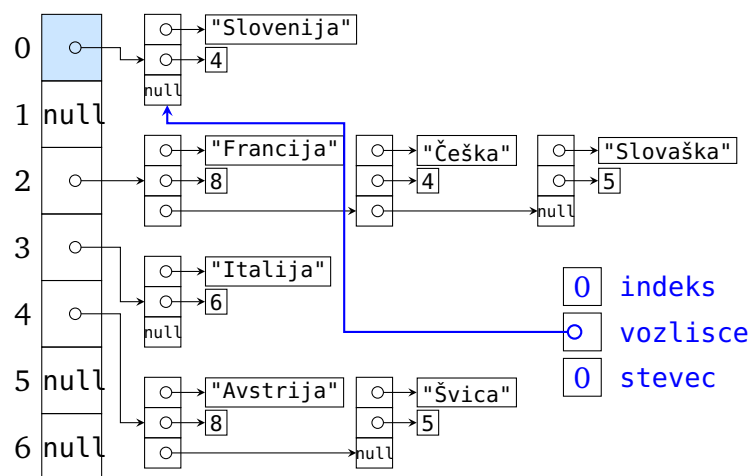
- indeks trenutnega povezanega seznama v tabeli (atribut `indeks`);
- kazalec na trenutno vozlišče v trenutnem povezanem seznamu (atribut `vozlisce`).

Dodali bomo še atribut `stevec`, ki bo hranil skupno število že obiskanih vozlišč. Lahko bi shajali tudi brez njega, vendar pa nam bo precej poenostavil metodo `hasNext`. Ker bomo vmesnik `Iterator` implementirali s statičnim notranjim razredom, bomo hranili tudi kazalec na slovar, po katerem se bomo sprehodili (atribut `slovar`).

Slika 9.1 prikazuje prve tri korake sprehoda po zgoščeni tabeli s slike 7.5. Na začetku je atribut `indeks` enak `-1`, atribut `stevec` ima vrednost `0`, atribut `vozlisce` pa `null`. V prvem koraku sprehoda (tj. v prvem klicu metode `next`) bomo poiskali prvi neprazen povezani seznam (v našem primeru je to povezani seznam, ki vsebuje ključ `Slovenija`). V vseh ostalih korakih pa bomo najprej preverili, ali smo že prišli do konca trenutnega povezanega seznama. Če smo, moramo poiskati naslednji neprazen povezani seznam, v nasprotnem primeru pa se zgolj prestavimo na naslednje vozlišče trenutnega povezanega seznama. V prvem primeru se bosta posodobila atributa `indeks` in `vozlisce`, v drugem pa samo `vozlisce`. Atribut `stevec` v obeh primerih povečamo za `1`.

Iterator se bo dejansko sprehajal samo po ključih, ne po celotnih vozliščih, saj na podlagi ključev zlahka pridobimo pripadajoče vrednosti. Obstaja še tehtnejši razlog za takšno odločitev: vozlišča so implementacijska podrobnost, ki uporabnika slovarja pravzaprav ne zanima. Z vidika uporabnika obstajajo le ključi in vrednosti, kako jih slovar hrani, pa ni pomembno. Za slovar s ključi tipa `K` in vrednostmi tipa `V` bo torej metoda `next` vračala vrednosti tipa `K`, ne `Vozlisce<K, V>` ali kaj drugega. Statični notranji razred za definicijo iteratorja (imenovali ga bomo `IteratorPoKljucih`) bo zato imel takšno glavo:

```
// koda 9.10 (vmesniki/iteratorji/Slovar.java)
public class Slovar<K, V> {
    ...
    private static class IteratorPoKljucih<K, V>
        implements Iterator<K> {
    }
}
```



Slika 9.1 Prvi trije koraki sprehoda iteratorja po zgoščeni tabeli.

Razred `IteratorPoKljucih` bo uporabljal tako tip `K` kot tip `V`, saj je atribut `slovar` tipa `Slovar<K, V>`, atribut `vozlisce` pa tipa `Vozlisce<K, V>`. Ker bo metoda `next` vračala vrednosti tipa `K`, naš razred implementira vmesnik `Iterator<K>`.

Atribute smo že našteali, konstruktor pa jih zgolj nastavi na začetne vrednosti:

```
private static class IteratorPoKljucih<K, V> implements Iterator<K> {
    private Slovar<K, V> slovar;
    private int indeks;
    private int stevec;
    private Vozlisce<K, V> vozlisce;

    public IteratorPoKljucih(Slovar<K, V> slovar) {
        this.slovar = slovar;
        this.indeks = -1;
        this.stevec = 0;
        this.vozlisce = null;
    }
}
```

Po zaslugi atributa `stevec` zlahka sprogramiramo logiko za preverjanje, ali smo se že sprehodili po vseh vozliščih:

```
private static class IteratorPoKljucih<K, V> implements Iterator<K> {
    ...
    @Override
    public boolean hasNext() {
        return this.stevec < this.slovar.stParov;
    }
}
```

Atribut `stParov` hrani skupno število vozlišč (parov ključ-vrednost) v slovarju. Ta atribut moramo dodati v razred `Slovar`. Mala malica: v konstruktorju razreda `Slovar` ga nastavimo na 0, ob vsakem dodajanju novega vozlišča (torej v bloku `else` v metodi `shrani`) pa ga povečamo za 1.

```
public class Slovar<K, V> {
    ...
    private int stParov;
    ...
    @SuppressWarnings("unchecked")
    public Slovar(int velikostTabele) {
        ...
        this.stParov = 0;
    }
}
```

```

    }

    public void shrani(K kljuc, V vrednost) {
        ...
        if (vozlisce != null) {
            ...
        } else {
            ...
            this.stParov++;
        }
    }
}

```

V metodi `next` v razredu `IteratorPoKljucih` najprej preverimo, ali smo sprehod že zaključili. Če smo, moramo po pravilih sprožiti izjemo tipa `NoSuchElementException`. V nasprotnem primeru pa se premaknemo bodisi na prvo vozlišče naslednjega nepraznega povezanega seznama (ob prvem klicu metode `next` ali pa potem, ko smo prispeli do konca trenutnega povezanega seznama) bodisi na naslednje vozlišče trenutnega povezanega seznama. Nato le še povečamo atribut `stevec` in vrnemo ključ v vozlišču, na katerem smo pristali.

```

private static class IteratorPoKljucih<K, V> implements Iterator<K> {
    ...
    @Override
    public K next() {
        if (!this.hasNext()) {
            throw new NoSuchElementException();
        }

        if (this.indeks < 0 || this.vozlisce.naslednje == null) {
            // poišči naslednji neprazni povezani seznam
            do {
                this.indeks++;
            } while (this.indeks < this.slovar.podatki.length &&
                    this.slovar.podatki[this.indeks] == null);
            this.vozlisce = this.slovar.podatki[this.indeks];
        } else {
            this.vozlisce = this.vozlisce.naslednje;
        }
        this.stevec++;
        return this.vozlisce.kljuc;
    }
}

```

```
}
```

Razred Slovar bomo dopolnili še z metodo `iterator`, ki vrne iterator po slovarju `this`:

```
public class Slovar<K, V> {
    ...
    public Iterator<K> iterator() {
        return new IteratorPoKljucih<K, V>(this);
    }
}
```

Iterator preizkusimo na podoben način kot pri vektorju:

```
// koda 9.11 (vmesniki/iteratorji/TestSlovar.java)
Slovar<String, Integer> drzavaVSosed = new Slovar<>();
drzavaVSosed.shrani("Slovenija", 4);
drzavaVSosed.shrani("Avstrija", 8);
...
Iterator<String> it = drzavaVSosed.iterator();
while (it.hasNext()) {
    String drzava = it.next();
    int stSosed = drzavaVSosed.vrni(drzava);
    System.out.printf("%s -> %d%n", drzava, stSosed);
}
```

#### Naloga 9.14 Napišite metodo

```
public static <T> Iterator<T> poVV(Vektor<Vektor<T>> vv)
```

ki vrne iterator po elementih podanega vektorja vektorjev.

## 9.5 Vmesnik Iterable

Pridevnik *iterable* bi dobessedno lahko prevedli kot *sprehodljiv*, v nekoliko lepši slovenščini pa kot *tak, po katerem se je mogoče sprehoditi*. Vmesnik `Iterable` ima eno samo abstraktno metodo:

```
public interface Iterable<T> {
    public abstract Iterator<T> iterator();
}
```

Za razred *R*, ki implementira vmesnik `Iterable<T>`, torej vemo, da vsebuje metodo, ki vrne iterator — objekt, s pomočjo katerega se je mogoče sprehoditi po ele-

mentih objekta razreda *R*. Objekti takega razreda so namreč tipično sestavljeni iz več elementov (sicer iterator ne bi imel pravega smisla), ti pa so tipa *T*. Velja pa še več: če je obj objekt takega razreda *R*, se lahko po njegovih elementih sprehodimo z zanko *for-each*.

```
for (T element: obj) {
    // obdelaj element
}
```

Ta zanka je enakovredna sledečemu odseku kode:

```
Iterator<T> iterator = obj.iterator();
while (iterator.hasNext()) {
    T element = iterator.next();
    // obdelaj element
}
```

Zanko *for-each* torej zlahka nadomestimo s klici iteratorjevih metod, vendar pa je krajša in preglednejša, zato vmesnik *Iterable* implementiramo, kadarkoli je taka zanka smiselna.

Pri razredih *Vektor* in *Slovar* smo metodo *iterator* pravzaprav že napisali. Vse, kar nam preostane, je, da glavi razredov dopolnimo z ustreznim pristavkom *implements*, pred metodi *iterator* pa postavimo oznako *@Override*:

```
public class Vektor<T> implements Iterable<T> {
    ...
    @Override
    public Iterator<T> iterator() {
        return new IteratorCezVektor<T>(this);
    }
}
```

```
public class Slovar<K, V> implements Iterable<K> {
    ...
    @Override
    public Iterator<K> iterator() {
        return new IteratorPoKljucih<K, V>(this);
    }
}
```

Kodo 9.9 lahko sedaj prepišemo takole ...

```
for (int stevilo: stevila) {
    System.out.println(stevilo);
}
```



```
}
```

... kodo 9.11 pa takole:

```
for (String drzava: drzavaVSosed) {
    int stSosed = drzavaVSosed.vrni(drzava);
    System.out.printf("%s -> %d%n", drzava, stSosed);
}
```

Včasih se je po vsebovalniku smiselno sprehajati na več načinov. Na primer, poleg iteratorja, ki se sprehodi od prvega do zadnjega elementa, bi lahko definirali tudi iterator, ki se po elementih sprehodi v obratnem vrstnem redu, iterator, ki elemente obiskuje glede na njihovo naravno urejenost, iterator, ki obišče le vsak  $k$ -ti element itd. V tem primeru z vmesnikom `Iterable` (in metodo `iterator`) definiramo *naravni sprehod* (pri vektorju, denimo, je to sprehod od prvega do zadnjega elementa), alternativne sprehode pa definiramo z drugimi metodami, ki vračajo objekte tipa `Iterator`. Odnos med vmesnikoma `Iterable` in `Iterator` je torej do neke mere podoben odnosu med vmesnikoma `Comparable` in `Comparator`.

#### Naloga 9.15 Napišite metodo

```
public static <T> Iterator<T> vsakDrugi(Iterator<T> iterator)
```

ki vrne iterator, ki svoj sprehod prične na istem objektu kot podani iterator, nato pa se ustavi na vsakem drugem objektu v zaporedju objektov, ki jih obišče podani iterator. Na primer, če bi podani iterator obiskal objekte  $a_0, a_1, a_2, a_3, a_4$  in  $a_5$ , bi vrnjeni iterator obiskal objekte  $a_0, a_2$  in  $a_4$ .

## 9.6 Notranji razredi

Kot že vemo, lahko razred, ki ga potrebujemo samo pri implementaciji razreda  $R$ , za zunanji svet pa ni pomemben, definiramo kot statični notranji razred v razredu  $R$ . Statični notranji razredi se sicer ne razlikujejo bistveno od zunanjih, z nestatičnimi in anonimnimi notranjimi razredi pa je nekoliko drugače. Kot bomo videli, lahko implementacijo vmesnikov z njihovo pomočjo pogosto skrajšamo.

### 9.6.1 Statični notranji razredi

Statični notranji razredi, ponovimo, se obnašajo natanko tako kot samostojni razredi, le da lahko v njihovih konstruktorjih in metodah iz čisto sintaktičnih razlogov neposredno dostopamo do privatnih elementov oklepajočega (zunanjega) razreda in se tako izognemo uporabi »getterjev«. Na primer:

```
public class Zunanji {
```

```

private int a;

public Zunanji(int a) {
    this.a = a;
}

private static class Notranji {
    int b;

    Notranji(Zunanji zun) {
        this.b = zun.a;    // OK
    }
}
}

```

### 9.6.2 Nestatični notranji razredi

*Nestatični* notranji razredi se od statičnih na prvi pogled ločijo le po tem, da v glavi nimajo besedice `static`. Vendar pa se precej drugače obnašajo. Objekt nestatičnega notranjega razreda lahko namreč izdelamo le tako, da ga vežemo na obstoječi objekt oklepajočega razreda. Praviloma to storimo tako, da objekt notranjega razreda ustvarimo v nestatični metodi oklepajočega razreda. Objekt notranjega razreda bo poleg svojih atributov vseboval tudi kazalec *Oklepajoci*. `this`, ki bo kazal na isti objekt kot kazalec `this` v metodi oklepajočega razreda.

Tole morda zveni zapleteno, zato se nam bo primer še kako prilegel:

```

// koda 9.12 (vmesniki/notranjiRazredi/Zunanji.java)
public class Zunanji {
    private int a;

    public Zunanji(int a) {
        this.a = a;
    }

    public void ustvariNotranji(int b) {
        Notranji notranji = this.new Notranji(b);
        // Notranji notranji = new Notranji(b); // krajše
        notranji.izpisi();
    }

    public class Notranji {
        private int b;
    }
}

```

```

    public Notranji(int b) {
        this.b = b;
    }

    public void izpisi() {
        System.out.println(this.b);
        System.out.println(Zunanji.this.a);
        // System.out.println(a); // krajše
    }
}

```

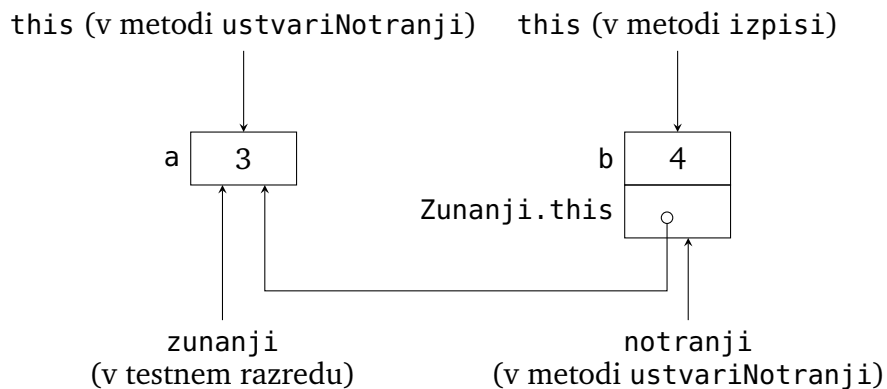
Recimo, da v testnem razredu izvršimo sledečo kodo:

```

// koda 9.13 (vmesniki/notranjiRazredi/TestZunanji.java)
Zunanji zunanji = new Zunanji(3);
zunanji.ustvariNotranji(4);    // najprej se izpiše 4, potem pa 3

```

V metodi `ustvariNotranji` se ustvari objekt `notranji`, ki poleg atributa `b` z vrednostjo 4 vsebuje še (skriti) atribut `Zunanji.this`, ki kaže na objekt razreda `Zunanji`, nad katerim smo poklicali metodo `ustvariNotranji` (v testnem razredu nanj kaže kazalec `zunanji`). Kazalec `Zunanji.this` nam omogoča dostop do atributov in metod tega objekta. Nastalo situacijo prikazuje slika 9.2.



**Slika 9.2** Objekt nestatičnega notranjega razreda vsebuje kazalec na objekt oklepajočega razreda.

Če razred `IteratorCezVektor` napišemo kot nestatični notranji razred, se koda nekoliko poenostavi, saj njegovemu objektu ni več treba posredovati kazalca na vektor, po katerem se bo iterator sprehajal; ta kazalec je namreč kar `Vektor.this`. Poleg tega razred `IteratorCezVektor` ne potrebuje parametra `T`; tipni argument,

s katerim ustvarimo objekt razreda `Vektor`, velja tudi za povezani objekt razreda `IteratorCezVektor`. Tipni parameter `T` v telesu razreda `IteratorCezVektor` se zato nanaša na isti tip kot parameter `T` v telesu razreda `Vektor`.

```
// koda 9.14 (vmesniki/notranjiRazredi/vektorNN/Vektor.java)
public class Vektor<T> implements Iterable<T> {
    ...
    private class IteratorCezVektor implements Iterator<T> {
        private int indeks;

        public IteratorCezVektor() {
            this.indeks = 0;
        }

        @Override
        public boolean hasNext() {
            return this.indeks < Vektor.this.steviloElementov();
        }

        @Override
        public T next() {
            if (!this.hasNext()) {
                throw new NoSuchElementException();
            }
            return Vektor.this.vrni(this.indeks++);
        }
    }
    ...
    @Override
    public Iterator<T> iterator() {
        return this.new IteratorCezVektor();
    }
}
```

### 9.6.3 Anonimni notranji razredi

*Anonimni* notranji razred je statični ali nestatični notranji razred »za enkratno uporabo«. Ustvarimo ga natanko v tistem trenutku, ko potrebujemo njegov objekt. Anonimni notranji razred *in* njegov objekt ustvarimo s kodo, ki je nekakšen križanec med kodo za definicijo razreda in kodo za izdelavo objekta:

```
R objekt = new R() {
```

```

    inicializacija atributov
    redefinicije oz. implementacije metod razreda oz. vmesnika R
    pomožne metode
};

```

Kaj natančno se zgodi pri izvedbi gornje kode?

1. Ustvari se nov brezimeni razred, ki razširja razred *R* oziroma implementira vmesnik *R*. Ustvarjeni razred poleg podedovanih atributov in metod vsebuje še podane attribute, privzeti konstruktor (tj. konstruktor brez parametrov, ki zgolj pokliče konstruktor nadrazreda), redefinicije oz. implementacije metod razreda oz. vmesnika *R* in morebitne pomožne metode.
2. Ustvari se objekt izdelanega razreda.

V anonimnem notranjem razredu ne moremo definirati lastnega konstruktorja, lahko pa v odseku *inicializacija atributov* podamo želene attribute in njihove začetne vrednosti.

Anonimni notranji razred je lahko statičen ali nestatičen: če ga ustvarimo znotraj statične metode, je statičen, če ga izdelamo v okviru nestatične metode, pa je nestatičen. Ni nam treba izrecno opredeljevati, ali je anonimni notranji razred statičen ali nestatičen, saj prevajalnik to ugotovi sam od sebe. Tako kot v običajnem nestatičnem notranjem razredu lahko tudi v nestatičnem anonimnem notranjem razredu s pomočjo kazalca *Oklepajoci.this* dostopamo do atributov in metod objekta oklepajočega razreda, na katerega je vezan objekt *this* anonimnega razreda.

Anonimni notranji razredi omogočajo še več. Tako v statični kot v nestatični različici lahko znotraj njihovih teles dostopamo do *parametrov* in *lokalnih spremenljivk* metode, v kateri ustvarimo anonimni razred in njegov objekt, če so te deklarirane kot konstante (z določilom *final*) ali pa če jih uporabljamo, kot da bi bile konstante (to pomeni, da jih inicializiramo in potem nikoli več ne spremenimo).

Recimo, da definiramo vmesnik, ki predstavlja matematične funkcije  $\mathbb{R} \rightarrow \mathbb{R}$ :

```

public interface Funkcija {
    public double f(double x);
}

```

Sledeča metoda vrne objekt tipa *Funkcija*, ki predstavlja linearno funkcijo s parametroma *k* in *n*:

```

public static Funkcija linearna(double k, double n) {
    return new Funkcija() {
        @Override
        public double f(double x) {
            return k * x + n;
        }
    };
}

```

```

    }
};
}

```

Kot vidimo, izdelamo (statični) anonimni notranji razred, ki implementira vmesnik `Funkcija`. V implementaciji metode `f` lahko uporabljamo parametra `k` in `n`, ker ju zgolj inicializiramo (to se samodejno zgodi ob klicu metode) in potem nikoli ne spremenimo.

S čisto tehničnega vidika ni razloga, zakaj spremenljivk `k` in `n` v metodi `linearna` ne bi smeli spreminjati, vendar pa bi na ta način lahko ustvarili zmotno predstavo o njunem dosegu ali življenjski dobi. Ko se ustvari objekt anonimnega razreda, se trenutni vrednosti spremenljivk `k` in `n` samodejno skopirata v skrita atributa, ki sta znotraj anonimnega razreda dostopna prek (povsem ločenih) spremenljivk `k` in `n`. Drugače ne gre, saj objekt anonimnega razreda obstaja tudi še potem, ko se metoda `linearna` zaključi, njeni lokalni spremenljivki `k` in `n` pa izgineta. Morebitne spremembe spremenljivk `k` in `n` zato ne bi mogle vplivati na spremenljivki `k` in `n` v metodi `f`, tudi če bi bile dovoljene.

Vrnimo se k iteratorju po vektorju. Ker razred `IteratorCezVektor` potrebujemo izključno za izdelavo objekta, ki se bo sprehajal po vektorju, je dober kandidat za pretvorbo v anonimni notranji razred. Naš razred implementira vmesnik `Iterator<T>`, zato ga (skupaj z njegovim objektom) ustvarimo takole:

```

new Iterator<T>() {
    ...
}

```

Atribut `indeks`, ki smo ga inicializirali v konstruktorju, sedaj inicializiramo v odseku *inicializacija atributov*, saj v anonimnem notranjem razredu ne moremo napisati konstruktorja. Preostanek kode je enak kot pri nestatičnem notranjem razredu `IteratorCezVektor`.

```

// koda 9.15 (vmesniki/notranjiRazredi/vektorAnon/Vektor.java)
public class Vektor<T> implements Iterable<T> {
    ...
    public Iterator<T> iterator() {
        return new Iterator<T>() {
            int indeks = 0;    // določilo private je tukaj odveč

            @Override
            public boolean hasNext() {
                return this.indeks < Vektor.this.steviloElementov();
            }
        }
    }
}

```

```

        @Override
        public T next() {
            if (!this.hasNext()) {
                throw new NoSuchElementException();
            }
            return Vektor.this.vrni(this.indeks++);
        }
    };
}
}

```

**Naloga 9.16** Je anonimni notranji razred v kodi 9.15 statičen ali nestatičen?

**Naloga 9.17** Razred `Notranji` v kodi 9.12 spremenite v statični notranji razred in ustrezno prilagodite še preostanek kode. Koliko atributov bo po novem imel razred `Notranji`?

**Naloga 9.18** Razreda, ki v kodi 9.5 implementirata vmesnik `Comparator`, prepisite v anonimna notranja razreda.

**Naloga 9.19** Napišite metodo

```
public static Comparator<Integer> poElementihNaIndeksih(int[] tab)
```

ki vrne primerjalnik, ki podani števili  $i$  in  $j$  primerja glede na elementa, ki se v tabeli nahajata na indeksih  $i$  in  $j$ . Na primer, pri tabeli  $\{20, 50, 10, 40\}$  bi primerjalnikova metoda `compare` za števili 0 in 1 morala vrniti negativno vrednost ( $20 < 50$ ), za števili 0 in 2 pa pozitivno vrednost ( $20 > 10$ ). Pomagajte si z anonimnim notranjim razredom.

**Naloga 9.20** Nalogo 9.4 rešite s pomočjo statičnega notranjega razreda, nato pa še anonimnega notranjega razreda.

**Naloga 9.21** V sledeči kodi nadomestite tropičje tako, da bo izpisovala izide metov kocke, dokler ne bo (v skupnem seštevku) trikrat padla šestica.

```

for (int met: ...) {
    System.out.println(met);
}

```

Namig: izdelajte objekt anonimnega notranjega razreda, ki implementira vmesnik `Iterable<Integer>`.

## 9.7 Povzetek

- Vmesnik je v osnovi abstraktni razred, ki lahko vsebuje le abstraktne metode, privzete metode in statične elemente. Za razliko od razredov pa nam vmesniki omogočajo večkratno dedovanje: razred je lahko podrazred le enega razreda, vendar pa lahko implementira poljubno mnogo vmesnikov.
- Vmesnik lahko razširja poljubno mnogo drugih vmesnikov.
- Vmesnik `Comparable` je namenjen definiciji naravne urejenosti objektov določenega tipa. Če želimo naravno urejenost definirati za objekte razreda  $R$ , določimo, da razred  $R$  implementira vmesnik `Comparable<R>`, nato pa njegovo abstraktno metodo `compareTo` implementiramo tako, da vrne negativno število, če objekt `this` po naravni urejenosti sodi pred objekt, ki je podan kot parameter, 0, če sta si objekta po tem kriteriju enakovredna, in pozitivno vrednost, če objekt `this` sodi za objekt, ki je podan kot parameter.
- Vmesnik `Comparator` je namenjen definiciji alternativne urejenosti ali urejenosti, ki uporablja zunanje vire. Vmesnika `Comparator` ne implementiramo v razredu, za katerega želimo definirati takšno urejenost, ampak v ločenem (ponavadi notranjem) razredu.
- Vmesnik `Iterator` praviloma implementiramo v razredih, ki določajo vsebovalnike. Z implementacijo metod `hasNext` in `next` definiramo sprehod po elementih vsebovalnika.
- Če razred za opis vsebovalnika implementira vmesnik `Iterable`, se lahko po njegovih objektih sprehajamo z zanko *for-each*.
- Notranji razredi so lahko statični, nestatični ali anonimni.
- Statični notranji razredi se obnašajo tako kot samostojni, edina razlika je ta, da lahko (iz golih sintaktičnih razlogov) neposredno dostopajo do privatnih elementov oklepajočega razreda. Objekte statičnih notranjih razredov lahko izdelujemo povsem neodvisno od objektov oklepajočih razredov.
- Objekt nestatičnega notranjega razreda je mogoče izdelati samo tako, da ga vežemo na obstoječi objekt oklepajočega razreda. Novoustvarjeni objekt notranjega razreda bo poleg svojih atributov vseboval še kazalec, ki kaže na povezani objekt oklepajočega razreda. V metodah notranjega razreda lahko zato neposredno dostopamo do elementov povezanega objekta.
- Anonimni notranji razred je brezimeni statični ali nestatični notranji razred za enkratno uporabo: razred izdelamo točno tam, kjer ga potrebujemo, hkrati z razredom pa izdelamo še njegov objekt.



### *Iz profesorjevega kabineta*

»Kako presneto omejene so te mašine!« se priduša profesor Doberšek. »Vsakemu faliranemu študentu matematike so neskončna zaporedja nekaj najbolj naravnega. V te stroje, ki naj bi, tako mulce menda celo učijo v šolah, znali izračunati vse, kar se izračunati dá, pa mi še nikoli ni uspelo stlačiti neskončnega zaporedja in mi bržkone tudi nikoli ne bo!«

»Imate popolnoma prav, profesor Doberšek,« pristavi pregovorni piskrček pokorni profesorjev pomočnik Slapšak. »Toda dovolite mi, da vam predstavim razrede in vmesnike iz paketa `java.util.stream`, ki so namenjeni točno temu: predstavitvi in obdelavi neskončnih zaporedij.«

»Paket `java.util.stream`, praviš? Aha, že vidim ... Jojme, kako je to komplicirano! Neskončna zaporedja so vendar tako zelo preprosta ...!«

»Ta paket je zelo močan,« se vmeša docentka Javornik, »a se splača poglobiti v kakšno knjigo, da dojameš njegov ustroj. Priporočam Urma in sod. (2014). Za prvo silo pa si lahko pomagata z vmesnikom, ki bi bil težko enostavnejši:

```
public interface Zaporedje {
    public abstract int naslednji();
}
```

Metoda `naslednji` i kratko malo vrne naslednji člen zaporedja. Neskončna zaporedja lahko sedaj predstavite z razredi, ki implementirajo ta vmesnik. Ni težko napisati, recimo, razredov za predstavitev zaporedja naravnih števil, Fibonaccijevega zaporedja, zaporedja praštevil ... Obenem lahko napišete metodo, ki vrne vsoto podanih neskončnih zaporedij, metodo, ki vrne zaporedje, ki vsebuje vsak  $k$ -ti element podanega zaporedja, metodo, ki vrne zaporedje, ki je enako podanemu zaporedju, le da ne vsebuje zaporednih podvojenih elementov, itd. Vseskozi delate z neskončnimi zaporedji. Seveda pa lahko svoje omiljeno neskončno zaporedje kadarkoli pretvorite v končno — recimo z metodo, ki vrne vektor s prvimi  $n$  elementi zaporedja.«

Bi znali sprogramirati dobrote, ki jih je iz rokava stresla docentka?



## 10 Vsebovalniki

Z vsebovalniki — objekti, namenjenimi hranjenju poljubnega števila elementov — smo se že srečali. Tabele, vektorji (objekti razreda *Vektor*) in slovarji (objekti razreda *Slovar*) so trije primeri vsebovalnikov. V tem poglavju bomo spoznali vsebovalnike, ki jih ponuja javin paket `java.util`. Ugotovili bomo, da so urejeni v smiselno hierarhijo in da ponujajo številne metode, ki nam lajšajo delo z njimi.

### 10.1 Pregled

Vsem vsebovalnikom, ki jih definirajo razredi in vmesniki v paketu `java.util`, je skupno to, da lahko hranijo le elemente referenčnih tipov. Namesto elementov tipa `int`, `double`, `boolean` itd. lahko vsebujejo le kazalce na objekte tipa `Integer`, `Double`, `Boolean` itd. Tabela potemtakem ni del hierarhije vsebovalnikov, čeprav je seveda prav tako namenjena hranjenju poljubnega števila elementov.

Vsi razredi in vmesniki, povezani z vsebovalniki, so generični. Objekt tipa *Vsebovalnik*<T> (kjer je *Vsebovalnik* poljuben razred ali vmesnik iz hierarhije vsebovalnikov) lahko hrani kazalce na objekte tipa `T` ali kateregakoli njegovega podtipa. Če želimo v istem vsebovalniku hraniti elemente poljubnega referenčnega tipa, izdelamo vsebovalnik tipa *Vsebovalnik*<`Object`>. Takšnim vsebovalnikom se v splošnem raje izogibajmo, saj se z njihovo uporabo dejansko odrečemo samodejnemu preverjanju tipov.

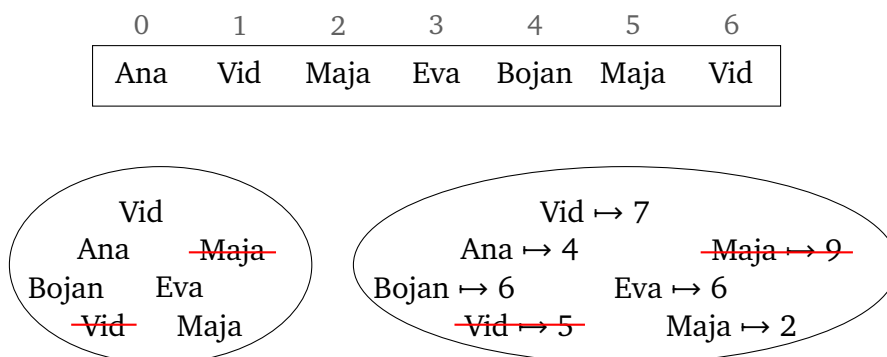
Seznani se bomo s tremi tipi vsebovalnikov:

**Seznam.** Seznam je vsebovalnik, v katerem so položaji posameznih elementov določeni z indeksi. Pri seznamu torej vemo, kateri element je prvi (to je tisti z indeksom 0), kateri drugi (to je tisti z indeksom 1) itd. Poleg tega se elementi v seznamu lahko podvajajo.

**Množica.** Elementi v množici nimajo indeksov, zato ne vemo, kateri je prvi, kateri drugi itd. Poleg tega se elementi v množicah ne morejo podvajati.

**Slovar.** Slovarji hranijo pare ključ-vrednost. Ključi nimajo indeksov (ne moremo vedeti, kateri je prvi, kateri drugi itd.), prav tako pa se ne morejo podvajati. Pri vrednostih ni te omejitve.

Slika 10.1 ilustrira vse tri tipe vsebovalnikov.



**Slika 10.1** Seznam (zgoraj), množica (levo spodaj) in slovar (desno spodaj).

Tabela, kot rečeno, ni del hierarhije vsebovalnikov, če pa bi jo že morali nekam umestiti, bi jo gotovo označili za seznam, saj so njeni elementi dostopni prek indeksov, prav tako pa se lahko podvajajo.

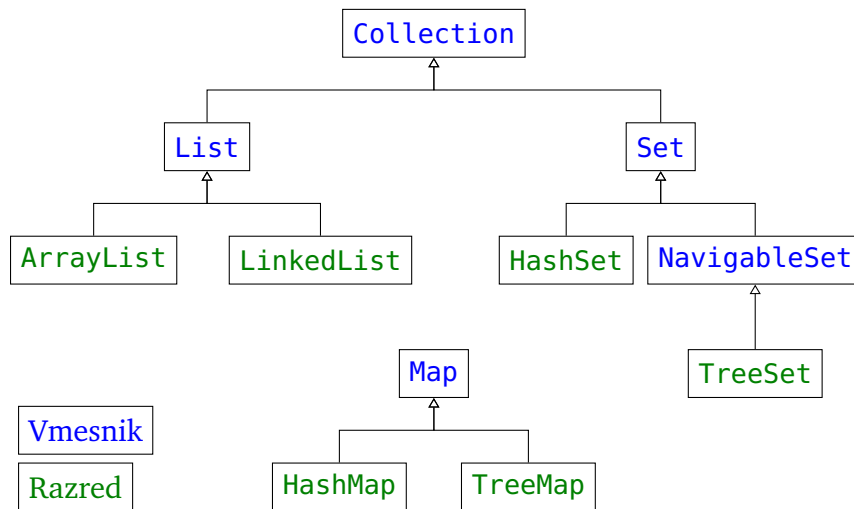
Kako vemo, kateri tip vsebovalnika je najprimernejši za naše potrebe? Kadar delamo z zaporedjem elementov ali pa če želimo do elementov dostopati prek indeksov, uporabimo seznam. Z množico si pomagamo, ko nas zanima predvsem pripadnost elementov (ali določen element pripada množici) ali pa ko potrebujemo učinkovite operacije preseka, unije ali razlike. Slovar pa izdelamo takrat, ko potrebujemo dostop do elementov prek poljubnih ključev. Slovar lahko po eni strani obravnavamo kot posplošitev tabele ali seznama (namesto indeksov imamo ključe), po drugi strani pa kot množico, pri kateri so elementom (ključem) pripisane še vrednosti.

## 10.2 Hierarhija vsebovalnikov

Omejili se bomo na razrede in vmesnike, ki jih prikazuje slika 10.2. Objekti tipa `List` so sezname, objekti tipa `Set` so množice, slovarje pa predstavimo z objekti tipa `Map`. Vsak vmesnik ima eno ali več *implementacij* — razredov, ki ga implementirajo. V osnovi lahko z objekti različnih implementacij počnemo iste reči, se pa lahko posamezne operacije med seboj razlikujejo po učinkovitosti. Na primer, pri objektih razreda `ArrayList` je dostop prek indeksa hiter, odstranjevanje elementa pa (relativno) počasno. Pri objektih razreda `LinkedList` je ravno obratno.

Vmesnika `List` in `Set` sta podvmesnika vmesnika `Collection`. Seznamom in množicam zato s skupnim imenom pravimo tudi *zbirke*. Slovarji niso zbirke, so pa še vedno vsebovalniki.

Poleg razredov in vmesnikov, ki tvorijo opisano hierarhijo, omenimo še razred `Collections` (bodimo pozorni na končni »s«). Ta vsebuje številne statične metode za delo z vsebovalniki različnih tipov.



Slika 10.2 Hierarhija razredov in vmesnikov za predstavitev vsebovalnikov.

### 10.3 Vmesnik Collection

Vmesnik Collection (pravzaprav `Collection<T>`) vsebuje abstraktne metode, ki so smiselne za vse zbirke. Naštejmo jih nekaj:

- `public abstract boolean add(T element)`

Podani element doda v zbirko `this`. Vrne `true` natanko v primeru, če se je zbirka `this` po izvedbi operacije spremenila. (Če je zbirka `this` množica, ki že vsebuje podani element, potem metoda vrne `false`, saj se množica ne spremeni.)

- `public abstract boolean addAll(Collection<? extends T> coll)`

Vse elemente iz zbirke `coll` doda v zbirko `this`. Vrne `true` natanko v primeru, če se je zbirka `this` po izvedbi operacije spremenila. Zaradi določila `? extends T` je lahko zbirka `coll` tipa `Collection<T>` ali `Collection<R>`, kjer je `R` poljuben podtip tipnega parametra `T`. (Metodi lahko seveda podamo tudi argument tipa `List<...>`, `Set<...>`, `ArrayList<...>` itd., saj so vsi ti tipi podtipi tipa `Collection<...>`.)

- `public abstract boolean contains(Object obj)`

Vrne `true` natanko v primeru, če zbirka `this` vsebuje podani objekt.

- `public abstract boolean containsAll(Collection<?> collection)`

Vrne `true` natanko v primeru, če zbirka `this` vsebuje vse elemente zbirke `coll`. Tip zbirke `coll` je lahko parametriziran s poljubnim referenčnim tipom.

- `public abstract boolean remove(Object obj)`  
Iz zbirke `this` odstrani eno od pojavitev podanega objekta. Vrne `true` natanko v primeru, če se je zbirka `this` po izvedbi operacije spremenila.
- `public abstract boolean removeAll(Collection<?> collection)`  
Iz zbirke `this` odstrani vse elemente, ki so prisotni tudi v podani zbirki. Vrne `true` natanko v primeru, če se je zbirka `this` po izvedbi operacije spremenila.
- `public abstract boolean retainAll(Collection<?> collection)`  
V zbirki `this` ohrani samo tiste elemente, ki so prisotni tudi v podani zbirki. Vrne `true` natanko v primeru, če se je zbirka `this` po izvedbi operacije spremenila.
- `public abstract void clear()`  
Iz zbirke `this` odstrani vse elemente.
- `public abstract boolean isEmpty()`  
Vrne `true` natanko v primeru, če je zbirka `this` prazna.
- `public abstract int size()`  
Vrne število elementov v zbirki `this`.
- `public abstract Iterator<T> iterator()`  
Vrne iterator, ki se sprehodi po vseh elementih zbirke `this`.
- `public abstract Object[] toArray()`  
Izdela in vrne tabelo z elementi zbirke `this`.

Vmesnik `Collection<T>` je podvmesnik vmesnika `Iterable<T>`, zato se lahko po vseh zbirkah sprehodimo z zanko *for-each*.

Podvmesniki vmesnika `Collection` natančneje opredeljujejo pričakovano delovanje nekaterih metod. Ker vrstni red elementov v zbirki ni nujno določen (v množici, denimo, ni), za metodo `add` v vmesniku `Collection` ni podano, kam naj element doda. V vmesniku `List` pa je pri metodi `add` izrecno navedeno, da element doda na konec seznama. Podobno velja za metode, ki temeljijo na preverjanju prisotnosti elementov (npr. `contains`, `remove` ...). Pri večini zbirk se objekta *a* in *b* proglasita za enaka, če velja `a.equals(b)` (klic `zbirka.contains(objekt)` torej vrne `true` natanko v primeru, če v zbirki obstaja vsaj en element *e*, za katerega velja `e.equals(objekt)`), obstajajo pa tudi zbirke (npr. `TreeSet`), pri katerih se enakost preverja drugače.

Sledi primer, ki uporablja nekatere metode vmesnika `Collection`. Zbirko `zbirka` ustvarimo kot množico, implementirano z zgoščeno tabelo, nato pa nad njo izvajamo operacije, ki so skupne vsem zbirkam. Metoda `List.of` vrne seznam, sestavljen iz podanih elementov, metoda `Set.of` pa množico, sestavljeno iz podanih elementov.

```
// koda 10.1 (vsebovalniki/collection/PrimeriCollection.java)
import java.util.*;

public class PrimeriCollection {

    public static void main(String[] args) {
        // ustvarimo zbirko ...
        Collection<Integer> zbirka = new HashSet<>();    // (1)

        // ... vanjo dodamo nekaj elementov ...
        zbirka.add(10);
        zbirka.add(20);
        zbirka.addAll(List.of(30, 40, 50));

        // ... jo izpišemo ...
        System.out.println(zbirka);

        // ... iz nje odstranimo element ...
        zbirka.remove(20);

        // ... preverimo, ali vsebuje določene elemente ...
        System.out.println(zbirka.contains(20));
        System.out.println(zbirka.containsAll(Set.of(40, 10)));

        // ... iz nje odstranimo vse elemente razen določenih ...
        zbirka.retainAll(List.of(30, 50, 60));
        System.out.println(zbirka);
        System.out.println(zbirka.size());

        // ... in se po njej še sprehodimo
        for (Integer element: zbirka) {
            System.out.println(element);
        }
    }
}
```

### Naloga 10.1 Napišite metodo

```
public static <T> boolean enakovredni(Collection<T> a,
                                     Collection<T> b)
```

ki vrne true natanko tedaj, ko vsak element zbirke a obstaja tudi v zbirki b in vsak

element zbirke *b* obstaja tudi v zbirki *a*. (To ne pomeni nujno, da sta zbirki enaki; na primer, seznama  $\langle 3, 5, 3 \rangle$  in  $\langle 5, 5, 3, 5 \rangle$  nista enaka, sta pa — za potrebe te naloge — enakovredna.)

### Naloga 10.2 Napišite metodo

```
public static <T extends Comparable<T>>
    void odstraniManjse(Collection<T> zbirka, T meja)
```

ki iz podane zbirke odstrani vse elemente, ki po naravni urejenosti tipa *T* sodijo pred objekt *meja*.

**Naloga 10.3** Razred `Vektor<T>` dopolnite tako, da bo implementiral vmesnik `Collection<T>`. Če vrstico (1) v razredu `PrimeriCollection` (koda 10.1) zamenjate s

```
Collection<Integer> zbirka = new Vektor<>();
```

bi se moral razred pravilno prevesti in pognati.

Namig: priporočamo vam, da vmesnika `Collection` ne implementirate neposredno, ampak da raje razširite abstraktni razred `AbstractCollection`, ki vsebuje privzete implementacije številnih metod vmesnika `Collection`. To pomeni, da bo zadoščalo, če boste implementirali abstraktne metode razreda `AbstractCollection` (teh je bistveno manj kot v vmesniku `Collection`) in redefinirali metode, ki sprožijo izjemo tipa `UnsupportedOperationException` (npr. `add`). Metode `addAll` vam, denimo, ne bo treba redefinirati, saj je v razredu `AbstractCollection` že pravilno implementirana (pomaga si z metodama `iterator` in `add`).

## 10.4 Seznami

Seznami so zbirke, pri katerih je natančno določeno, kateri element je prvi, kateri drugi itd., poleg tega pa lahko vsebujejo več kopij istega elementa. Podobno kot pri navadnih tabelah so elementi seznama dostopni prek *indeksov*: prvi element ima indeks 0, drugi 1 itd.

### 10.4.1 Vmesnik List

Javanski seznami so objekti tipa `List` (oziroma `List<T>`, če smo natančni). Vmesnik `List<T>` je podvmesnik vmesnika `Collection<T>`, zato podeduje vse njegove abstraktne metode, doda pa tudi nekaj lastnih. Navedimo nekatere od njih:

- `public abstract void add(int index, T element)`



V seznam `this` vstavi podani element pred element s podanim indeksom. Če je podani indeks enak številu elementov seznama, doda element na konec seznama.

- `public abstract T get(int index)`

Vrne element, ki se v seznamu `this` nahaja na podanem indeksu.

- `public abstract int indexOf(Object obj)`

Vrne indeks prvega elementa *e* v seznamu `this`, za katerega velja `e.equals(obj)`.

- `public abstract T remove(int index)`

Iz seznama `this` odstrani element na podanem indeksu in vrne odstranjeni element.

- `public abstract void sort(Comparator<? super T> comp)`

Uredi seznam `this` glede na podani primerjalnik. Če je parameter `comp` enak `null`, se seznam uredi po naravni urejenosti (v tem primeru morajo vsi elementi pripadati razredom, ki implementirajo vmesnik `Comparable`).

- `public static <T> List<T> of(T... elements)`

Izdela in vrne *nespremenljiv* seznam s podanimi elementi.<sup>1</sup> Nespremenljivega seznama ni mogoče spreminjati, še vedno pa se lahko po njem sprehajamo in pridobivamo njegove elemente.

#### 10.4.2 Razreda `ArrayList` in `LinkedList`

Ogledali si bomo dve implementaciji vmesnika `List`: razred `ArrayList` in razred `LinkedList`. Oba razreda implementirata metode vmesnika `List` v skladu z njihovimi opisi, razlikujeta pa se po učinkovitosti nekaterih operacij.

Razred `ArrayList` je zelo podoben našemu razredu `Vektor`, saj seznam implementira s *tabelo*. Tako kot pri razredu `Vektor` ločimo med kapaciteto in številom elementov. Kapaciteta je dolžina tabele, število elementov pa je dejansko število elementov, ki jih hrani seznam (to število je vedno manjše ali enako kapaciteti). Začetno kapaciteto lahko po želji nastavimo ob izdelavi seznama. Ko se tabela napolni do svoje kapacitete, se izdela nova, večja tabela, nato pa se vanjo skopirajo elementi originalne tabele. Seznam se tako samodejno »razteza«.

Razred `ArrayList<T>` ponuja sledeče konstruktorje:

---

<sup>1</sup>Oznaka `T ...` pove, da lahko ob klicu metode podamo poljubno število parametrov tipa `T`. Najbolj znan primer metode s poljubnim številom parametrov je `System.out.printf`.

- `public ArrayList()`  
Izdela prazen seznam s kapaciteto 10.
- `public ArrayList(int initialCapacity)`  
Izdela prazen seznam s podano začetno kapaciteto.
- `public ArrayList(Collection<? extends T> collection)`  
Izdela seznam z elementi iz podane zbirke. Elementi se v seznam dodajo v vrstnem redu, ki ga določa iterator nad zbirko.

Drugi konstruktor je specifičen za razred `ArrayList`, konstruktorja, enakovredna prvemu in tretjemu, pa ponujajo vsi razredi iz hierarhije vsebovalnikov, zato ju za ostale razrede ne bomo navajali. Konstruktorji, ki kot svoj parameter sprejmejo zbirko, omogočajo enostavno pretakanje podatkov med različnimi tipi zbirk.

V razdelku 6.10.3 smo nalogo Izštevanka rešili s pomočjo našega lastnega razreda `Vektor`. Lotimo se je še s pomočjo razreda `ArrayList`:

```
// koda 10.2 (vsebovalniki/list/Izstevanka.java)
import java.util.*;

public class Izstevanka {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int stOtrok = sc.nextInt();
        int stBesed = sc.nextInt();

        // preberi imena otrok v seznam
        List<String> otroci = new ArrayList<>();
        for (int i = 0; i < stOtrok; i++) {
            otroci.add(sc.next());
        }

        // število krogov izštevanka
        int stKrogov = stOtrok - 1;

        // simuliraj izštevanko
        for (int krog = 1; krog <= stKrogov; krog++) {
            // ugotovi, kdo izpade
            int ixIzpadlega = (stBesed - 1) % stOtrok;
            System.out.println(otroci.get(ixIzpadlega));

            // izloči izpadlega
        }
    }
}
```

```

        otroci.remove(ixIzpadlega);
        stOtrok--;
    }
}

```

Vidimo, da se razreda razlikujeta bolj ali manj le v imenih metod. No, če pokukamo pod pokrov, ugotovimo, da je razred `ArrayList` varnejši, saj v primeru dostopa do neobstoječega indeksa sproži izjemo. Na primer, koda

```

Vektor<Integer> vektor = new Vektor<>(10);
vektor.nastavi(5, 42);

```

bo v celico na indeksu 5, ki *ni* del vektorja, brez besed vpisala vrednost 42, medtem ko bo koda

```

List<Integer> seznam = new ArrayList<>(10);
seznam.set(5, 42);

```

sprožila izjemo tipa `IndexOutOfBoundsException`. Razred `Vektor` bomo sedaj častno upokojili, saj je svojo nalogo opravil: pokazal nam je, kako lahko tak vsebovalnik napišemo sami.

Zakaj namesto

```

ArrayList<String> otroci = new ArrayList<>();

```

pišemo takole?

```

List<String> otroci = new ArrayList<>();

```

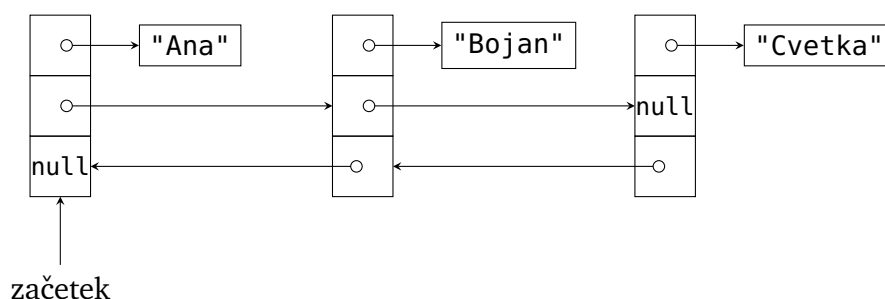
Prvič zato, ker skoraj ničesar ne izgubimo: velika večina metod, ki jih ponuja razred `ArrayList`, obstaja tudi v vmesniku `List`. Drugič pa zato, ker smo na ta način prožnejši: če se odločimo razred `ArrayList` zamenjati s katero drugo implementacijo vmesnika `List`, moramo popraviti zgolj desno stran gornjega prireditvenega stavka. Še posebej pri parametrih metod velja, da se jih splača deklarirati s hierarhično čim višjim tipom, saj so tako karseda splošne. Pri parametru tipa `ArrayList` je argument lahko zgolj tipa `ArrayList`, pri parametru tipa `List` pa je argument lahko tudi tipa `LinkedList` ali katerega drugega podtipa tipa `List`. Pri parametru tipa `Collection` je na voljo še več izbire glede tipa argumenta, seveda pa imamo tudi manj izbire glede metod, ki jih lahko uporabimo.

Dostop do elementa na podanem indeksu (metoda `get`) je pri razredu `ArrayList` zelo učinkovit, saj to velja tudi za navadno tabelo.<sup>2</sup> Učinkovito je tudi dodajanje

<sup>2</sup>Ker so elementi tabele v pomnilniku shranjeni eden za drugim in ker vsi zavzemajo isto število bajtov, zlahka izračunamo pomnilniški naslov elementa na podanem indeksu, če poznamo pomnilniški naslov prvega elementa. Uporabniku tabele ali razreda `ArrayList` se teh podrobnosti ni treba

elementov na konec seznama, razen takrat, ko je treba povečati kapaciteto. Brisanje zadnjega elementa je prav tako učinkovito. Povsem drugače pa je pri vstavljanju in odstranjevanju na nekončnem položaju, saj moramo v tem primeru vse elemente desno od tega položaja premakniti za eno mesto v desno oziroma v levo.

Razred `LinkedList` implementira seznam z *dvosmernim povezanim seznamom* (slika 10.3). Povezani seznam smo spoznali že v okviru razreda `Slovar` (razdelek 7.7), le da je bil tam zgolj enosmerni. Metoda `get` pri razredu `LinkedList` ni učinkovita, saj se moramo za dostop do elementa na indeksu  $i$  sprehoditi po  $i$  elementih. Po drugi strani pa je dodajanje in odstranjevanje elementov učinkovito, saj moramo prevezati zgolj nekaj kazalcev. Seveda pa to velja le ob predpostavki, da smo do želenega položaja že prispeli; če upoštevamo še čas, ki ga potrebujemo za pot do mesta vstavljanja oziroma odstranjevanja, potem ti operaciji v povprečju nista nič učinkovitejši kot pri razredu `ArrayList`. Operaciji vstavljanja in odstranjevanja na začetku in koncu seznama pa sta vedno učinkoviti.



Slika 10.3 Primer dvosmernega povezanega seznama.

`ArrayList` ali `LinkedList`? Odgovor na to vprašanje je odvisen od več faktorjev, v grobem pa se lahko ravnamo po sledečih priporočilih. Za razred `LinkedList` lahko najdemo številne standardne primere uporabe (povezani seznam smo nenazadnje uporabili tudi pri zgošчени tabeli), v večini situacij pa nam bo razred `ArrayList` bolje služil, saj je operacija dostopa do elementa na podanem indeksu pogostejša od operacij vstavljanja in odstranjevanja elementov. Tudi pri izštevanki, kjer v vsakem krogu odstranimo nek element, je razred `ArrayList` hitrejši — preprosto zato, ker moramo pri uporabi razreda `LinkedList` do iskanega elementa najprej prikorakati. Razmere pa so drugačne, če pričakujemo veliko dodajanj in odstranjevanj na začetku in koncu seznama.

Razred `ArrayList` je tudi prostorsko učinkovitejši od razreda `LinkedList` (v povezanem seznamu moramo hraniti tudi kazalce, ki povezujejo sosednja vozlišča), vendar pa je razmerje tem manjše, čim večji so objekti, ki jih hranimo v seznamu.

---

zavedati, jih je pa koristno poznati, če želimo razumeti, zakaj je dostop do elementa na podanem indeksu učinkovit.

Kaj pa dobra stara tabela? V primeru uporabe referenčnih tipov tabela nima skoraj nobenih prednosti pred razredom `ArrayList`. Če delamo s seznamom fiksne dolžine, potem sta tabela in razred `ArrayList` načeloma enakovredna, čeprav nam drugi omogoča uporabo metod, ki jih ponujata razreda `ArrayList` in `Collections`, pri prvi pa smo omejeni na razmeroma skromen razred `Arrays`. Razmere pa so lahko drugačne, kadar rokujemo s primitivnimi tipi. Tabela tipa `int[]` zasede bistveno manj prostora od objekta tipa `ArrayList<Integer>`, saj hrani le vrednosti, ne pa tudi kazalcev nanje. Tabela prekaša razred `ArrayList` tudi po časovni učinkovitosti. Pri uporabi primitivnih tipov je tabela zato marsikdaj najboljša izbira.

**Naloga 10.4** Razred `ArrayList<Integer>`, razred `LinkedList<Integer>` in tabelo tipa `int[]` primerjajte po času, ki ga porabijo operacije dodajanja na konec, dodajanja na začetek, brisanja zadnjega elementa in brisanja prvega elementa. Izberite si primerno veliko število  $N$  in najprej v prazen seznam oz. v prazno tabelo dolžine  $N$  dodajte  $N$  elementov na konec, nato  $N$ -krat pobrišite zadnji element, nazadnje pa na podoben način preizkusite še dodajanje na začetek in brisanje začetnega elementa.

**Naloga 10.5** Napišite razred `Vrsta<T>`, ki implementira *vrsto* — okrnjen seznam, ki omogoča le dve operaciji: dodajanje na konec in odvzemanje z začetka. Razred `Vrsta` naj torej ponuja sledeči metodi:

- `public void dodaj(T element)`  
Doda element na konec vrste `this`.
- `public T odstrani()`  
Odstrani prvi element vrste `this` in ga vrne kot rezultat.

Morda se vam razred splača deklarirati kot podrazred katerega od obstoječih razredov za predstavitev seznamov.

**Naloga 10.6** Napišite razred `PovezaniSeznam<T>`, ki seznam implementira z dvo-smernim povezanim seznamom. Razred naj ponuja iste metode kot naša zadnja različica razreda `Vektor<T>` (razdelek 9.5).

## 10.5 Množice

Pri množici nas praviloma zanima le to, ali ji določen element pripada. Zato ni smiselno, da bi hranila podvojene elemente; v nobenem od javinih razredov, ki implementirajo vmesnik `Set`, to niti ni mogoče. Vmesnik `Set` ne ponuja metode, s katero bi lahko dostopali do elementa na podanem indeksu, saj pri množicah ne moremo govoriti o tem, kateri element je prvi, kateri je drugi itd. Seveda pa bomo pri

sprehodu po množici z iteratorjem elemente še vedno obiskali v določenem vrstnem redu. Ta je v splošnem lahko poljuben, pri množicah tipa `TreeSet` pa je določen s primerjalnikom.

### 10.5.1 Vmesnik Set

Vsi razredi za predstavitev množic implementirajo vmesnik `Set`. Ta vmesnik razširja vmesnik `Collection`, vendar razen statičnih metod `of`, ki vrnejo nespremenljivo množico, sestavljeno iz podanih elementov, ne ponuja nobenih dodatnih funkcionalnosti. Vmesnik `Set` le natančneje določa podedovane abstraktne metode vmesnika `Collection`. Medtem ko je, recimo, metodi `add` iz razreda `Collection` »vseeno«, če zbirka že vsebuje podani element, je pri vmesniku `Set` izrecno navedeno, da metoda podani element v množico doda samo v primeru, če ta ne vsebuje nobenega elementa, ki bi bil enak podanemu. Pojem dvojnika (duplikata) je po specifikaciji vmesnika `Set` določen z metodo `equals` (elementa  $a$  in  $b$  se proglasita za dvojnika, če velja  $a.equals(b)$ ), kot bomo videli, pa implementacija `TreeSet` pogoj za dvojnika opredeljuje drugače.

### 10.5.2 Razred HashSet

Razred `HashSet` implementira množico s pomočjo zgoščene tabele. Zgoščeno tabelo smo v razdelku 7.7.2 uporabili za implementacijo slovarja, vendar pa jo zlahka prilagodimo za množice. Množico si namreč lahko predstavljamo kot slovar, v katerem elementi množice funkcionirajo kot ključi, vrednosti pa nas ne zanimajo (lahko si zamišljamo, da so vse enake `null`). Če zgoščena tabela ni prenapolnjena in če zgoščevalna funkcija (v kombinaciji z ostankom pri deljenju) približno enakomerno razporedi elemente po povezanih seznamih, potem zgoščena tabela omogoča učinkovito preverjanje prisotnosti elementov ter njihovo dodajanje in brisanje.

Kot smo videli pri implementaciji razreda `Slovar`, zgoščena tabela temelji na metodah `equals` in `hashCode`. Če nas zanima, ali zgoščena tabela vsebuje podani ključ, najprej s pomočjo metode `hashCode` izračunamo indeks edinega povezanega seznama, ki bi lahko vseboval iskani ključ, z metodo `equals` pa iskani ključ primerjamo s ključi v vozliščih tega povezanega seznama. Če torej želimo uporabljati razred `HashSet<T>`, mora razred `T` implementirati metodi `equals` in `hashCode`. Metodi morata biti usklajeni ( $a.equals(b) \implies (a.hashCode() == b.hashCode())$ ). Poleg tega se moramo zavedati, da metoda `add` doda element  $a$  v množico tipa `HashSet` samo v primeru, če v njej še ne obstaja element  $b$  z lastnostjo  $b.equals(a)$ .

Definirajmo dva razreda, ki se zgledujeta po našem starem znancu, razredu `Cas`. Razred `CasBrezEH` vsebuje atributa `ura` in `minuta`, konstruktor in metodo `toString`, v razredu `CasEH` pa poleg tega še redefiniramo metodi `equals` in `hashCode`. Spomnimo se: razred `CasBrezEH` ravno tako vsebuje obe metodi (podeduje ju od ra-

zreda Object), vendar pa equals deluje enako kot operator ==, hashCode pa vrne pomnilniški naslov objekta.

*// koda 10.3 (vsebovalniki/set/hash/CasBrezEH.java)*

```
public class CasBrezEH {
    private int ura;
    private int minuta;

    public CasBrezEH(int ura, int minuta) {
        this.ura = ura;
        this.minuta = minuta;
    }

    @Override
    public String toString() {
        return String.format("%d:%02d", this.ura, this.minuta);
    }
}
```

*// koda 10.4 (vsebovalniki/set/hash/CasEH.java)*

```
public class CasEH {
    private int ura;
    private int minuta;

    public CasEH(int ura, int minuta) {
        this.ura = ura;
        this.minuta = minuta;
    }

    public boolean equals(Object drugi) {
        if (this == drugi) {
            return true;
        }
        if (!(drugi instanceof CasEH)) {
            return false;
        }
        CasEH drugiCas = (CasEH) drugi;
        return this.ura == drugiCas.ura &&
            this.minuta == drugiCas.minuta;
    }

    @Override
```

```

    public int hashCode() {
        return 17 * Integer.hashCode(this.ura) +
            31 * Integer.hashCode(this.minuta);
    }

    @Override
    public String toString() {
        return String.format("%d:%02d", this.ura, this.minuta);
    }
}

```

Ustvarimo množici kot objekta razreda `HashSet` in vanju dodajmo nekaj objektov tipa `CasBrezEH` oziroma `CasEH`:

```

// koda 10.5 (vsebovalniki/set/hash/TestCas.java)
Set<CasBrezEH> prva = new HashSet<>();
CasBrezEH cbe = new CasBrezEH(15, 40);
prva.add(cbe);
prva.add(new CasBrezEH(15, 40));
prva.add(cbe);
prva.add(new CasBrezEH(15, 40));

Set<CasEH> druga = new HashSet<>();
CasEH ceh = new CasEH(15, 40);
druga.add(ceh);
druga.add(new CasEH(15, 40));
druga.add(ceh);
druga.add(new CasEH(15, 40));

```

Kaj izpišeta sledeča stavka?

```

System.out.println(prva.size());
System.out.println(druga.size());

```

Razmislimo. Pri prvi množici je enakost objektov določena z njihovo identiteto: objekta *a* in *b* sta proglašena za enaka natanko tedaj, ko gre za en in isti objekt. V množico poskusimo dvakrat dodati isti objekt, v dveh poskusih pa obakrat dodamo nov objekt. Množica na koncu torej vsebuje tri elemente.

Pri drugi množici pa je enakost objektov določena z njihovo vsebino. Ker imajo vsi objekti, ki jih poskusimo dodati, enako vsebino, je v množici na koncu en sam objekt.



**Naloga 10.7** Simetrična razlika množic  $A$  in  $B$  je množica  $A \triangle B = (A \setminus B) \cup (B \setminus A)$ . Napišite metodo

```
public static <T> Set<T> simetricnaRazlika(Set<T> a, Set<T> b)
```

ki vrne simetrično razliko podanih množic. Metoda naj *ne* spremeni množic  $a$  in  $b$ .

**Naloga 10.8** Kako bi morali definirati metodo `equals` v razredu `CasEH`, da bo sledeča koda izpisala vrednost 2? (To vprašanje ima zgolj teoretičen pomen. V praksi se z metodo `equals` ne »igramo«, ampak jo definiramo v skladu z uveljavljenimi smernicami.)

```
Set<CasEH> mnozica = new HashSet<>();
mnozica.add(new CasEH(15, 40));
mnozica.add(new CasEH(15, 40));
mnozica.add(new CasEH(15, 40));
mnozica.add(new CasEH(15, 40));
System.out.println(mnozica.size());
```

**Naloga 10.9** Napišite metodo

```
public static <T> Set<? extends Set<T>> potencna(Set<T> mnozica)
```

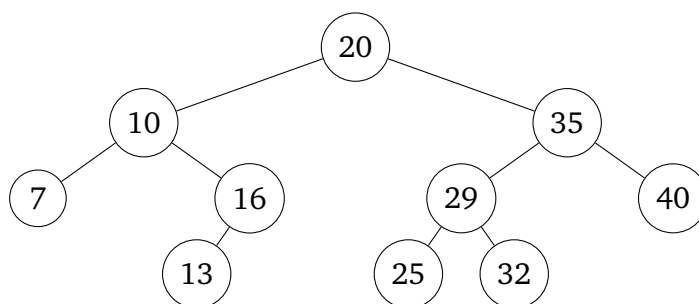
ki vrne potenčno množico podane množice. Upoštevajte, da velja  $\mathcal{P}(\emptyset) = \{\emptyset\}$ , množico  $\mathcal{P}(A)$  za neprazno množico  $A$  pa dobimo kot unijo množice  $\mathcal{P}(A \setminus \{e\})$ , kjer je  $e$  poljuben element množice  $A$ , in množice množic, ki jih dobimo tako, da vsaki od množic v  $\mathcal{P}(A \setminus \{e\})$  dodamo element  $e$ .

### 10.5.3 Razred `TreeSet`

Razred `TreeSet` implementira množico s pomočjo *dvojiškega iskalnega drevesa*. Dvojiško drevo je podatkovna struktura, sestavljena iz medsebojno povezanih vozlišč, v kateri ima vsako vozlišče največ dva naslednika (otroka), vsako vozlišče razen korenskega pa ima natanko enega predhodnika (starša). Vozlišča hranijo elemente množic (kazalce na objekte). *Levo poddrevo* danega vozlišča je sestavljeno iz njegovega levega otroka in vseh potomcev (neposrednih in posrednih naslednikov) tega otroka, *desno poddrevo* pa tvori desni otrok in vsi njegovi potomci.

Dvojiška *iskalna* drevesa so urejena: za vsako vozlišče  $v$  velja, da so vsi elementi v njegovem levem poddrevesu strogo manjši, vsi elementi v njegovem desnem poddrevesu pa strogo večji od elementa v vozlišču  $v$ . Če je dvojiško iskalno drevo vsaj približno uravnoteženo, omogoča hitro iskanje, pa tudi dodajanje in odstranjevanje elementov. Na primer, kako bi preverili, ali drevo na sliki 10.4 vsebuje element 14?

Element 14 je manjši od elementa v korenu (20), zato vemo, da se lahko nahaja kvečjemu v njegovem levem poddrevesu. Na ta način v optimalnem primeru (če je drevo idealno uravnoteženo) že takoj izločimo polovico elementov. Nato element 14 primerjamo z elementom v korenu tega poddrevesa, torej z elementom 10. Vidimo, da je element 14 večji, zato se usmerimo desno na element 16. V naslednjem koraku gremo levo na element 13, nato pa spet desno. Ker tam ni več ničesar, lahko zaključimo, da drevo ne vsebuje elementa 14.



**Slika 10.4** Dvojiško iskalno drevo.

Dvojiško iskalno drevo temelji na primerjanju elementov. Objekt razreda `TreeSet` (oz. `TreeSet<T>`) lahko tako ustvarimo s konstruktorjem

```
public TreeSet(Comparator<? super T> comparator)
```

ki sprejme primerjalnik, ki se bo uporabljal za medsebojno primerjavo dodanih elementov, če pa uporabimo privzeti konstruktor, lahko v objekt razreda `TreeSet` dodajamo samo objekte, ki pripadajo razredom, ki implementirajo vmesnik `Comparable`. V prvem primeru se bodo elementi med seboj primerjali s primerjalnikovo metodo `compare`, v drugem pa z metodo `compareTo` iz njihovega lastnega razreda (oziroma z metodami `compareTo` iz njihovih lastnih razredov, če pripadajo različnim razredom).

Metoda `compareTo` oziroma `compare` se uporablja tudi za preverjanje prisotnosti elementa. Elementa  $a$  in  $b$  se proglasita za dvojnika, če velja

```
a.compareTo(b) == 0
```

oziroma

```
primerjalnik.compare(a, b) == 0
```

V razredu  $T$ , ki mu pripadajo elementi množice tipa `TreeSet<T>`, nam torej sploh ni treba redefinirati metode `equals`, saj je metode razreda `TreeSet` ne bodo klicale. Tudi če metodo `equals` redefiniramo, ni nujno, da zagotovimo konsistentnost z metodo `compare` oz. `compareTo`. Na primer, medtem ko je pri razredu `Cas` takšna

konsistentnost smiselna (objekta, ki predstavljata isti trenutek, je smiselno proglašiti za enaka tako po metodi `equals` kot po metodi `compareTo`), pri razredu `Oseba` ni (objekta, ki predstavljata osebi z istima imenom in priimkom, sta pri naravni urejenosti enaka po metodi `compareTo`, ne pa po metodi `equals`, saj še vedno predstavljata dve različni osebi). Zavedati pa se moramo, da neusklajenost metod `equals` in `compareTo` pomeni, da razred `TreeSet` ne bo skladen z zahtevami vmesnika `Set`, saj ta določa, da se obstoj dvojnika preverja s pomočjo metode `equals`.

Če se po množici tipa `TreeSet` sprehodimo z iteratorjem, bodo dobljeni elementi urejeni glede na implementacijo vmesnika `Comparable` ali `Comparator` (odvisno od tega, s katerim konstruktorjem izdelamo množico). Enak vrstni red dobimo pri uporabi metode `toString`, saj si ta pomaga z iteratorjem.

Čas je za primer. Razred `Oseba` (koda 9.3 in 9.5) implementira vmesnik `Comparable`, vsebuje pa tudi metode, ki vračajo objekte tipa `Comparator`. V naslednjem odseku kode se bodo osebe iz množice `a` izpisale v naravnem vrstnem redu, torej primarno po leksikografsko urejenih priimkih, sekundarno pa po leksikografsko urejenih imenih, kot namreč določa naša implementacija vmesnika `Comparable`. Osebe iz množice `b` pa bodo razvrščene po spolu (ženske pred moškimi), v okviru istega spola pa po padajoči starosti.

```
// koda 10.6 (vsebovalniki/set/tree/TestOseba.java)
Oseba[] osebe = {
    new Oseba("Janez", "Novak", 'M', 1970),
    new Oseba("Janez", "Novak", 'M', 1952),
    new Oseba("Jan", "Novak", 'M', 1970),
    new Oseba("Mojca", "Novak", 'Z', 1982),
    new Oseba("Mojca", "Pirc", 'Z', 1954),
    new Oseba("Ivan", "Pirc", 'M', 1973),
    new Oseba("Darja", "Fink", 'Z', 1954)
};

Set<Oseba> a = new TreeSet<>();
for (Oseba oseba: osebe) {
    a.add(oseba);
}
System.out.println(a);

Set<Oseba> b = new TreeSet<>(Oseba.primerjalnikPoSpoluInStarosti());
for (Oseba oseba: osebe) {
    b.add(oseba);
}
System.out.println(b);
```

Gornja koda proizvede tak izpis (z dodanimi prelomi vrstice):

```
[Darja Fink (Z), 1954, Jan Novak (M), 1970, Janez Novak (M), 1970,
    Mojca Novak (Z), 1982, Ivan Pirc (M), 1973, Mojca Pirc (Z), 1954]
[Mojca Pirc (Z), 1954, Mojca Novak (Z), 1982, Janez Novak (M), 1952,
    Janez Novak (M), 1970, Ivan Pirc (M), 1973]
```

Vidimo, da v množici ne moremo imeti več oseb, ki se po primerjalnem kriteriju med seboj ne razlikujejo, tudi če se razlikujejo po drugih atributih.

Razred `TreeSet` implementira vmesnik `NavigableSet`. Ta vmesnik razširja vmesnik `Set` z metodami, ki temeljijo na urejenosti. Oglejmo si nekatere od njih:

- `T first()`

Vrne najmanjši element množice `this` glede na podani primerjalnik. (To je prvi element, ki ga dobimo pri prehodu po množici z iteratorjem.)

- `T floor(T element)`

Vrne največji element množice `this`, ki je manjši ali enak podanemu elementu. Če takega elementa ni, vrne `null`.

- `NavigableSet<T> headSet(T element, boolean inclusive)`

Vrne podmnožico množice `this`, sestavljeno iz elementov, strogo manjših od podanega elementa (če je parameter `inclusive` enak `false`) oziroma manjših ali enakih podanemu elementu (če je parameter `inclusive` enak `true`).

Metode `last`, `ceiling` in `tailSet` se obnašajo enako kot metode `first`, `floor` in `headSet`, le da vrnejo največji element (`last`), najmanjši element, ki je večji ali enak podanemu (`ceiling`), in množico elementov, večjih od podanega oz. večjih ali enakih podanemu (`tailSet`).

Sledeči primer temelji na naravni urejenosti objektov tipa `Oseba`:

```
// koda 10.7 (vsebovalniki/set/navigable/TestOseba.java)
NavigableSet<Oseba> osebe = new TreeSet<>(Set.of(
    new Oseba("Janez", "Novak", 'M', 1970),
    new Oseba("Jan", "Novak", 'M', 1970),
    new Oseba("Mojca", "Novak", 'Z', 1982),
    new Oseba("Mojca", "Pirc", 'Z', 1954),
    new Oseba("Ivan", "Pirc", 'M', 1973),
    new Oseba("Darja", "Fink", 'Z', 1954)
));

System.out.println(osebe.first());    // Darja Fink (Z), 1954
System.out.println(osebe.last());     // Mojca Pirc (Z), 1954
```

```
Oseba mejna = new Oseba("Katja", "Novak", 'Z', 1975);
System.out.println(osebe.floor(mejna));    // Janez Novak (M), 1970
System.out.println(osebe.ceiling(mejna));   // Mojca Novak (Z), 1982
```

Mimogrede smo spoznali nov način izdelave množic (oziroma vsebovalnikov na sploh). Metoda `Set.of` izdelava nespremenljivo množico podanih elementov, konstruktor razreda `TreeSet` pa elemente te množice doda v novoustanovljeno spreminljivo množico. Na podoben način lahko izdelujemo tudi vsebovalnike drugih tipov.

Metodi `headSet` in `tailSet` sta zanimivi, saj ne vrneta kopije ustrezne podmnožice množice `this`, ampak *pogled* na podmnožico. To pomeni, da se vse operacije, ki jih izvedemo na dobljeni podmnožici, odražajo tudi na izvorni množici in obratno. Na primer:

```
// koda 10.8 (vsebovalniki/set/navigable/TestStevila.java)
NavigableSet<Integer> mnozica =
    new TreeSet<>(Set.of(10, 20, 30, 40, 50));
NavigableSet<Integer> podmnozica = mnozica.headSet(30, true);
System.out.println(podmnozica); // [10, 20, 30]

podmnozica.add(5);
System.out.println(podmnozica); // [5, 10, 20, 30]
System.out.println(mnozica);    // [5, 10, 20, 30, 40, 50]

mnozica.remove(20);
System.out.println(podmnozica); // [5, 10, 30]
System.out.println(mnozica);    // [5, 10, 30, 40, 50]

podmnozica.add(35); // sproži se izjema tipa IllegalArgumentException
```

**Naloga 10.10** Razred `PostniNaslov` z atributi `ulicaHS` (tipa `String`), `postnaStevilka` (tipa `int`) in `posta` (tipa `String`) napišite tako, da bosta obe sledeči metodi vrnila kopijo podane zbirke, a brez duplikatov. Objekta tipa `PostniNaslov` proglasimo za dvojnika, če sta enaka po vseh treh atributih.

```
public static Collection<PostniNaslov> filtriraj1(
    Collection<PostniNaslov> naslovi) {
    return new HashSet<>(naslovi);
}

public static Collection<PostniNaslov> filtriraj2(
    Collection<PostniNaslov> naslovi) {
```

```
return new TreeSet<>(naslovi);
}
```

**Naloga 10.11** Napišite metodo

```
public static List<Ulomek> med(List<Ulomek> ulomki,
                               Ulomek tla, Ulomek strop)
```

ki vrne seznam tistih ulomkov iz podanega seznama, ki so večji ali enaki ulomku tla in strogo manjši od ulomka strop. Telo metode naj obsega zgolj stavek return.

**Naloga 10.12** Za vsak  $n \geq 0$  obstaja *natanko eno* popolnoma uravnoreženo dvojiško iskalno drevo z elementi od 1 do vključno  $2^n - 1$ . Na primer, pri  $n = 3$  se na prvem nivoju tega drevesa nahaja element 4, na drugem elementa 2 in 6, na tretjem pa elementi 1, 3, 5 in 7. Napišite metodo

```
public static int naMestu(int n, int nivo, int indeks)
```

ki za podani  $n$  vrne element, ki se v popolnoma uravnoreženem dvojiškem iskalnem drevesu nahaja na podanem indeksu znotraj podanega nivoja (indeks 0 pomeni prvi element, indeks 1 drugi itd.).

**10.5.4 Uporaba množic**

O množicah smo kar dosti povedali, a še vedno ne vemo čisto dobro, kdaj bi nam lahko prišle prav. V primerjavi s tabelami in razredom `ArrayList` so množice učinkovitejše pri dodajanju in brisanju elementov, dostop do posameznih elementov pa je mogoč le prek iteratorja, kar je seveda bistveno manj učinkovito od dostopa prek indeksa. Seveda množice pridejo v poštev le takrat, ko nimamo oziroma nočemo podvojenih elementov in ko nas vrstni red elementov bodisi ne zanima (`HashSet`) ali pa je določen s primerjalnikom (`TreeSet`).

**Primer 10.1.** Napišimo program, ki za podano vhodno zaporedje besed, ki so med seboj ločene s poljubnim nepraznim zaporedjem presledkov, tabulatorjev in prelomov vrstice, izpiše leksikografsko urejen seznam besed, ki se pojavijo vsaj po enkrat.

*Rešitev.* Program je presenetljivo kratek. S pomočjo klica `sc.next()` po vrsti beremo besede in vsako dodamo v množico tipa `TreeSet`. Na ta način ubijemo dve muhi na en mah: brez kakršnegakoli dodatnega napora se znebimo dvojnikov, obenem pa dosežemo, da bodo besede v izpisu nanizane v leksikografskem vrstnem redu, saj razred `String` implementira vmesnik `Comparable`.

```
// koda 10.9 (vsebovalniki/set/besede/Besede.java)
```

```

import java.util.*;

public class Besede {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Set<String> mnozica = new TreeSet<>();

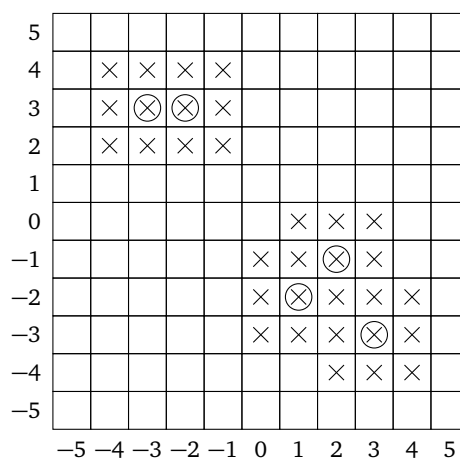
        while (sc.hasNext()) {
            String beseda = sc.next();
            mnozica.add(beseda);
        }
        for (String beseda: mnozica) {
            System.out.println(beseda);
        }
    }
}

```

□

**Primer 10.2.** Na celoštevilski koordinatni sistem sveti  $n \in [1, 10^5]$  luči. Vsaka luč osvetljuje celico tik pod seboj in vseh njenih osem sosed. Napišimo program, ki za podane položaje luči izpiše število osvetljenih celic. Položaji luči so podani s pari celoštevilskih koordinat  $(x, y)$ , pri čemer je  $x, y \in [-M, M]$  in  $M = 10^9$ .

Na primer, pri  $n = 6$  in lučeh na koordinatah  $(1, -2)$ ,  $(-2, 3)$ ,  $(2, -1)$ ,  $(3, -3)$ ,  $(-3, 3)$  in  $(2, -1)$  je osvetljenih 32 celic (slika 10.5). V tem primeru se dve luči nahajata na istem mestu, vendar pa to ne vpliva na domet osvetlitve.



Slika 10.5 Razporeditev luči in osvetljenost v javnem testnem primeru.

Predpostavimo, da je v prvi vrstici vhoda podano število  $n$ , v naslednjih  $n$  vrsticah pa položaji posameznih luči. Za primer s slike 10.5 bi vhod torej izgledal takole:

```
6
1 -2
-2 3
2 -1
3 -3
-3 3
2 -1
```

Program bi za ta vhod izpisal

```
32
```

*Rešitev.* Če bi bilo število  $M$  (precej) manjše, denimo 1000, bi si lahko pripravili tabelo z  $(2M + 3) \times (2M + 3)$  elementi `false` in za vsako luč nastavili na `true` vse elemente tabele, ki pripadajo osvetljenim celicam. Nato bi se po tabeli sprehodili in prešteli elemente `true`. Pri  $M = 10^9$  pa je takšna rešitev vsaj v času pisanja te knjige povsem neuporabna. Tabela bi potrebovala približno  $4 \cdot 10^{18} \text{ B} \approx 4 \cdot 10^9 \text{ GiB}$  pomnilnika (v letu 2023 znaša tipična kapaciteta pomnilnika 16 GiB), a tudi če bi nam nekako uspelo zagotoviti zadostno količino pomnilnika, bi sprehod po tabeli lahko trajal več let.

Druga možnost je, da koordinate osvetljenih celic neposredno shranjujemo v tabelo. Pri  $n$  lučeh lahko imamo največ  $8n$  osvetljenih celic. Ker je  $n$  lahko največ  $10^5$ , takšno tabelo zlahka shranimo v pomnilnik in tudi sprehod po njej traja le delček sekunde. Nekoliko težji zalogaj pa je iskanje dvojnikov: vsako osvetljeno celico moramo šteti le enkrat, tudi če jo osvetljuje več luči. Naivni pristop k štetju osvetljenih celic (za vsak  $i \in \{0, \dots, k-1\}$ , kjer je  $k$  dolžina tabele, se sprehodimo po elementih tabele z indeksi  $0, 1, \dots, i-1$  in preverimo, ali je kateri od njih enak elementu na indeksu  $i$ ; če ni, povečamo števec za 1) je neučinkovit: avtorjev računalnik že za  $n = 10^4$  potrebuje okrog 4,5 sekunde, pri  $n = 10^5$  pa poraba časa naraste na približno 20 minut. Bistveno učinkovitejšo rešitev dobimo, če tabelo osvetljenih celic najprej uredimo (npr. primarno po koordinatah  $x$  in sekundarno po koordinatah  $y$ ). V urejeni tabeli so vsi dvojniki nanizani zaporedno, zato se izognemo časovno potratni dvojni zanki. Nekoliko jedrnatejša (a nič manj učinkovita) rešitev nastane, če namesto tabele uporabimo seznam tipa `ArrayList`.

Najelegantnejša pot do rešitve pa vodi prek razreda `HashSet` (ali `TreeSet`). Ideja je enostavna:

- Napišemo razred `Celica` z atributoma  $x$  in  $y$ , konstruktorjem ter metodama `equals` in `hashCode`.



- Pripravimo si množico tipa `HashSet<Celica>` in vanjo za vsako luč dodamo koordinate celic, ki jih luč osvetljuje.
- Izpišemo velikost množice.

Množica bo sama poskrbela, da v njej ne bo podvojenih celic; zagotoviti moramo le pravilno in konsistentno implementacijo metod `equals` in `hashCode`. Nič lažjega: celici sta enaki natanko v primeru, če imata enaki koordinati  $x$  in  $y$ . Za takšni celici mora metoda `hashCode` vrniti enak rezultat.

Razred `Celica` lahko sedaj domala stresemo iz rokava:

```
// koda 10.10 (vsebovalniki/set/luci/Celica.java)
public class Celica {
    private int x;
    private int y;

    public Celica(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Celica)) {
            return false;
        }
        Celica celica = (Celica) obj;
        return this.x == celica.x && this.y == celica.y;
    }

    @Override
    public int hashCode() {
        return 17 * this.x + 31 * this.y;
    }
}
```

Glavni razred je prav tako kratek in eleganten:

```
// koda 10.11 (vsebovalniki/set/luci/Luci.java)
import java.util.*;
```

```

public class Luci {
    private static final int[][] ODMIKI = {
        {-1, -1}, {0, -1}, {1, -1},
        {-1, 0}, {0, 0}, {1, 0},
        {-1, 1}, {0, 1}, {1, 1}
    };

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int stLuci = sc.nextInt();
        Set<Celica> osvetljene = new HashSet<>();

        for (int i = 0; i < stLuci; i++) {
            int x0 = sc.nextInt();
            int y0 = sc.nextInt();
            for (int[] odmik: ODMIKI) {
                osvetljene.add(new Celica(x0 + odmik[0],
                                           y0 + odmik[1]));
            }
        }
        System.out.println(osvetljene.size());
    }
}

```

Naš program je dovolj učinkovit, da avtorjev računalnik pri  $n = 10^5$  izpiše rezultat v manj kot sekundi. □

**Naloga 10.13** S karseda majhno spremembo kode 10.9 dosežite, da se bodo besede izpisale po naraščajoči dolžini, besede iste dolžine pa leksikografsko.

**Naloga 10.14** V programu iz primera 10.2 se uporaba razreda `TreeSet` namesto razreda `HashSet` zdi pretirana, saj nam celic ni treba urejati; zadošča, da imamo možnost preverjanja njihove enakosti. Kljub temu je rešitev z razredom `TreeSet` še krajša (in ne bistveno manj učinkovita). Prepričajte se, da je res tako.

**Naloga 10.15** Nalogo iz primera 10.2 rešite z uporabo razreda `ArrayList` in njegove metode `sort`.

## 10.6 Slovarji

Slovarji v javi so objekti tipa `Map`, ne pa objekti tipa `Collection`, saj ne gre za običajne zbirke elementov. Ker pa se ključi ne morejo podvajati in ker njihov vrstni red

načeloma ni določen, lahko slovarje obravnavamo kot množice ključev s pripisanimi vrednostmi. Zato je hierarhija vmesnikov in razredov za predstavitev slovarjev zelo podobna tisti za predstavitev množic.

Najprej si bomo ogledali vmesnik `Map`, nato pa dve njegovi implementaciji: razreda `HashMap` in `TreeMap`. Na koncu bomo spoznali, kako nam lahko slovarji koristijo v praksi.

### 10.6.1 Vmesnik `Map`

Vmesnik `Map` je izhodišče hierarhije vmesnikov in razredov za predstavitev slovarjev. Vmesnik je deklariran kot

```
public interface Map<K, V>
```

pri čemer je `K` tip ključa, `V` pa tip vrednosti. Naštejmo nekaj njegovih metod:

- `public abstract boolean containsKey(Object key)`  
Vrne `true` natanko v primeru, če slovar `this` vsebuje podani ključ.
- `public abstract V get(Object key)`  
Vrne vrednost, ki v slovarju `this` pripada podanemu ključu. Če slovar ne vsebuje podanega ključa, vrne `null`.
- `public abstract V put(K key, V value)`  
Če slovar `this` ne vsebuje ključa `key`, potem metoda v slovar doda ključ `key` in pripadajočo vrednost `value`, v nasprotnem primeru pa vrednost, ki pripada ključu `key`, nastavi na `value`. V prvem primeru vrne `null`, v drugem pa prejšnjo vrednost pri podanem ključu.
- `public abstract void putAll(Map<? extends K, ? extends V> map)`  
V slovar `this` doda vse pare ključ-vrednost iz slovarja `map`. Vrednosti pri ključih, ki že obstajajo, se nadomestijo s pripadajočimi vrednostmi iz slovarja `map`.
- `public abstract V remove(Object key)`  
Iz slovarja `this` odstrani podani ključ in pripadajočo vrednost, če obstajata, v nasprotnem primeru pa ne stori ničesar. Vrne vrednost, ki je pripadala podanemu ključu pred njegovo odstranitvijo, oziroma `null`, če slovar ni vseboval podanega ključa.
- `public abstract boolean isEmpty()`  
Vrne `true` natanko v primeru, če je slovar `this` prazen.

- `public abstract int size()`  
Vrne število parov ključ-vrednost v slovarju `this`.
- `public abstract void clear()`  
Izprazni slovar `this`.
- `public abstract boolean containsValue(Object value)`  
Vrne `true` natanko v primeru, če slovar `this` vsebuje podano vrednost.
- `public abstract Set<K> keySet()`  
Vrne množico ključev slovarja `this`.
- `public abstract Set<Map.Entry<K, V>> entrySet()`  
Vrne množico parov ključ-vrednost v slovarju `this`. Vmesnik `Map.Entry` ponuja metodi `getKey` in `getValue`, ki vrmeta ključ oziroma vrednost, ki ju hrani par `this`.
- `public abstract Collection<V> values()`  
Vrne zbirko vrednosti v slovarju `this`.
- `public static <K, V> Map<K, V> of():`  
Vrne nespremenljiv prazen slovar.
- `public static <K, V> Map<K, V> of(K k1, V v1):`  
Vrne nespremenljiv slovar, ki vsebuje ključ `k1` in pripadajočo vrednost `v1`.
- `public static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2):`  
Vrne nespremenljiv slovar, ki vsebuje para `(k1, v1)` in `(k2, v2)`. Obstajajo tudi metode `of`, ki sprejmejo tri in več parov ključ-vrednost.

**Naloga 10.16** Zakaj metodi `keySet` in `entrySet` vrmeta objekt tipa `Set`, metoda `values` pa objekt tipa `Collection`?

**Naloga 10.17** Napišite metodo

```
public static <K, V> boolean jeInjektiven(Map<K, V> slovar)
```

ki vrne `true` natanko v primeru, če se vsak medsebojno različen par ključev preslika v medsebojno različni vrednosti.

### 10.6.2 Razreda HashMap in TreeMap

Razreda HashMap in TreeMap sta pri slovarjih (če jih obravnavamo kot množice ključev s prirejenimi vrednostmi) natanko to, kar sta razreda HashSet in TreeSet pri množicah. Slovar tipa `HashMap<K, V>` je neurejen, pri iskanju ključev pa se uporabljata metodi `hashCode` in `equals` nad objekti tipa `K`. Metodi `hashCode` in `equals` morata biti usklajeni. Slovar vsebuje ključ `k` (`slovar.containsKey(k)`), če v njem obstaja ključ `k'`, tako da velja `k'.equals(k)`.

Slovar tipa `TreeMap<K, V>` je urejen po ključih. Če želimo uporabljati tak slovar, ga lahko ustvarimo s konstruktorjem, ki sprejme objekt tipa `Comparator<K>`, če ga ustvarimo s privzetim konstruktorjem, pa lahko vanj dodajamo samo take ključe, ki pripadajo razredom, ki implementirajo vmesnik `Comparable`. V slovarju tipa `TreeMap` se ključi primerjajo in iščejo s pomočjo metode `compareTo` (če slovar ustvarimo s privzetim konstruktorjem) oziroma `compare` (če mu ob izdelavi posredujemo primerjalnik). Slovar vsebuje ključ `k` (`slovar.containsKey(k)`), če v njem obstaja ključ `k'`, tako da velja

```
k'.compareTo(k) == 0
```

oziroma

```
primerjalnik.compare(k', k) == 0
```

Metoda `equals` se ne uporablja.

Razliko med razredoma `HashMap` in `TreeMap` si oglejmo na primeru slovarja opravkov. V tem slovarju so ključi tipa `Cas` (uporabljamo različico iz razdelka 9.2 — koda 9.2), vrednosti pa tipa `String`. Slovar tipa `HashMap` ni urejen. Čeprav razred `Cas` implementira vmesnik `Comparable<Cas>`, se to dejstvo ignorira. Iskanje ključev temelji na metodah `hashCode` in `equals`: s pomočjo prve metode se poišče povezani seznam, ki edini lahko vsebuje iskani ključ, druga pa se uporablja za primerjavo iskanega ključa s ključi v povezanem seznamu.

Najprej se poigrajmo s slovarjem tipa `HashMap`:

```
// koda 10.12 (vsebovalniki/map/hash/Opravki.java)
// ustvarimo slovar in vanj dodajmo nekaj parov termin-dejavnost
Map<Cas, String> opravki = new HashMap<>();
opravki.put(new Cas( 8, 15), "predavanja");
opravki.put(new Cas(11, 10), "govorilne ure");
opravki.put(new Cas(12, 30), "kosilo");
opravki.put(new Cas(14,  0), "sestane");
opravki.put(new Cas(18, 30), "večerja");

// izpišimo vsebino slovarja
System.out.println(opravki); // 18:30=večerja, 14:00=sestane, ...
```

```

// pridobimo vrednosti za obstoječ in neobstoječ ključ
System.out.println(opravki.get(new Cas(14, 0))); // sestanek
System.out.println(opravki.get(new Cas(12, 0))); // null

// preverimo prisotnost ključa
System.out.println(opravki.containsKey(new Cas(8, 15))); // true

// odstranimo par ključ-vrednost
opravki.remove(new Cas(14, 0));

// spremenimo vrednost, prirejeno enemu od ključev
opravki.put(new Cas(18, 30), "tenis");

// izpišimo vsebino slovarja
System.out.println(opravki); // 18:30=tenis, 14:00=sestanek, ...

// izpišimo slovar s sprehodom po množici ključev
for (Cas cas: opravki.keySet()) {
    System.out.printf("%s -> %s%n", cas, opravki.get(cas));
}
// 18:30 -> tenis
// 11:10 -> govorilne ure
// ...

```

Vrstni red izpisov je v splošnem lahko poljuben.

Pri slovarju tipa `TreeMap` bo slovar urejen po ključih, in sicer tako, kot narekuje naravna urejenost razreda `Cas`.

```

// koda 10.13 (vsebovalniki/map/tree/Opravki.java)
// metoda Map.of ustvari nespremenljiv slovar, a ga lahko podtaknemo
// konstruktorju razreda HashMap ali TreeMap in dobimo običajen
// spremljiv slovar
Map<Cas, String> opravki = new TreeMap<>(Map.of(
    new Cas(8, 15), "predavanja",
    new Cas(11, 10), "govorilne ure",
    new Cas(12, 30), "kosilo",
    new Cas(14, 0), "sestanek",
    new Cas(18, 30), "večerja"
));

// izpišimo vsebino slovarja

```

```

System.out.println(opravki);
// {8:15=predavanja, 11:10=govorilne ure, 12:30=kosilo, ...}

// izpišimo slovar s sprehodom po množici parov ključ-vrednost
for (Map.Entry<Cas, String> par: opravki.entrySet()) {
    System.out.printf("%s -> %s%n", par.getKey(), par.getValue());
}
// 8:15 -> predavanja
// 11:10 -> govorilne ure
// 12:30 -> kosilo
// 14:00 -> sestanek
// 18:30 -> večerja

```

Razred `TreeMap` implementira vmesnik `NavigableMap`, ki ponuja podobne metode kot vmesnik `NavigableSet`:

```

// koda 10.14 (vsebovalniki/map/navigable/Opravki.java)
NavigableMap<Cas, String> opravki = new TreeMap<>(Map.of(
    new Cas(8, 15), "predavanja",
    new Cas(11, 10), "govorilne ure",
    new Cas(12, 30), "kosilo",
    new Cas(14, 0), "sestanek",
    new Cas(18, 30), "večerja"
));

Cas poldan = new Cas(12, 0);

// prvi popoldanski termin
System.out.println(opravki.higherKey(poldan));    // 12:30

// zadnji dopoldanski par termin-dejavnost
Map.Entry<Cas, String> par = opravki.lowerEntry(poldan);
System.out.println(par.getKey());    // 11:10
System.out.println(par.getValue());  // govorilne ure

// slovar dopoldanskih opravkov
Map<Cas, String> dopoldanski = opravki.headMap(poldan, true);
System.out.println(dopoldanski);
// {8:15=predavanja, 11:10=govorilne ure}

// dodajmo še en dopoldanski opravek
dopoldanski.put(new Cas(6, 10), "tek");

```

```
System.out.println(dopoldanski);
// {6:10=tek, 8:15=predavanja, 11:10=govorilne ure}

System.out.println(opravki);
// {6:10=tek, 8:15=predavanja, 11:10=govorilne ure,
// 12:30=kosilo, 14:00=sestane, 18:30=večerja}
```

Vidimo, da metoda `headMap` ne vrne kopije podslovarja, sestavljenega iz ključev, manjših od objekta `polDan`, in pripadajočih vrednosti, ampak zgolj *pogled* na ta slovar, prek katerega lahko vplivamo na matični slovar. Ko v slovar `dopoldanski` dodamo jutranji tek, se ta doda tudi v slovar `opravki`.

**Naloga 10.18** Kako bi se izpisi v kodi 10.12 spremenili, če bi iz razreda `Cas` odstranili metodi `equals` in `hashCode`?

**Naloga 10.19** Popravite in dopolnite kodo 10.13 tako, da bo slovar urejen po padajočih terminih, a ne spreminjajte naravne urejenosti razreda `Cas`.

**Naloga 10.20** Napišite metodo

```
public static <P, Q, R> Map<P, R> stik(Map<P, Q> prvi,
                                     Map<Q, R> drugi)
```

ki vrne slovar, ki vsak ključ  $p$  iz slovarja `prvi` preslika v vrednost, v katero se v slovarju `drugi` preslika vrednost  $q$ , ki v slovarju `prvi` pripada ključu  $p$ . Če slovar `drugi` nima ključa  $q$ , naj vrnjeni slovar *ne* vsebuje ključa  $p$ .

**Naloga 10.21** Napišite metodo

```
public static Map<String, Map<String, List<Oseba>>>
    poPriimkuInImenu(Collection<Oseba> osebe)
```

ki vrne slovar, v katerem ključu  $p$  pripada slovar, v katerem ključu  $i$  pripada seznam vseh oseb iz podane zbirke, ki jim je ime  $i$ , pišejo pa se  $p$ .

### 10.6.3 Uporaba slovarjev

Slovarji, kot smo že ugotovili, so nekakšne dvoživke, saj jih lahko uporabljamo bodisi kot množice ključev s pripisanimi vrednostmi (oz. množice parov ključ-vrednost) ali pa kot tabele s splošnejšimi indeksi. V razdelku 7.7.3 smo slovar uporabili za vzdrževanje preprostega telefonskega imenika. Na tem mestu pa bomo s pomočjo slovarja nadgradili primer 10.1 v razdelku 10.5.4: besed, ki nastopajo v danem besedilu, ne bomo zgolj našeli, ampak bomo tudi določili njihove pogostosti.



**Primer 10.3.** Napišimo program, ki za podano vhodno zaporedje besed, ki so med seboj ločene s poljubnim nepraznim zaporedjem presledkov, tabulatorjev in prelomov vrstice, izpiše leksikografsko urejen seznam vseh besed skupaj z njihovim številom pojavitev.

*Rešitev.* Namesto množice besed bomo tokrat vzdrževali slovar tipa `Map<String, Integer>`, ki bo podano besedo preslikal v število njenih pojavitev. Ker bodo besede (ključi) pri izpisu leksikografsko urejene, bomo slovar izdelali kot objekt razreda `TreeMap`. Za vsako izluščeno besedo bomo preverili, ali v slovarju že nastopa kot ključ. Če ne, bomo v slovar dodali par (*beseda*, 0). Nato bomo vrednost, ki pripada besedi (torej število njenih pojavitev), v vsakem primeru povečali za 1. Na koncu se bomo le še sprehodili po slovarju in izpisali njegove ključe in vrednosti.

```
// koda 10.15 (vsebovalniki/map/besede/PogostostBesed.java)
import java.util.*;

public class PogostostBesed {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, Integer> slovar = new TreeMap<>();
        while (sc.hasNext()) {
            String beseda = sc.next();
            int pogostost = slovar.containsKey(beseda) ?
                           slovar.get(beseda) : 0;
            slovar.put(beseda, pogostost + 1);
        }
        for (String beseda: slovar.keySet()) {
            System.out.printf("%s: %d%n",
                              beseda, slovar.get(beseda));
        }
    }
}
```

□

**Primer 10.4.** Napišimo program, ki banki omogoča vzdrževanje komitentov in njihovih računov. Vsak komitent lahko ima poljubno mnogo računov. Tako komitenti kot računi so določeni z enoličnimi šiframi. Program bo omogočal dodajanje komitentov, odpiranje računov, transakcije (pologe, dvige in prenose), izpis stanja podanega računa in izpis podatkov o podanem komitentu. Sledi primer vhoda ...

```
nk K326 Janez_Novak    // nov komitent s šifro K326
nk K715 Mojca_Bizjak
nr K326 RA5046         // nov račun komitenta K326 (Janez_Novak)
```

```

nr K715 RA8391
nr K326 RA1940           // Janez_Novak ima sedaj dva računa
+ RA8391 100             // na račun RA8391 položi 100 denarnih enot
- RA1940 30              // z računa RA1940 dvigni 30 denarnih enot
?r RA8391                // izpiši stanje računa RA8391
?r RA1940
-> RA8391 RA5046 40       // prenesi 40 den. enot z RA8391 na RA5046
?k K715                  // izpiši podatke o komitentu K715
?k K326

```

... in pripadajočega izhoda (izhod tvorijo zgolj ukazi, ki se pričnejo z vprašajem):

```

100
-30
Mojca_Bizjak | RA8391: 60
Janez_Novak | RA5046: 40 | RA1940: -30

```

*Rešitev.* Rešitev bomo sestavili iz štirih razredov. Razreda *Racun* in *Komitent* bo sta služila za predstavitev posameznih računov in komitentov, objekt razreda *Banka* bo predstavljal banko, razred *Glavni* pa bo vseboval metodo *main*, ki bo brala in izvrševala vhodne ukaze.

Račun je določen s svojo šifro in trenutnim stanjem. Ko račun ustvarimo, je prazen.

```

// koda 10.16 (vsebovalniki/map/banka/Racun.java)
public class Racun {
    private String sifra;
    private int stanje;

    public Racun(String sifra) {
        this.sifra = sifra;
        this.stanje = 0;
    }
}

```

Vsak komitent ima svojo šifro, ime in seznam bančnih računov. Šifra in ime sta seveda podatka tipa *String*, bančne račune pa bi lahko predstavili z atributom tipa *List<Racun>*, a glede na strukturo ukazov je do računov bistveno bolj smiselno dostopati prek njihovih šifer kot prek njihovih indeksov v seznamu. Račune bomo zato predstavili s slovarjem, ki šifro računa preslika v račun s to šifro.

```

// koda 10.17 (vsebovalniki/map/banka/Komitent.java)
import java.util.*;

```

```

public class Komitent {
    private String sifra;
    private String ime;
    private Map<String, Racun> racuni;

    public Komitent(String sifra, String ime) {
        this.sifra = sifra;
        this.ime = ime;
        this.racuni = new HashMap<>();
    }
}

```

Banka hrani seznam svojih komitentov in računov. Tudi tokrat je smiselno, da so oboji dostopni prek svojih šifer. Slovar komitenti bo tako preslikoval šifre komitentov v pripadajoče komitente, slovar racuni pa šifre računov v pripadajoče račune.

```

// koda 10.18 (vsebovalniki/map/banka/Banka.java)
import java.util.*;

public class Banka {
    private Map<String, Komitent> komitenti;
    private Map<String, Racun> racuni;

    public Banka() {
        this.komitenti = new HashMap<>();
        this.racuni = new HashMap<>();
    }
}

```

Morda se vam zdi slovar racuni odveč. Res je: lahko bi shajali zgolj s slovarjem komitenti, saj je vsak račun povezan z nekim komitentom; če se sprehodimo po računih vsakega posameznega komitenta, bomo na ta način prečesali vse račune. Ker pa se nekateri ukazi nanašajo neposredno na račune, je smiselno, da lahko do računov dostopamo tudi po alternativni poti — direktno prek njihovih šifer. Vsak račun bo potemtakem dosegljiv na dva načina: prek njegovega komitenta in neposredno prek šifre. Ali to pomeni, da bo vsak račun predstavljen z dvema enakima objektoma tipa Racun? Ne: objekt bo en sam, bosta pa nanj kazala dva kazalca.

V metodi main v razredu Glavni beremo, razčlenjujemo in izvajamo vhodne ukaze. Če je sc objekt tipa Scanner, potem s stavkom

```
String vrstica = sc.nextLine().strip();
```

preberemo naslednjo vhodno vrstico in odstranimo morebitne začetne in končne presledke in tabulatorje, s stavkom

```
String[] deli = vrstica.split("\\s+");
```

pa prebrano vrstico razbijemo na posamezne dele, če so ti med seboj ločeni s poljubnim nepraznim zaporedjem presledkov in tabulatorjev. Vsak ukaz se prične z navodilom (nk, nr, +, -, ->, ?r ali ?k), nato pa sledijo argumenti navodila.

```
// koda 10.19 (vsebovalniki/map/banka/Glavni.java)
import java.util.*;

public class Glavni {

    public static void main(String[] args) {
        Banka banka = new Banka();
        Scanner sc = new Scanner(System.in);

        while (sc.hasNextLine()) {
            String vrstica = sc.nextLine().strip();
            String[] deli = vrstica.split("\\s+");

            switch (deli[0]) {
                case "nk":
                    banka.dodajKomitenta(deli[1], deli[2]);
                    break;

                case "nr":
                    banka.dodajRacun(deli[1], deli[2]);
                    break;

                case "+":
                    banka.polog(deli[1], Integer.parseInt(deli[2]));
                    break;

                case "-":
                    banka.dvig(deli[1], Integer.parseInt(deli[2]));
                    break;

                case "->":
                    banka.prenos(deli[1], deli[2],
                                Integer.parseInt(deli[3]));
                    break;
            }
        }
    }
}
```

```

        case "?r": {
            Racun racun = banka.racun(deli[1]);
            System.out.println(racun.vrniStanje());
            break;
        }

        case "?k":
            System.out.println(banka.komitent(deli[1]));
            break;
    }
}
}
}
}

```

Razrede Banka, Racun in Komitent bomo morali še nekoliko dopolniti. Metoda dodajKomitenta v razredu Banka ustvari komitenta s šifro sifra in imenom ime ter ga pod ključem sifra doda v slovar komitenti:

```

public class Banka {
    ...
    public void dodajKomitenta(String sifra, String ime) {
        this.komitenti.put(sifra, new Komitent(sifra, ime));
    }
}

```

Metoda dodajRacun za komitenta s šifro sifraKomitenta ustvari račun s šifro sifraRacuna in ga doda tako med komitentove račune kot med račune v slovarju racuni.

```

public class Banka {
    ...
    public void dodajRacun(String sifraKomitenta,
                           String sifraRacuna) {
        Racun racun = new Racun(sifraRacuna);
        this.komitenti.get(sifraKomitenta).dodajRacun(racun);
        this.racuni.put(sifraRacuna, racun);
    }
}

```

Komitentova metoda dodajRacun doda podani račun v slovar racuni, pri čemer kot ključ uporabi šifro računa.

```

public class Komitent {

```

```

    ...
    public void dodajRacun(Racun racun) {
        this.racuni.put(racun.vrniSifro(), racun);
    }
}

public class Racun {
    ...
    public String vrniSifro() {
        return this.sifra;
    }
}

```

Metodi polog in dvig v razredu Banka sprejmeta šifro računa in znesek, ki ga želimo položiti oziroma dvigniti, metodi prenos pa moramo poleg zneska podati šifri izvirnega in ciljnega računa. Vse tri metode pridobijo ustrezne račune iz slovarja racuni.

```

public class Banka {
    ...
    public void polog(String sifraRacuna, int znesek) {
        this.racuni.get(sifraRacuna).dodaj(znesek);
    }

    public void dvig(String sifraRacuna, int znesek) {
        this.racuni.get(sifraRacuna).dodaj(-znesek);
    }

    public void prenos(String sifraIzvor, String sifraCilj,
                       int znesek) {
        Racun izvor = this.racuni.get(sifraIzvor);
        Racun cilj = this.racuni.get(sifraCilj);
        izvor.dodaj(-znesek);
        cilj.dodaj(znesek);
    }
}

public class Racun {
    ...
    public void dodaj(int znesek) {
        this.stanje += znesek;
    }
}

```

```
}
```

Metodi `racun` in `komitent` zgolj vrneta račun oziroma komitenta s podano šifro:

```
public class Banka {
    ...
    public Racun racun(String sifraRacuna) {
        return this.racuni.get(sifraRacuna);
    }

    public Komitent komitent(String sifraKomitenta) {
        return this.komitenti.get(sifraKomitenta);
    }
}
```

Metoda `main` v razredu `Glavni` vsebuje sledeči stavek:

```
System.out.println(banka.komitent(deli[1]));
```

Ker ta stavek nad pridobljenim komitentom pokliče metodo `toString`, jo moramo v razredu `Komitent` ustrezno redefinirati. Metoda mora torej vrniti niz, ki vsebuje ime komitenta in podatke o njegovih računih.

```
public class Komitent {
    ...
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(this.ime);
        for (Racun racun: this.racuni.values()) {
            sb.append(" | " + racun.toString());
        }
        return sb.toString();
    }
}
```

Manjkata nam le še metodi `vrniStanje` in `toString` v razredu `Racun`:

```
public class Racun {
    ...
    public int vrniStanje() {
        return this.stanje;
    }

    @Override
```

```

    public String toString() {
        return String.format("%s: %s", this.sifra, this.stanje);
    }
}

```

□

**Naloga 10.22** Popravite program `PogostostBesed` tako, da bo besede uredil po padajočem številu pojavitev, besede z enakim številom pojavitev pa po abecedi.

**Naloga 10.23** Napišite program, ki za vsako vhodno besedo  $b$  izpiše množico vseh kontekstov, v katerih nastopa. Kontekst besede  $b$  je zaporedje, sestavljeno iz  $k$  besed pred besedo  $b$ , besede  $b$  in  $k$  besed za besedo  $b$ . Na primer, pri  $k = 2$  in besedilu

dan es je lep dan danes grem na izlet danes bo vse super

so konteksti besede danes takšni: (1) danes je lep; (2) lep dan danes grem na; (3) na izlet danes bo vse.

V prvi vrstici vhoda je navedeno število  $k$ , nato pa sledi besedilo v enaki obliki kot v primeru 10.1. Elemente množice (kontekste) izpišite po abecedi.

**Naloga 10.24** Bi lahko metodo `dodajRacun` v razredu `Banka` ...

```

public void dodajRacun(String sifraKomitenta, String sifraRacuna) {
    Racun racun = new Racun(sifraRacuna);
    this.komitenti.get(sifraKomitenta).dodajRacun(racun);
    this.racuni.put(sifraRacuna, racun);
}

```

... napisali tudi takole?

```

public void dodajRacun(String sifraKomitenta, String sifraRacuna) {
    this.komitenti.get(sifraKomitenta).
        dodajRacun(new Racun(sifraRacuna));
    this.racuni.put(sifraRacuna, new Racun(sifraRacuna));
}

```

**Naloga 10.25** Popravite program za vzdrževanje bančnih komitentov in računov tako, da bo javil napako, če poskusimo dodati komitenta ali račun s šifro, ki že obstaja, ali če želimo dostopati do komitenta ali računa z neobstoječo šifro.

**Naloga 10.26** Napišite program, ki simulira delovanje knjižnice. Nekaj iztočnic: knjižni naslovi, izvodi (knjižnica lahko hrani več izvodov istega naslova, npr. deset izvodov Cankarjevih Hlapcev), člani, izposoje ...



## 10.7 Razred Collections

Razred Collections ponuja različne generične statične metode za delo z vsebovalniki. Namesto da bi suhoparno prepisovali javansko dokumentacijo, si jih raje nekaj oglejmo na primeru.

```
// koda 10.20 (vsebovalniki/collections/PrimeriCollections.java)
import java.util.*;

public class PrimeriCollections {

    private static class PrimerjalnikPoDolzini
        implements Comparator<String> {
        @Override
        public int compare(String prvi, String drugi) {
            return prvi.length() - drugi.length();
        }
    }

    public static void main(String[] args) {
        List<String> imena = new ArrayList<>(List.of(
            "Cvetka", "Danijel", "Janko", "Bernarda", "Ernest",
            "Hinko", "Anka", "Filip", "Iva", "Gabrijela"
        ));

        // največji element glede na naravno urejenost
        System.out.println(Collections.max(imena)); // Janko

        // najmanjši element glede na podani primerjalnik
        System.out.println(Collections.min(
            imena, new PrimerjalnikPoDolzini())); // Iva

        // ureditev seznama glede na naravno urejenost
        Collections.sort(imena);
        System.out.println(imena);
        // [Anka, Bernarda, Cvetka, Danijel, Ernest, Filip,
        // Gabrijela, Hinko, Iva, Janko]

        // iskanje v urejenem seznamu
        System.out.println(Collections.binarySearch(imena, "Filip"));
        // 5
    }
}
```

```

System.out.println(Collections.binarySearch(imena, "Irena"));
// -9
// (Irena bi v urejenem seznamu sodila na indeks
// -(-9) - 1 = 8)

// medsebojna zamenjava elementov
Collections.swap(imena, 1, 4);
System.out.println(imena);
// [Anka, Ernest, Cvetka, Danijel, Bernarda, Filip,
// Gabrijela, Hinko, Iva, Janko]

// naključno mešanje
Collections.shuffle(imena, new Random(12345));
System.out.println(imena);
// [Danijel, Cvetka, Anka, Filip, Iva, Janko,
// Gabrijela, Hinko, Bernarda, Ernest]

List<Integer> stevila = new ArrayList<>(List.of(
    50, 20, 40, 30, 20, 10, 50, 30, 20, 30));

// število pojavitev
System.out.println(Collections.frequency(stevila, 20));
// 3

// indeks podseznama
System.out.println(Collections.indexOfSubList(
    stevila, List.of(50, 30, 20))); // 6

// zamenjava vseh pojavitev elementa
Collections.replaceAll(stevila, 30, -1);
System.out.println(stevila);
// [50, 20, 40, -1, 20, 10, 50, -1, 20, -1]

// true natanko tedaj, ko podani zbirk
// nimata skupnih elementov
System.out.println(Collections.disjoint(
    stevila, Set.of(3, 30, 300))); // true

// zamenjava vseh elementov z istim elementom
Collections.fill(stevila, 42);
System.out.println(stevila);
// [42, 42, 42, 42, 42, 42, 42, 42, 42, 42]

```

```

    // seznam, sestavljen iz podanega števila kopij
    // podanega elementa
    List<Character> polja = new ArrayList<>(
                                Collections.nCopies(5, '_'));
    System.out.println(polja);    // [-, -, -, -, -]
}
}

```

## 10.8 Povzetek

- Vsebovalniki so objekti, namenjeni hranjenju poljubnega števila elementov.
- Javanske vsebovalnike delimo na zbirke in slovarje. Elementi v zbirkah nastopajo samostojno, v slovarjih pa kot pari ključev in pripadajočih vrednosti.
- Razredi in vmesniki za predstavitev vsebovalnikov so organizirani v hierarhijo, na čelu katere sta vmesnika `Collection` (za zbirke) in `Map` (za slovarje). Vsi razredi in vmesniki v tej hierarhiji so generični.
- Zbirke delimo na sezname in množice. V seznamih so položaji elementov določeni z indeksi, pri množicah pa vrstni red elementov v splošnem ni določen (in tudi če je, do elementov ni mogoče dostopati prek indeksov). Poleg tega se elementi v seznamih lahko podvajajo, pri množicah pa to ni mogoče.
- Vmesnik `List` je izhodišče hierarhije razredov in vmesnikov za predstavitev seznamov. Med razrede, ki implementirajo vmesnik `List`, sodita razreda `ArrayList` in `LinkedList`. V objektih razreda `ArrayList` je seznam implementiran z »raztegljivo«  
tabelo, v objektih razreda `LinkedList` pa s povezanim seznamom. Seznami tipa `ArrayList` so zato učinkoviti za dostop do elementa prek podanega indeksa, seznami tipa `LinkedList` pa za vstavljanje in odstranjevanje elementa na poljubnem položaju.
- Vmesnik `Set` je izhodišče hierarhije razredov in vmesnikov za predstavitev množic. Med razrede, ki implementirajo vmesnik `Set`, sodita razreda `HashSet` in `TreeSet`. V objektih razreda `HashSet` je množica implementirana z zgoščeno tabelo, v objektih razreda `TreeSet` pa z dvojiškim iskalnim drevesom. Množice tipa `TreeSet` so urejene glede na naravno urejenost ali glede na podani primerjalnik. Enakost elementov se pri množicah tipa `HashSet` preverja z metodo `equals`, pri množicah tipa `TreeSet` pa z metodo `compareTo` ali `compare`. Razred `TreeSet` implementira vmesnik `NavigableSet`, ki ponuja metode za pridobivanje pogledov na podmnožice, ki vsebujejo podani interval elementov izhodiščne množice.

- Hierarhija razredov in vmesnikov za predstavitev slovarjev, ki izhaja iz vmesnika `Map`, je povsem analogna hierarhiji, ki izhaja iz vmesnika `Set`, saj lahko slovarje obravnavamo kot množice ključev s pripisanimi vrednostmi. Ključi v slovarjih se namreč obravnavajo na enak način kot elementi v množicah.
- Razred `Collections` vsebuje različne generične statične metode za delo z vsebovalniki.

### ***Iz profesorjevega kabineta***

»01101000100000001000000000000001...«

»Kaj predstavljajo te enke in ničle, profesor Doberšek?«

»Karakteristični vektor množice  $\{2^k \mid k \geq 0\}$ .«

»Karakterni vektor?«

»Karakteristični, Jože! Če imamo množico z elementi  $a_1, a_2, \dots, a_n$ , ima njen karakteristični vektor enice na indeksih  $a_1, a_2, \dots, a_n$ , na vseh ostalih pa ničle.«

»Zanimivo! Hm ... bi morda lahko s karakterističnim vektorjem predstavili tudi množico v javi?«

»Seveda,« se vmeša znani glas, »ta predstavitev je izjemno učinkovita, če so tvoji elementi cela števila z intervala  $[0, M]$ , kjer  $M$  ni prevelik. Če je  $M = 63$ , pa je sploh imenitno, saj lahko celotno množico predstaviš z enim samim številom tipa `long`.«

»Kako, prosim?!«

»Recimo, množico  $\{7, 13, 50\}$  predstaviš s številom, ki ima v dvojiškem zapisu enice na položajih 7, 13 in 50, torej s številom

```
(1L << 7) | (1L << 13) | (1L << 50)
```

To je, mimogrede, ravno zrcalna slika karakterističnega vektorja te množice. Vse operacije nad množico lahko sedaj implementiraš z bitnimi operacijami, ki so, kot najbrž veš, izjemno hitre. Dodajanje je bitni *ali*, odzemanje je bitni *in* ...«

»Genovefa, ne mi vsega izdati! Me že srbijo prsti!«

Napišite razred za predstavitev podmnožic množice  $\{0, 1, \dots, 63\}$ . Razred naj implementira vmesnik `Set<Integer>`. (Če boste razširili razred `AbstractSet<Integer>`, boste imeli precej manj dela, kot če bi neposredno implementirali vmesnik `Set<Integer>`.)

## 11 Lambde

Konstrukti, ki jih imenujemo *lambde*, nam omogočajo, da objekte vmesnikov z eno samo abstraktno metodo (pravimo jim *funkcijski vmesniki*) izdelujemo z bistveno manj kode, kot je bilo potrebno pred njihovo uvedbo. Namesto da bi najprej izdelali poseben implementacijski razred z implementacijo abstraktne metode in nato še objekt tega razreda, pri uporabi `lambd` navedemo samo najpomembnejše: telo implementacije abstraktne metode. Za vse ostalo poskrbi prevajalnik. Lambde torej ne ponujajo nič takega, česar z že znanimi tehnikami ne bi mogli sprogramirati, kljub temu pa so vredne svojega poglavja, saj marsikdaj vodijo do bistveno krajše in preglednejše kode.

### 11.1 Od samostojnega do anonimnega razreda

Recimo, da želimo seznam objektov razreda `Oseba` (razdelek 9.2, koda 9.3) urediti po spolu in starosti, ne da bi spreminjali naravno urejenost tega razreda. Željo lahko uresničimo s kodo

```
List<Oseba> osebe = new ArrayList<>(List.of(...));
osebe.sort(Oseba.primerjalnikSpolStarost());
```

pri čemer metoda `primerjalnikSpolStarost` vrne objekt tipa `Comparator<Oseba>`, ki predstavlja ciljno urejenost. Sedaj že dobro vemo, da potrebujemo razred, ki ta vmesnik implementira. Prav tako vemo, da lahko ta razred napišemo kot

- samostojni razred:

```
// koda 11.1 (lambde/uvod/samostojni/*.java)
public class PrimerjalnikSpolStarost implements
    Comparator<Oseba> {
    @Override
    public int compare(Oseba prva, Oseba druga) {
        if (prva.vrniSpol() != druga.vrniSpol()) {
            return druga.vrniSpol() - prva.vrniSpol();
        }
        return prva.vrniLetoRojstva() - druga.vrniLetoRojstva();
    }
}
```

```

    }
}

public class Oseba {
    ...
    public static Comparator<Oseba> primerjalnikSpolStarost() {
        return new PrimerjalnikSpolStarost();
    }
}

```

- statični notranji razred:

```

// koda 11.2 (lambde/uvod/staticniNotranji/*.java)
public class Oseba {
    ...
    private static class PrimerjalnikSpolStarost
        implements Comparator<Oseba> {
        @Override
        public int compare(Oseba prva, Oseba druga) {
            if (prva.spol != druga.spol) {
                return druga.spol - prva.spol;
            }
            return prva.letoSrojstva - druga.letoSrojstva;
        }
    }

    public static Comparator<Oseba> primerjalnikSpolStarost() {
        return new PrimerjalnikSpolStarost();
    }
}

```

- anonimni notranji razred:

```

// koda 11.3 (lambde/uvod/anonimni/*.java)
public class Oseba {
    ...
    public static Comparator<Oseba> primerjalnikSpolStarost() {
        return new Comparator<Oseba>() {
            @Override
            public int compare(Oseba prva, Oseba druga) {
                if (prva.spol != druga.spol) {
                    return druga.spol - prva.spol;
                }
            }
        };
    }
}

```

```

    }
    return prva.letoSrojstva - druga.letoSrojstva;
}
}
};
}

```

Rešitev z anonimnim notranjim razredom je med predstavljenimi najkrajša, a še vedno ni tako jedrnata in elegantna, kot bi lahko bila. Še vedno izdelamo razred, ki ga potrebujemo zgolj za en namen, in še vedno imamo precej kode, ki ne prispeva ničesar k razumevanju metode, ampak tam pač mora biti zaradi sintaktičnih zahtev jezika.

**Naloga 11.1** Je anonimni notranji razred v kodi 11.3 statičen ali nestatičen?

**Naloga 11.2** Na vprašanje, ali bi lahko razred PrimerjalnikSpolStarost zapisali kot nestatični notranji razred, se odgovor glasi »da, a to ni smiselno«. Zakaj?

## 11.2 Lambda

Od javine različice 8 naprej lahko kodo v primerih, ko potrebujemo objekt vmesnika z eno samo abstraktno metodo, skróčimo na tisto, kar je res pomembno — telo implementacije abstraktne metode. V našem primeru se torej osredotočimo na telo metode `compare`:

```

// koda 11.4 (lambde/uvod/lambda/*.java)
public class Oseba {
    ...
    public static Comparator<Oseba> primerjalnikSpolStarost() {
        return (Oseba prva, Oseba druga) -> {
            if (prva.spol != druga.spol) {
                return druga.spol - prva.spol;
            }
            return prva.letoSrojstva - druga.letoSrojstva;
        };
    }
}

```

Odsek

```

(Oseba prva, Oseba druga) -> {
    if (prva.spol != druga.spol) {

```

```

        return druga.spol - prva.spol;
    }
    return prva.letoSjajstva - druga.letoSjajstva;
}

```

imenujemo *lambda*. Lambda si lahko predstavljamo kot anonimno metodo. Tako kot običajna metoda ima namreč svojo *glavo* (seznam deklaracij parametrov) in *telo*:

```

( $T_1$   $p_1$ ,  $T_2$   $p_2$ , ...) -> {
    telo metode
}

```

Ker naša lambda implementira metodo `compare` v okviru vmesnika `Comparator<Oseba>`, sprejme dva parametra tipa `Oseba` in vrne nekaj negativnega, če prva oseba po primerjalnem kriteriju sodi pred drugo, 0, če se osebi po primerjalnem kriteriju ne razlikujeta, in nekaj pozitivnega, če prva oseba po primerjalnem kriteriju sodi za drugo.

Ampak — kako prevajalnik ve, da z lambda implementiramo metodo `compare` v okviru vmesnika `Comparator<Oseba>`? Ali lahko lambda uporabimo tudi za kak drug vmesnik? Kaj pravzaprav sploh je lambda? Če gre za poseben izraz, kakšna je njegova vrednost in kakšen je njegov tip? Na ta vprašanja bomo poskusili odgovoriti v naslednjem razdelku.

### 11.3 Vrednost in tip lambde

Prvič, lambda lahko uporabimo samo takrat, ko potrebujemo objekt vmesnika. Drugič, ta vmesnik ne more biti kakršenkoli, ampak mora imeti *natanko eno abstraktno metodo*. Takemu vmesniku v javanski terminologiji pravimo *funkcijski vmesnik*. Vmesniki `Comparable`, `Comparator` in `Iterable` so torej funkcijski vmesniki, saj imajo vsi trije natanko po eno abstraktno metodo. Vmesnik `Iterator` pa ima dve abstraktni metodi (`hasNext` in `next`), zato *ni* funkcijski vmesnik.

Lambda je *izraz*. Njena *vrednost* je kazalec na objekt vmesnika, njen *tip* (tj. vmesnik, ki mu pripada) pa se določi iz *konteksta*. Lambda namreč vedno nastopa v okviru nečesa, iz česar lahko prevajalnik izlušči njen tip.

Definirajmo funkcijska vmesnika `Predikat<T>` in `DvojiskaFunkcija<T, U, R>`. Objekt prvega predstavlja funkcijo s parametrom tipa `T` in rezultatom tipa `boolean`, objekt drugega pa funkcijo s parametroma tipa `T` in `U` in rezultatom tipa `R`:

```

// koda 11.5 (lambda/uvod/osnovniPrimeri/*.java)
public interface Predikat<T> {
    public abstract boolean preveri(T a);
}

```



```
public interface DvojiskaFunkcija<T, U, R> {
    public abstract R izvedi(T a, U b);
}
```

Oglejmo si sledeči stavek:

```
Predikat<String> dolg = (String a) -> { return a.length() >= 10; };
```

Prevajalnik na podlagi leve strani stavka ve, da je tip lambde na desni strani enak `Predikat<String>`. Tipni parameter `T` se očitno nadomesti s tipom `String`. Vrednost lambde (in s tem celotne desne strani) je potemtakem kazalec na objekt (nevidnega) razreda, ki implementira vmesnik `Predikat<String>`, pri čemer je metoda `preveri` implementirana takole:

```
public boolean preveri(String a) {
    return a.length() >= 10;
}
```

Sledeči stavek je enakovreden gornjemu, le da se namesto lambde poslužuje anonimnega notranjega razreda:

```
Predikat<String> dolg = new Predikat<>() {
    @Override
    public boolean preveri(String a) {
        return a.length() >= 10;
    }
};
```

V primeru stavka

```
DvojiskaFunkcija<String, Integer, Character> izlusciZnak =
    (String niz, Integer indeks) -> { return niz.charAt(indeks); };
```

prevajalnik ugotovi, da tip lambde na desni strani ne more biti nič drugega kot `DvojiskaFunkcija<String, Integer, Character>`. Tipni parametri `T`, `U` in `R` se torej zamenjajo s tipi `String`, `Integer` in `Character` (v tem vrstnem redu), lambda pa je objekt nevidnega razreda, ki implementira vmesnik `DvojiskaFunkcija<String, Integer, Character>` ter s tem metodo `izvedi` s parametroma tipa `String` in `Integer` in izhodnim tipom `Character`:

```
public Character izvedi(String niz, Integer indeks) {
    return niz.charAt(indeks);
}
```

Ker imena parametrov niso pomembna, smo splošni imeni `a` in `b` zamenjali z bolj smiselnima `niz` in `indeks`.

Lambde lahko nastopajo v kateremkoli kontekstu, v katerem je mogoče določiti njen tip. Ker lambda v sledečem primeru nastopa kot rezultat metode, ki vrača vrednost tipa `Predikat<Integer>` ...

```
public static Predikat<Integer> jeDeljivZ(int delitelj) {
    return (Integer stevilo) -> { return stevilo % delitelj == 0; };
}
```

... prevajalnik ve, da je njen tip enak `Predikat<Integer>` in da implementira metodo preveri s parametrom tipa `Integer`:

```
public boolean preveri(Integer stevilo) {
    return stevilo % delitelj == 0;
}
```

Oglejmo si še en primer. Definirajmo sledečo metodo:

```
public static <T> T poisciPrvega(List<T> seznam,
                                Predikat<T> predikat) {
    for (T element: seznam) {
        if (predikat.preveri(element)) {
            return element;
        }
    }
    return null;
}
```

V kodi

```
// koda 11.6
List<Cas> casi = List.of(new Cas(10, 20), new Cas(7, 30),
                        new Cas(15, 10), new Cas(13, 40));
System.out.println(poisciPrvega(
    casi,
    (Cas cas) -> { return cas.compareTo(new Cas(12, 0)) > 0; }
));
```

je prvi argument pri klicu funkcije `poisciPrvega` tipa `List<Cas>`. Tipni parameter `T` v funkciji `poisciPrvega` se potemtakem nadomesti s tipnim argumentom `Cas`, zato prevajalnik ve, da mora biti tip drugega argumenta (lambda) enak `Predikat<Cas>`. To pomeni, da lambda implementira sledečo metodo:

```
public boolean preveri(Cas cas) {
    return cas.compareTo(new Cas(12, 0)) > 0;
}
```

Koda 11.6 izpiše 15:10, saj gre za prvi objekt tipa `Cas`, ki po naravni urejenosti sodi za trenutek 12:00.

Če ne bi bilo `lambd`, bi metodi lahko podali kvečjemu objekt anonimnega razreda:

```
List<Cas> casi = List.of(
    new Cas(10, 20), new Cas(7, 30), new Cas(15, 10), new Cas(13, 40)
);
System.out.println(poisciPrvega(
    casi,
    new Predikat<Cas>() {
        @Override
        public boolean preveri(Cas cas) {
            return cas.compareTo(new Cas(12, 0)) > 0;
        }
    }
));
```

Razlika je očitna ...

**Naloga 11.3** Sledeči klic metode `sort` dopolnite tako, da bo pare uredil primarno po naraščajoči drugi komponenti, sekundarno pa po padajoči prvi komponenti. Nalogo rešite na tri načine: z nestatičnim notranjim razredom, z anonimnim notranjim razredom in z `lambdo`.

```
List<Par<String, Integer>> pari = new ArrayList<>(List.of(...));
pari.sort(...);
```

**Naloga 11.4** Sledeča koda ustvari dva ločena vzporedna podprocesa (*niti*):

```
public static void main(String[] args) {
    ustvariNit(10, "prva", 100);
    ustvariNit(5, "druga", 200);
}

public static void ustvariNit(int n, String niz, int premor) {
    new Thread(() -> {
        for (int i = 0; i < n; i++) {
            System.out.println(niz);
            try { Thread.sleep(premor); }
            catch (InterruptedException ex) {}
        }
    }).start();
}
```

Lambda v metodi `ustvariNit` zamenjajte z ustreznim anonimnim notranjim razredom.

**Naloga 11.5** Po zgledu vmesnika `DvojiskaFunkcija<T, U, R>` napišite vmesnik `EniskaFunkcija<T, R>`, nato pa napišite metodo

```
public static <T, U, R> EniskaFunkcija<T, R>
    fiksirajDrugega(DvojiskaFunkcija<T, U, R> f, U vrednost)
```

ki vrne eniško funkcijo, ki deluje tako, kot bi delovala podana dvojiška funkcija, če bi bil njen drugi parameter fiksiran na podano vrednost.

## 11.4 Nadaljnje poenostavitve

Kot smo videli, so lambde v primerjavi s »tradicionalnimi« tehniki bistveno kompaktnejša možnost za izdelavo objektov funkcijskih vmesnikov. Kljub temu pa jih v nekaterih primerih lahko še nadalje poenostavimo.

Prvič, če telo lambde vsebuje zgolj stavek `return ...`

```
(T1 p1, T2 p2, ...) -> {
    return izraz;
}
```

... potem lahko par zaviti oklepajev, besedo `return` in podpičje preprosto izpustimo:

```
(T1 p1, T2 p2, ...) -> izraz
```

Na primer, namesto

```
(Oseba prva, Oseba druga) -> {
    return prva.priimek.compareTo(druga.priimek);
}
```

lahko pišemo

```
(Oseba prva, Oseba druga) -> prva.priimek.compareTo(druga.priimek)
```

Druga poenostavitev se nanaša na tipe parametrov lambde. Kadar jih prevajalnik lahko izlušči sam, jih lahko izpustimo. Na primer, pri stavku

```
Comparator<Oseba> poPriimku =
    (Oseba prva, Oseba druga) ->
        prva.priimek.compareTo(druga.priimek);
```

prevajalnik na podlagi tipa na levi strani (`Comparator<Oseba>`) ugotovi, da je tip lambde enak `Comparator<Oseba>`, zato morata biti parametra metode `compare`, ki jo lambda implementira, prav tako tipa `Oseba`. To pomeni, da nam tipov `Oseba` v glavi lambde ni treba navesti:

```
Comparator<Oseba> poPriimku =
    (prva, druga) -> prva.priimek.compareTo(druga.priimek);
```

**Naloga 11.6** Lambde v razdelku 11.3 karseda poenostavite.

**Naloga 11.7** Napišite metodo

```
public static <T> T drugiNajvecji(
    Collection<T> zbirka, Comparator<T> primerjalnik)
```

ki vrne drugi največji element zbirke glede na podani primerjalnik, in jo pokličite tako, da bo pri podani zbirki objektov tipa `Oseba` vrnila osebo z drugo največjo vsoto dolžin imena in priimka.

## 11.5 Vgrajeni funkcijski vmesniki

Nevede smo spoznali že tri funkcijske vmesnike: `Comparable`, `Comparator` in `Iterable`. Še več jih najdemo v paketu `java.util.function`. Vmesniki v tem paketu ponujajo abstraktne metode z različnimi sezname tipov parametrov in izhodnimi tipi. Nekaj jih je skupaj z njihovimi abstraktnimi metodami naštetih v tabeli 11.1.

**Tabela 11.1** Nekateri funkcijski vmesniki v paketu `java.util.function`.

Vmesnik	Abstraktna metoda
<code>Consumer&lt;T&gt;</code>	<code>public abstract void accept(T t)</code>
<code>Supplier&lt;T&gt;</code>	<code>public abstract T get()</code>
<code>Function&lt;T, R&gt;</code>	<code>public abstract R apply(T t)</code>
<code>Predicate&lt;T&gt;</code>	<code>public abstract boolean test(T t)</code>
<code>UnaryOperator&lt;T&gt;</code>	<code>public abstract T apply(T t)</code>
<code>BiConsumer&lt;T, U&gt;</code>	<code>public abstract void accept(T t, U u)</code>
<code>BiFunction&lt;T, U, R&gt;</code>	<code>public abstract R apply(T t, U u)</code>
<code>BiPredicate&lt;T, U&gt;</code>	<code>public abstract boolean test(T t, U u)</code>
<code>BinaryOperator&lt;T&gt;</code>	<code>public abstract T apply(T t, T u)</code>

Če pobrskamo po paketu `java.util.function`, bomo našli tudi vmesnike, pri katerih imamo namesto enega ali več tipnih parametrov primitivne tipe `int`, `long`

ali double. Na primer, vmesnik `IntPredicate` ponuja metodo

```
public abstract boolean test(int value)
```

vmesnik `DoubleToIntFunction` ponuja metodo

```
public abstract int applyAsInt(double value)
```

vmesnik `LongConsumer` pa ponuja metodo

```
public abstract void accept(long value)
```

**Primer 11.1.** Napišimo metodo `tabelaOperacije`, ki izpiše tabelo dvojiške operacije *op* za števila od 1 do *n*. Element tabele v *i*-ti vrstici in *j*-tem stolpcu naj podaja vrednost *i op j*. Na primer, za operacijo množenja in *n* = 5 bi dobili tak izpis:

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

*Rešitev.* Naša metoda bo sprejela dva parametra: eden od njih je seveda *n*, z drugim pa moramo predstaviti dvojiško operacijo. Dvojiško operacijo bi v nekaterih programskih jezikih lahko predstavili s funkcijo, ki sprejme dva parametra tipa `int` in vrne rezultat istega tipa, v javi pa metode žal ne moremo posredovati kot parameter drugi metodi. Lahko pa operacijo predstavimo kot objekt funkcijskega vmesnika, ki ponuja tako metodo. Tak funkcijski vmesnik lahko napišemo sami, lahko pa ga poskusimo poiskati v paketu `java.util.function`. Z malo truda bomo odkrili vmesnik `IntBinaryOperator`, ki ponuja sledečo abstraktno metodo:

```
public abstract int applyAsInt(int left, int right)
```

Metodo za izpis tabele dvojiške operacije bomo torej deklarirali takole:

```
public static void tabelaOperacije(int n, IntBinaryOperator op)
```

Tabelo izpišemo s preprosto dvojno zanko. Za vsak par števil *i* in *j* pokličemo metodo `op.applyAsInt`.

```
// koda 11.7 (lambde/tabelaOperacije/TabelaOperacije.java)
public static void tabelaOperacije(int n, IntBinaryOperator op) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            System.out.printf(" %3d", op.applyAsInt(i, j));
        }
    }
}
```

```

    }
    System.out.println();
}

```

Metodi `tabelaOperacije` lahko posredujemo objekt anonimnega notranjega razreda ...

```

public static void main(String[] args) {
    tabelaOperacije(5, new IntBinaryOperator() {
        @Override
        public int applyAsInt(int a, int b) {
            return a * b;
        }
    });
}

```

... lambde pa nam omogočajo bistveno jedrnatejšo izdelavo objekta:

```

public static void main(String[] args) {
    tabelaOperacije(5, (int a, int b) -> a * b);
}

```

Ker se morata tipa parametrov lambde ujemati s tipoma parametrov edine abstraktne metode vmesnika `IntBinaryOperator`, prevajalnik sam ugotovi, da morata biti oba `int`, zato ju lahko izpustimo:

```

public static void main(String[] args) {
    tabelaOperacije(5, (a, b) -> a * b);
}

```

Si lahko zamislimo še elegantnejšo rešitev? □

### Naloga 11.8 Napišite metodo

```

public static <T> void izpisi(List<T> seznam,
                             BiPredicate<T, T> predikat)

```

ki izpiše vse pare elementov seznama, za katere je podani predikat resničen. (Predikat je za podana objekta resničen natanko tedaj, ko metoda `test` vrne `true`, če ji ju podamo kot parametra.)

### Naloga 11.9 Napišite metodo

```

public static <T, U, R> List<R> preslikaj(
    List<Par<T, U>> pari, BiFunction<T, U, R> funkcija)

```

ki vrne seznam, sestavljen iz rezultatov uporabe podane funkcije na elementih podanega seznama.

**Naloga 11.10** Napišite metodo

```
public static <T> Predicate<T> nasprotje(Predicate<T> predikat)
```

ki vrne predikat, ki deluje ravno nasprotno od podanega predikata: če je podani predikat za nek objekt resničen, je vrnjeni predikat neresničen, in obratno.

**Naloga 11.11** Napišite metodo

```
public static IntSupplier fibgen()
```

ki vrne objekt, ki nam da Fibonaccijevo zaporedje, če nad njim zaporedoma kličemo metodo `getAsInt`.

## 11.6 Lambde in lokalne spremenljivke

Ali lahko v telesu lambde uporabljamo tudi spremenljivke, definirane izven lambde? Odgovor je povsem enak kot pri anonimnih notranjih razredih: da, a le tiste, ki so deklarirane z določilom `final` (in so torej konstante), in tiste, ki jih uporabljamo, kot da bi bile konstante (nastavimo jih natanko enkrat, potem pa jih več ne spreminjamo). Na primer, sledeči program se bo brezhibno prevedel:

```
// koda 11.8 (lambde/lokalneSpremenljivke/LokalneSpremenljivke.java)
import java.util.function.*;

public class LokalneSpremenljivke {
    public static void main(String[] args) {
        IntUnaryOperator krat5 = mnozilnik(5);
        System.out.println(krat5.applyAsInt(3)); // 15
        System.out.println(krat5.applyAsInt(10)); // 50
    }

    public static IntUnaryOperator mnozilnik(int faktor) {
        // spremenljivke faktor ne smemo spreminjati!
        return n -> n * faktor;
    }
}
```



**Naloga 11.12** Metodo množilnik prepisite na dva načina: (1) tako, da bo vrnila objekt anonimnega notranjega razreda; (2) tako, da bo vrnila objekt nestatičnega notranjega razreda.

**Naloga 11.13** Napišite metodo

```
public static <T extends Comparable<T>> Predicate<T> pred(T meja)
```

ki vrne predikat, ki je resničen natanko tedaj, ko podani objekt sodi pred podano mejo.

## 11.7 Lambde in zbirke

Lambde nam pogosto pridejo prav pri klicih metod, ki se sprehajajo po podani zbirki in sproti obdelujejo njene elemente. Tipični primeri takih metod so:

- metoda, ki prešteje elemente, ki izpolnjujejo podani pogoj;
- metoda, ki določeno opravilo izvrši za vsak element zbirke;
- metoda, ki vrne rezultat zaporednega združevanja elementov zbirke z dvojiškim operatorjem;
- metoda, ki elemente zbirke razporedi v skupine glede na rezultate podane metode.

Takšne metode najdemo tudi v javini standardni knjižnici, vendar pa jih je mogoče uporabljati le nad tokovi (objekti tipa `java.util.stream.Stream`), ki jih v tej knjigi ne obravnavamo.

V nadaljevanju bomo napisali vse štiri metode. Metode bodo generične, da jih bomo lahko uporabili na zbirkah z elementi poljubnega tipa.

### 11.7.1 Štetje elementov, ki izpolnjujejo pogoj

Ta metoda sprejme zbirko in pogoj, vrne pa število elementov zbirke, ki izpolnjujejo pogoj. Deklarirali jo bomo takole:

```
public static <T> int prestej(Collection<T> zbirka, pogoj)
```

Kako predstavimo *pogoj*? Z objektom funkcijskega vmesnika, kakopak. Abstraktna metoda, ki jo vmesnik ponuja, mora sprejeti objekt tipa `T`, vrniti pa mora vrednost tipa `boolean` (`true`, če je pogoj izpolnjen, in `false`, če ni). Tak vmesnik zlahka napišemo sami ...

```
public interface Pogoj<T> {
    public abstract <T> boolean preveri(T t);
}
```

... lahko pa pokukamo v paket `java.util.function` in uporabimo vmesnik `Predicate`, ki ponuja točno takšno abstraktno metodo, le da se imenuje `test`.

Metoda `prestej` se enostavno sprehodi po elementih podane zbirke in prešteje tiste, za katere je podani predikat resničen (metoda `test` vrne `true`). Spomnimo se, da je vmesnik `Collection` podvmesnik vmesnika `Iterable`, zato se je čez vsako zbirko mogoče sprehoditi z zanko *for-each*:

```
// koda 11.9 (lambde/lambdeInZbirke/LambdeInZbirke.java)
public static <T> int prestej(Collection<T> zbirka,
                             Predicate<T> pogoj) {
    int stevec = 0;
    for (T element: zbirka) {
        if (pogoj.test(element)) {
            stevec++;
        }
    }
    return stevec;
}
```

Ker je `Predicate` funkcijski vmesnik, lahko njegov objekt ustvarimo kot *lambdo*. V sledečem primeru izpišemo število sodih števil v seznamu tipa `List<Integer>` in število nizov določene dolžine v množici tipa `Set<String>`.

```
List<Integer> stevila = List.of(20, 15, 32, 7, 19, 14, 23, 35);
int stSodih = prestej(stevila, n -> n % 2 == 0);    // 3

int dolzina = 5;
Set<String> imena = Set.of("Ana", "Branko", "Cvetka", "Denis");
int stImen = prestej(imena, ime -> ime.length() == dolzina); // 1
```

V *lambdi* lahko uporabimo lokalno spremenljivko `dolzina`, ker jo nastavimo samo enkrat.

#### Naloga 11.14 Napišite metodo

```
public static <T> boolean vsi(Collection<T> zbirka,
                             Predicate<T> pogoj)
```

ki vrne `true` natanko v primeru, če vsi elementi podane zbirke izpolnjujejo podani pogoj.

**Naloga 11.15** Napišite metodo

```
public static boolean jePrastevilo(int stevilo)
```

ki vrne `true` natanko v primeru, če je podano število praštevilo. Pomagajte si z metodo vsi iz prejšnje naloge in z metodo, ki vrne seznam števil od *a* do *b*.

**11.7.2 Izvedba opravila za vsak element zbirke**

Ta metoda bo za vsak element zbirke izvedla določeno opravilo:

```
public static <T> void zaVsak(Collection<T> zbirka, opravilo)
```

Opravilo bomo predstavili z metodo, ki sprejme element tipa *T* in ne vrne ničesar. Parameter *opravilo* bo potemtakem objekt vmesnika, ki ponuja metodo

```
public abstract void metoda(T t)
```

Tudi tak vmesnik lahko najdemo v paketu `java.util.function`. Gre za vmesnik `Consumer`, njegova edina abstraktna metoda pa se imenuje `accept`. Metoda `zaVsak` se torej sprehodi po vseh elementih zbirke in za vsakega pokliče metodo `accept`:

```
// koda 11.10 (lambde/lambdeInZbirke/LambdeInZbirke.java)
public static <T> void zaVsak(Collection<T> zbirka,
                             Consumer<T> opravilo) {
    for (T element: zbirka) {
        opravilo.accept(element);
    }
}
```

Najenostavnejši primer uporabe metode `zaVsak` je izpis vsakega posameznega elementa:

```
List<Integer> stevila = List.of(20, 15, 32, 7, 19, 14, 23, 35);
zaVsak(stevila, n -> { System.out.println(n); });
```

V sledečem primeru vsako ime iz množice dodamo kot ključ v slovar. Pripadajoča vrednost je dolžina imena.

```
Set<String> imena = Set.of("Ana", "Branko", "Cvetka", "Denis");
Map<String, Integer> ime2dolzina = new TreeMap<>();
zaVsak(imena, ime -> { ime2dolzina.put(ime, ime.length()); });
// ime2dolzina: Ana -> 3, Branko -> 6, Cvetka -> 6, Denis -> 5
```

Zopet izkoriščamo možnost, da lahko v lambdi uporabljamo zunanje lokalne spremenljivke, če jih nastavimo samo enkrat.

**Naloga 11.16** Napišite metodo

```
public static <T> List<R> pretvori(Collection<T> zbirka,
                                   Function<T, R> funkcija)
```

ki na vsakem elementu podane zbirke uporabi podano funkcijo in vrne seznam dobljenih rezultatov.

**Naloga 11.17** Metodo pretvori uporabite, da

- seznam nizov celoštevilске oblike pretvorite v seznam števil (npr. ["5", "-17", "42"]  $\mapsto$  [5, -17, 42]);
- seznam pozitivnih števil pretvorite v seznam seznamov njihovih deliteljev (npr. [5, 1, 42]  $\mapsto$  [[1, 5], [1], [1, 2, 3, 6, 7, 14, 21, 42]]);
- slovar pretvorite v seznam objektov tipa Map.Entry.

**11.7.3 Združevanje elementov z dvojiškim operatorjem**

Ta metoda ima sledečo glavo:

```
public static <T> T zdruzi(Collection<T> zbirka,
                          BinaryOperator<T> operator, T zacetek)
```

Vmesnik BinaryOperator<T> ponuja metodo, ki sprejme dva parametra tipa T in vrne rezultat istega tipa:

```
public abstract T apply(T t, T u)
```

Metoda zdruzi najprej uporabi podani operator na elementu zacetek in prvem elementu podane zbirke. Nato operator uporabi na dobljenem rezultatu in drugem elementu zbirke. V naslednjem koraku uporabi operator na pravkar dobljenem rezultatu in tretjem elementu zbirke. Tako nadaljuje do konca zbirke. Če elemente zbirke označimo z  $e_0, e_1, \dots, e_{n-1}$ , operator pa s simbolom  $\circ$ , potem metoda zdruzi vrne rezultat izraza

$$((((zacetek \circ e_0) \circ e_1) \circ e_2) \circ \dots) \circ e_{n-1}.$$

Opisane metode ni težko napisati:

```
// koda 11.11 (lambde/lambdeInZbirke/LambdeInZbirke.java)
public static <T> T zdruzi(Collection<T> zbirka,
                          BinaryOperator<T> operator, T zacetek) {
    T rezultat = zacetek;
```

```

    for (T element: zbirka) {
        rezultat = operator.apply(rezultat, element);
    }
    return rezultat;
}

```

Kako bi nam metoda `zdruzi` lahko koristila? Če ji podamo zbirko števil, operator seštevanja in začetno vrednost 0, potem vrne vrednost  $((((0+e_0)+e_1)+e_2)+\dots)+e_{n-1}$ , torej vsoto elementov:

```

List<Integer> stevila = List.of(20, 15, 32, 7, 19, 14, 23, 35);
int vsota = zdruzi(stevila, (a, b) -> a + b, 0); // 165

```

Če metodi `zdruzi` podtaknemo operator, ki vrne večjega izmed podanih elementov, lahko elegantno pridobimo največji element:

```

Set<String> imena = Set.of("Ana", "Branko", "Cvetka", "Denis");
String najdaljseIme = zdruzi(imena,
    (a, b) -> (a.length() > b.length() ? a : b),
    ""); // Branko

```

**Naloga 11.18** S pomočjo metode `zdruzi` napišite metodo

```

public static <T> List<T> splosci(List<List<T>> seznamSeznamov)

```

ki vrne seznam, ki po vrsti vsebuje elemente posameznih seznamov v podanem seznamu seznamov. Na primer, če metodi podamo seznam `[[5, 2, 10], [3], [4, 6]]`, dobimo seznam `[5, 2, 10, 3, 4, 6]`.

#### 11.7.4 Grupiranje elementov po rezultatih funkcije

Metoda

```

public static <T, R> Map<R, List<T>> grupiraj(
    Collection<T> zbirka, Function<T, R> funkcija)

```

uporabi podano funkcijo na vsakem elementu podane zbirke in vrne slovar, ki vsak posamezni rezultat funkcije preslika v seznam elementov zbirke, ki so dali ta rezultat. Na primer, klic

```

grupiraj(List.of(-3, 5, -5, 1, 3, -3), a -> a * a)

```

vrne slovar s preslikavami  $1 \mapsto [1]$ ,  $9 \mapsto [-3, 3, -3]$  in  $25 \mapsto [5, -5]$ .

Metoda `grupiraj` najprej ustvari prazen slovar (kot objekt tipa `HashMap`, ne `TreeMap`, saj ne moremo predpostaviti, da je tip `T` podtip tipa `Comparable<T>`), nato

pa v zanki uporabi podano funkcijo na vsakem elementu zbirke in sproti posodablja slovar. To stori tako, da preveri, ali rezultat klica funkcije že obstaja kot ključ v slovarju. Če še ne obstaja, ga doda kot ključ in mu pripiše seznam, sestavljen iz trenutnega elementa zbirke. V nasprotnem primeru pa v seznam, ki pripada rezultatu, zgolj doda trenutni element zbirke.

```
// koda 11.12 (lambde/lambdeInZbirke/LambdeInZbirke.java)
public static <T, R> Map<R, List<T>> grupiraj(
    Collection<T> zbirka, Function<T, R> funkcija) {

    Map<R, List<T>> slovar = new HashMap<>();
    for (T element: zbirka) {
        R rezultat = funkcija.apply(element);
        List<T> elementiZaRezultat = slovar.get(rezultat);
        if (elementiZaRezultat == null) {
            elementiZaRezultat = new ArrayList<T>();
            slovar.put(rezultat, elementiZaRezultat);
        }
        elementiZaRezultat.add(element);
    }
    return slovar;
}
```

V sledečem primeru števila grupiramo glede na njihovo sodost, nize pa glede na njihovo dolžino:

```
List<Integer> stevila = List.of(20, 15, 32, 7, 19, 14, 23, 35);
Map<Boolean, List<Integer>> sodost2stevila = grupiraj(
    stevila, n -> n % 2 == 0);
// false -> [15, 7, 19, 23, 35], true -> [20, 32, 14]

Set<String> imena = Set.of("Ana", "Branko", "Cvetka", "Denis");
Map<Integer, List<String>> dolzina2imena = grupiraj(
    imena, ime -> ime.length());
// 3 -> [Ana], 5 -> [Denis], 6 -> [Branko, Cvetka]
```

### Naloga 11.19 Napišite metodo

```
public static <T, R> List<List<T>> ekvivalencniRazredi(
    Collection<T> zbirka, Function<T, R> funkcija)
```

ki vrne seznam, sestavljen iz seznamov elementov podane zbirke, ki jih podana funkcija preslika v isto vrednost. Na primer, pri zbirki `[-3, -1, 1, 1, 3]` in

funkciji  $x \rightarrow x * x$  naj metoda vrne seznam `[[-3, 3], [-1, 1, 1]]`.

**Naloga 11.20** Napišite metodo

```
public static <T, R> Map<R, List<Par<T, T>>> grupirajPare(
    Collection<T> zbirka, BiFunction<T, T, R> funkcija)
```

ki vrne slovar, ki vrednost  $r$  preslika v seznam vseh parov elementov  $p$  in  $q$  iz podane zbirke, za katere podana funkcija vrne rezultat  $r$ . Na primer, klic

```
grupirajPare(Set.of(1, 2, 3), (a, b) -> a < b)
```

vrne slovar, ki vrednost `true` preslika v seznam parov (1,2), (1,3) in (2,3), vrednost `false` pa v seznam parov (1,1), (2,1), (2,2), (3,1), (3,2) in (3,3).

**Naloga 11.21** Napišite metodo

```
public static <T, R> Function<T, R>
    izSlovarja(Map<R, List<T>>> slovar)
```

ki vrne funkcijo, za katero bi metoda `grupiraj` vrnila podani slovar.

## 11.8 Primer: obdelava rezultatov izpita

Poglavje bomo sklenili s programom za obdelavo rezultatov izpita, spotoma pa se bomo še česa naučili.

Študentje ljubljanske Fakultete za računalništvo in informatiko opravljajo izpit pri predmetu Programiranje 1 na univerzitetnih študijskih programih tako, da na računalnik rešujejo programerske naloge, te pa se nato s pomočjo testnih primerov preverijo in ocenijo. Za vsak testni primer, pri katerem njihov program za določeno nalogo v predpisani časovni omejitvi proizvede pravilen rezultat, prejmejo po eno točko. Napišimo program, ki na podlagi datoteke s podatki o udeležencih izpita in datotek s podatki o njihovih rezultatih pri posameznih nalogah izdela dve zbirni datoteki. Obe za vsakega udeleženca izpita podajata število doseženih točk pri posameznih nalogah in skupno število točk na izpitu, razlika je le v tem, da so udeleženci v prvi izhodni datoteki urejeni po priimkih (primarno) in imenih (sekundarno), v drugi pa po padajočem skupnem številu doseženih točk.

### 11.8.1 Vhod in izhod

Naši dosedanji programi so brali s standardnega vhoda in pisali na standardni izhod, tokrat pa bomo podatke brali iz *datotek*, vanje pa bomo tudi izpisovali rezultate. Zakaj ne bi tudi tokrat enostavno preusmerili standardnega vhoda in izhoda? Pri standardnem vhodu in izhodu smo omejeni na en sam vhodni in izhodni tok, tokrat

pa imamo  $n + 1$  vhodnih tokov (pri čemer je  $n$  število izpitnih nalog) in dva izhodna tokova. Brez skrbi, bojazen je odveč: besedilne datoteke lahko obravnavamo na enak način kot standardni vhod in izhod, edina razlika je pravzaprav ta, da moramo vsako datoteko *odpreti* in *zapreti*, standardni vhod in izhod pa sta vseskozi odprta.

Vse datoteke bodo zapisane v formatu CSV (angl. comma-separated values). Vsaka vrstica take datoteke vsebuje določeno število *polj* (npr. števil ali nizov), ki so med seboj ločena z vejicami ali katerim drugim ločilom. Podatki so ponavadi konceptualno organizirani v stolpce: vsa polja do prvega ločila torej tvorijo prvi stolpec, polja med prvim in drugim ločilom tvorijo drugi stolpec itd. Datoteke CSV lahko uvozimo v številne elektronske preglednice, zato so precej priljubljene.

V našem primeru bo datoteka s podatki o udeležencih razdeljena na stolpce VŠ (vpisna številka), Ime in Priimek. Vpisna številka je šifra, ki enolično določa študenta univerze. Ker se vpisna številka lahko prične z ničlo, jo bomo obravnavali kot niz, ne kot število. Datoteka z rezultati posameznih nalog bo razdeljena na stolpce VŠ, 1, 2, ...,  $m$ , kjer je  $m$  število testnih primerov. Stolpci 1, ...,  $m$  lahko vsebujejo le enice in ničle: enica v stolpcu  $i$  pove, da je študentov program pravilno obravnaval  $i$ -ti testni primer. Predpostavili bomo, da je število testnih primerov pri vseh nalogah enako.

Izhodni datoteki bosta sestavljeni na enak način, razlikovali se bosta le po vrstnem redu vrstic. Prvi trije stolpci bodo podajali vpisno številko ter ime in priimek študenta. V stolpcih N1, N2, ...,  $Nn$  bo zapisano skupno število točk pri posameznih nalogah, v stolpcu Skupaj pa skupno število doseženih točk na izpitu.

Slika 11.1 prikazuje primer za  $n = 3$  in  $m = 10$ . Datoteke udelezenci.csv, naloga1.csv, naloga2.csv in naloga3.csv so vhodne, datoteki poPriimkih.csv in poTockah.csv pa izhodni. Vidimo, da ima vsaka datoteka tudi *glavo* — vrstico z imeni stolpcev.

Program bomo poimenovali ObdelajIzpit in ga v terminalu pognali takole:

```
terminal> java ObdelajIzpit datUdeleženci datNaloga1 datNaloga2 ...
                        datNalogaN datPoPriimkih datPoTockah
```

Vidimo, da bomo poti do vhodnih in izhodnih datotek podali kar kot *argumente ukazne vrstice*. Kako jih lahko program pridobi? Preprosto. Argumenti ukazne vrstice se ob zagonu programa prenesejo v tabelo args, ki jo kot parameter sprejme metoda main:

```
public static void main(String[] args)
```

V primeru na sliki 11.1 program pokličemo takole:

```
terminal> java ObdelajIzpit udelezenci.csv naloga1.csv naloga2.csv
                        naloga3.csv poPriimkih.csv poTockah.csv
```



```
VŠ;Ime;Priimek
345;Janez;Novak
072;Mojca;Bergant
296;Iva;Novak
515;Peter;Vovk
723;Denis;Novak
429;Darja;Bergant
```

udelezenci.csv

```
VŠ;1;2;3;4;5;6;7;8;9;10
429;1;1;1;1;0;0;0;0;0;0
515;1;1;1;1;1;1;1;1;1;1
296;1;1;1;1;0;1;0;1;1;1
072;0;0;0;0;0;0;0;0;0;0
345;0;1;0;0;0;1;0;0;0;0
```

naloga1.csv

```
VŠ;1;2;3;4;5;6;7;8;9;10
296;1;1;1;1;1;1;1;1;1;0
072;1;1;1;0;0;0;1;0;0;0
515;1;1;1;1;1;1;0;0;0;0
345;0;0;1;1;1;0;0;0;0;0
```

naloga2.csv

```
VŠ;1;2;3;4;5;6;7;8;9;10
515;1;1;1;1;0;0;0;0;0;0
429;1;1;0;0;1;1;1;1;1;1
296;1;1;1;1;1;1;1;1;0;1
072;1;1;0;1;1;1;1;0;0;0
```

naloga3.csv

```
VŠ;Ime;Priimek;N1;N2;N3;Skupaj
429;Darja;Bergant;4;0;8;12
072;Mojca;Bergant;0;4;6;10
723;Denis;Novak;0;0;0;0
296;Iva;Novak;8;9;9;26
345;Janez;Novak;2;3;0;5
515;Peter;Vovk;10;6;4;20
```

poPriimkih.csv

```
VŠ;Ime;Priimek;N1;N2;N3;Skupaj
296;Iva;Novak;8;9;9;26
515;Peter;Vovk;10;6;4;20
429;Darja;Bergant;4;0;8;12
072;Mojca;Bergant;0;4;6;10
345;Janez;Novak;2;3;0;5
723;Denis;Novak;0;0;0;0
```

poTockah.csv

**Slika 11.1** Program za obdelavo rezultatov izpita na podlagi datotek udelezenci.csv, naloga1.csv, naloga2.csv in naloga3.csv izdela datoteki poPriimkih.csv in poTockah.csv. V arhivu ZIP na spletni strani knjige so datoteke dostopne v imeniku lambde/obdelavaRezultatovIzpita.

Niz `udelezenci.csv` se bo torej prenesel v element `args[0]`, niz `naloga1.csv` v element `args[1]` itd.

Končno smo spoznali vse komponente glave metode `main`!

### 11.8.2 Razred Student

Udeležence izpita — študente — bomo predstavili kot objekte tipa `Student`. Vsak študent je opisan s tremi podatki: vpisno številko, imenom in priimkom.

```
// koda 11.13 (lambde/izpit/Student.java)
public class Student {
    private String vpisna;
    private String ime;
    private String priimek;

    public Student(String vpisna, String ime, String priimek) {
        this.vpisna = vpisna;
        this.ime = ime;
        this.priimek = priimek;
    }
}
```

Razred `Student` bomo v nadaljevanju še dopolnili.

### 11.8.3 Branje

V glavnem razredu (`ObdelajIzpit`) moramo najprej analizirati argumente ukazne vrstice. Ta vsebuje poti do vhodnih in izhodnih datotek. Prvi argument je datoteka s podatki o udeležencih izpita, nato sledi  $n$  datotek s podatki o doseženih točkah pri posameznih nalogah, zadnja dva argumenta pa sta obe izhodni datoteki. Števila  $n$  vnaprej ne poznamo, a ga ni težko izračunati, saj vemo, da je število vseh argumentov ukazne vrstice enako `args.length`.

```
// koda 11.14 (lambde/izpit/ObdelajIzpit.java)
import java.util.*;
import java.io.*;    // razredi za delo z datotekami

public class ObdelajIzpit {
    public static void main(String[] args) {
        String datStudentje = args[0];
        int stNalog = args.length - 3;

        String[] datNaloge = new String[stNalog];
```

```

        for (int i = 0; i < stNalog; i++) {
            datNaloge[i] = args[i + 1];
        }
        String datPoPriimkih = args[args.length - 2];
        String datPoTockah = args[args.length - 1];
    }
}

```

Udeležence izpita bi lahko prebrali v objekt tipa `List<Student>`, a si bomo raje pomagali s slovarjem, v katerem so študentje dostopni prek njihovih vpisnih števil. Podobno bomo ravnali tudi pri podatkih o dosežkih na izpitu: prebrali jih bomo v slovar, ki vpisno številko študenta preslika v seznam točk, ki jih je zbral pri posameznih nalogah. Tako bomo na naraven način vzpostavili povezavo med osebnimi podatki študentov in njihovimi dosežki na izpitu: če poznamo vpisno številko študenta, zlahka pridobimo oboje.

```

public class ObdelajIzpit {
    public static void main(String[] args) {
        ...
        Map<String, Student> vpisna2student =
            preberiStudente(datStudentje);
        Map<String, List<Integer>> vpisna2tocke =
            preberiTocke(datNaloge);
    }
}

```

Metoda `preberiStudente` iz podane datoteke prebere podatke o študentih. Java ponuja številne razrede za branje besedilnih datotek, najpripravnejši pa bo naš stari znanec `Scanner`, saj ga lahko uporabljamo na enak način kot pri branju s standardnega vhoda. Razlika je le v inicializaciji: medtem ko smo pri branju s standardnega vhoda novonastalemu objektu tipa `Scanner` podali objekt `System.in`, mu moramo pri branju iz datoteke podati objekt tipa `File`. Temu objektu ob izdelavi podamo pot do datoteke, iz katere bomo brali:

```

Scanner sc = new Scanner(new File(potDoDatoteke));

```

Ta vrstica datoteko *odpre* (pripravi za branje) in izdelava objekt tipa `Scanner`, s pomočjo katerega bomo datoteko lahko prebrali. Pri odpiranju datoteke lahko pride do napake, ki sproži izjemo: lahko se, denimo, zgodi, da podana datoteka ne obstaja. Kadar odsek kode lahko sproži izjemo, ki je posledica zunanjih okoliščin, ne pa programerskih napak, ki jih je načeloma mogoče preprečiti (kot je npr. dostop do tabele z neveljavnim indeksom), moramo morebitno izjemo *uloviti* ali pa z deklaracijo `throws` naznaniti, da lahko metoda sproži izjemo. Praviloma je prva možnost

boljša.

Če lahko stavki *stavki* sprožijo izjemo tipa *TipIzjeme*, jo ulovimo takole:

```
try {
    stavki
} catch (TipIzjeme izjema) {
    obdelaj izjemo
}
```

Kako se izvrši stavek try-catch? Koda v odseku try se izvrši na običajen način. Če se pri tem ne sproži izjema tipa *TipIzjeme*, se odsek catch ignorira. Če se izjema takega tipa sproži, pa se odsek try takoj prekine in se izvede odsek catch. V odseku catch lahko uporabljamo objekt *izjema*, ki predstavlja sproženo izjemo.

Koda za odpiranje datoteke lahko sproži izjemo tipa *FileNotFoundException*, zato je ena od možnih rešitev takšna:

```
try {
    Scanner sc = new Scanner(new File(datoteka));
    preberi datoteko
    sc.close(); // zapri datoteko
} catch (FileNotFoundException ex) {
    izpiši obvestilo, da datoteka ne obstaja
}
```

Ta rešitev sicer deluje, vendar pa moramo upoštevati, da se izjema lahko sproži tudi med branjem datoteke (če, na primer, poskusimo ime študenta prebrati s klicem *sc.nextInt()*). Če take izjeme ne ulovimo, datoteke morda ne bomo zaprli. Ker naj bi datoteko na koncu zaprli v vseh mogočih primerih, raje uporabljamo nekoliko drugačno obliko stavka try-catch, ki nam to zagotavlja:

```
try (Scanner sc = new Scanner(new File(datoteka))) {
    preberi datoteko
} catch (FileNotFoundException ex) {
    izpiši obvestilo, da datoteka ne obstaja
}
```

Klic *sc.close()* ni več potreben, saj gornja oblika stavka try-catch datoteko samodejno zapre.

Sedaj smo pripravljeni, da razred *ObdelajIzpit* dopolnimo z metodo *preberiStudente*:

```
private static Map<String, Student> preberiStudente(String datoteka){
    Map<String, Student> vpisna2student = new HashMap<>();
```

```

int stVrstice = 2;
try (Scanner sc = new Scanner(new File(datoteka))) {
    sc.nextLine(); // preskočimo glavo
    while (sc.hasNextLine()) {
        String vrstica = sc.nextLine();
        String[] komponente = vrstica.split(";");
        String vpisna = komponente[0];
        String ime = komponente[1];
        String priimek = komponente[2];
        vpisna2student.put(vpisna,
                           new Student(vpisna, ime, priimek));
        stVrstice++;
    }
} catch (FileNotFoundException ex) {
    System.err.printf("Datoteka %s ne obstaja.%n", datoteka);
} catch (ArrayIndexOutOfBoundsException ex) {
    System.err.printf("Datoteka %s: napaka v %d. vrstici%n",
                      datoteka, stVrstice);
}

return vpisna2student;
}

```

Datoteko beremo po vrsticah (klic `sc.hasNextLine()` vrne `true`, če na vhodu še obstaja kakšna vrstica), vsako prebrano vrstico pa razbijemo z metodo `split`. Vsakega študenta shranimo v slovar, pri čemer nam vpisna številka služi kot ključ, objekt tipa `Student` pa kot vrednost.

Poleg odseka `catch`, ki lovi izjeme tipa `FileNotFoundException`, smo zapisali tudi odsek, ki lovi izjeme tipa `ArrayIndexOutOfBoundsException`. Kot že vemo, se izjema tega tipa sproži, ko do tabele dostopamo z neveljavnim indeksom. To se lahko zgodi le v primeru, če ima vrstica manj kot tri komponente, torej če ni skladna z dogovorjenim formatom. Ob sprožitvi izjeme izpišemo zaporedno številko problematične vrstice.

Kaj pa predstavlja beseda `err` v odseku `catch`? Poleg običajnega standardnega izhoda, ki ga predstavlja objekt `System.out`, obstaja še *standardni izhod za napake*, ki ga predstavlja objekt `System.err`. Ta izhod je namenjen obvestilom o napakah. V lupini operacijskega sistema lahko izhoda ločeno obdelujemo (na primer, standardni izhod za napake preusmerimo v dnevniško datoteko, običajni standardni izhod pa posredujemo nekemu drugemu procesu).

Metoda `preberiTocke`, ki iz podanih datotek prebere podatke o zbranih točkah pri posameznih nalogah, deluje po enakih načelih:

```

private static Map<String, List<Integer>>
    preberiTocke(String[] datoteke) {

    int stNalog = datoteke.length;
    Map<String, List<Integer>> vpisna2tocke = new HashMap<>();

    for (int iNaloga = 0; iNaloga < datoteke.length; iNaloga++) {

        try (Scanner sc = new Scanner(new File(datoteke[iNaloga]))) {
            sc.nextLine();    // preskočimo glavo

            while (sc.hasNextLine()) {
                String vrstica = sc.nextLine();
                String[] komponente = vrstica.split(";");
                String vpisna = komponente[0];
                int tocke = 0;
                for (int t = 1; t < komponente.length; t++) {
                    tocke += Integer.parseInt(komponente[t]);
                }

                List<Integer> tockeNalog =
                    vpisna2tocke.get(vpisna);    // (1)
                if (tockeNalog == null) {
                    tockeNalog = new ArrayList<>(
                        Collections.nCopies(stNalog, 0));
                    vpisna2tocke.put(vpisna, tockeNalog);
                }
                tockeNalog.set(iNaloga, tocke);    // (2)
            }
        } catch (FileNotFoundException ex) {
            System.err.printf("Datoteka %s ne obstaja.",
                               datoteke[iNaloga]);
        }
    }
    return vpisna2tocke;
}

```

Za vsako nalogo preberemo pripadajočo datoteko. Vsaka vrstica v datoteki je sestavljena iz vpisne številke in zaporedja enic in ničel. Enice in ničle enostavno seštejemo in dobimo skupno število točk.

Ni nujno, da študent odda programe za vse naloge. Če katerega od programov ne odda, ima pri pripadajoči nalogi pač ničlo. Vsakemu študentu pripišemo seznam

tipa `List<Integer>`, v katerem  $i$ -ti element hrani število točk za  $i$ -to nalogo. V vrstici (1) pridobimo seznam, ki pripada trenutno obravnavanemu študentu. Če ta seznam obstaja (to se zgodi v primeru, če je študent oddal program že za katero od prejšnjih nalog), potem v vrstici (2) zgolj nastavimo element seznama z indeksom `iNaloga` na zbrano število točk. Če za tekočega študenta še nimamo seznama točk, pa seznam pred to operacijo ustvarimo (metoda `Collections.nCopies(n, k)` vrne seznam, sestavljen iz  $n$  kopij elementa  $k$ ) in ga pripišemo tekočemu študentu. Vpisna številka je ponovno ključ, izdelani seznam pa je pripadajoča vrednost.

#### 11.8.4 Urejanje

Študente bomo urejali bodisi po priimkih in imenih bodisi po zbranem številu točk. Za potrebe urejanja po prvem kriteriju bomo razred `Student` dopolnili z metodo `primerjaj`. Metoda primerja študenta `this` in drugi po njunih priimkih, študenta z enakima priimkoma pa še po imenih:

```
public class Student {
    ...
    public int primerjaj(Student drugi) {
        int c = this.priimek.compareTo(drugi.priimek);
        return (c != 0) ? (c) : (this.ime.compareTo(drugi.ime));
    }
}
```

Zakaj nismo preprosto implementirali vmesnika `Comparable<Student>` in definirali metode `compareTo`? Zato, ker ni povsem jasno, kakšna je naravna urejenost pri študentih. Urejenost po vpisnih številkah ni nič manj »naravna« kot urejenost po priimkih in imenih.

Ker lahko vse podatke o posameznih študentih pridobimo na podlagi njihovih vpisnih števil, si bomo pripravili seznam vpisnih števil vseh študentov in ga uredili glede na priimke in imena oziroma glede na vsoto zbranih točk pripadajočih študentov. Seznam vpisnih števil lahko izdelamo na podlagi množice ključev enega ali drugega slovarja:

```
List<String> vpisne = new ArrayList<>(vpisna2student.keySet());
```

Seznam uredimo z metodo `sort`. Metodi kot argument posredujemo primerjalnik (objekt tipa `Comparator<String>`), ki na podlagi podanega para vpisnih števil pridobi pripadajoča študenta (objekta tipa `Student`) in vrne rezultat njune primerjave po izbranem kriteriju. Pri urejanju po priimkih in imenih si pomagamo z metodo `primerjaj` iz razreda `Student`:

```
vpisne.sort((vp1, vp2) ->
    vpisna2student.get(vp1).primerjaj(vpisna2student.get(vp2)));
```

Pri urejanju po vsoti zbranih točk ravnamo podobno. Primerjalnikova metoda `compare`, ki jo zapišemo z lambda, sprejme par vpisnih števil (vp1 in vp2) in vrne razliko vsot zbranih točk pripadajočih študentov. Ker vpisne številke urejamo po padajoči vsoti točk, mora metoda `compare` vrniti negativen rezultat natanko tedaj, ko ima študent z vpisno številko vp2 manjšo vsoto točk od študenta z vpisno številko vp1.

```
vpisne.sort((vp1, vp2) ->
    vsota(vpisna2tocke.get(vp2)) - vsota(vpisna2tocke.get(vp1)));
```

Metoda `vsota` vrne vsoto podanega seznama:

```
public class ObdelajIzpit {
    ...
    private static int vsota(List<Integer> tocke) {
        if (tocke == null) {
            return 0;
        }
        int vsota = 0;
        for (int t: tocke) {
            vsota += t;
        }
        return vsota;
    }
}
```

Zakaj preverimo, ali je parameter `tocke` enak `null`? Zato, ker se lahko zgodi, da študent ne odda nobene naloge. Na takega študenta v metodi `preberiTocke` ne bomo naleteli, zato zanj ne bomo ustvarili seznama zbranih točk in ne bomo vnesli para (vpisna številka, seznam točk) v slovar `vpisna2tocke`. Klic `vpisna2tocke.get(vpisna)` bo za takega študenta vrnil `null`.

### 11.8.5 Izpis

Pisanje v datoteko je še enostavnejše od branja. S stavkom

```
Writer writer = new FileWriter(potDoDatoteke);
```

odpremo datoteko za pisanje in ustvarimo objekt, nad katerim lahko kličemo metodo `write` in s tem v datoteko zapisujemo posamezne nize. Datoteko moramo na koncu obvezno zapreti ...

```
writer.close()
```

... lahko pa za to namesto nas poskrbi stavek `try-catch`:



```
try (Writer writer = new FileWriter(potDoDatoteke)) {
    piši v datoteko s klici writer.write(...)
} catch (IOException ex) {
    System.err.printf("Napaka pri pisanju v datoteko %s.", datoteka);
}
```

Metoda izpisi se sprehodi po (ustrezno urejenih) vpisnih številkah (parameter vpisne) in jih uporabi kot ključe za dostop do podatkov o študentih in njihovih točkah v slovarjih vpisna2student in vpisna2tocke. Če študent ni oddal nobenega programa, je pri vseh nalogah zbral 0 točk.

```
private static void izpisi(
    List<String> vpisne,
    Map<String, Student> vpisna2student,
    Map<String, List<Integer>> vpisna2tocke,
    int stNalog,
    String datoteka) {

    try (Writer writer = new FileWriter(datoteka)) {

        // zapišemo glavo
        StringBuilder sbNaloge = new StringBuilder();
        for (int i = 0; i < stNalog; i++) {
            sbNaloge.append(String.format("N%d;", i + 1));
        }
        writer.write(String.format("VŠ;Ime;Priimek;%sSkupaj%n",
            sbNaloge));

        for (String vpisna: vpisne) {
            Student student = vpisna2student.get(vpisna);
            List<Integer> tocke = vpisna2tocke.get(vpisna);
            if (tocke == null) {
                tocke = new ArrayList<>(
                    Collections.nCopies(stNalog, 0));
            }

            String strTocke = tocke.toString(); // (1)
            strTocke = strTocke.substring(1, strTocke.length() - 1);
            strTocke = strTocke.replace(", ", ";");
            writer.write(String.format("%s;%s;%d%n",
                student.csv(), strTocke, vsota(tocke)));
        }
    }
}
```

```

    } catch (IOException ex) {
        System.err.printf("Napaka pri pisanju v datoteko %s.",
                           datoteka);
    }
}

```

V odseku kode, ki se prične v vrstici (1), na podlagi niza  $[t_1, t_2, \dots, t_n]$ , ki ga proizvede metoda `toString`, izdelamo niz  $t_1; t_2; \dots; t_n$  in ga zapišemo v datoteko.

Metoda `csv` v razredu `Student` vrne niz, v katerem so podatki o študentu `this` ločeni s podpičjema:

```

public class Student {
    ...
    public String csv() {
        return String.format("%s;%s;%s",
                              this.vpisna, this.ime, this.priimek);
    }
}

```

Dokončati moramo še metodo `main`. Pridobiti moramo seznam vpisnih števil, ga urediti po obeh kriterijih in ustvariti obe izhodni datoteki.

```

public class ObdelajIzpit {
    public static void main(String[] args) {
        ...
        List<String> vpisne = new ArrayList<>(
            vpisna2student.keySet());

        vpisne.sort((vp1, vp2) -> vpisna2student.get(vp1).primerjaj(
            vpisna2student.get(vp2)));
        izpisi(vpisne, vpisna2student, vpisna2tocke,
            stNalog, datPoPriimkih);

        vpisne.sort((vp1, vp2) -> vsota(vpisna2tocke.get(vp2)) -
            vsota(vpisna2tocke.get(vp1)));
        izpisi(vpisne, vpisna2student, vpisna2tocke,
            stNalog, datPoTockah);
    }
}

```

**Naloga 11.22** Popravite program tako, da bo urejanje študentov po obeh kriterijih stabilno: če se študenta po izbranem kriteriju urejanja med seboj ne razlikujeta, mora biti njun medsebojni vrstni red v izhodni datoteki enak kot v datoteki s seznamom udeležencev izpita. (Naloga ni posebej težka, saj metoda `sort` v vmesniku `List` sezname že sama po sebi stabilno ureja.)

**Naloga 11.23** Nadgradite program tako, da bo proizvedel eno samo izhodno datoteko, ta pa bo urejena tako, kot narekuje dodatni (npr. prvi) argument ukazne vrstice. Na primer, če program poženemo z ukazno vrstico

```
terminal> java ObdelajIzpit 'P,-2,S,-I' ostali_argumenti...
```

naj bodo študentje v izhodni datoteki primarno urejeni po »naraščajočih« priimkih (P), sekundarno po padajočem številu točk pri drugi nalogi (-2), terciarno po naraščajočem skupnem številu točk (S), kvartarno pa po »padajočih« imenih (-I).

## 11.9 Povzetek

- Funkcijski vmesnik je vmesnik z eno samo abstraktno metodo.
- S pomočjo lambde lahko ustvarimo objekt funkcijskega vmesnika, ne da bi nam bilo treba izdelati razred, ki ta vmesnik implementira. Določiti moramo zgolj implementacijo abstraktne metode.
- Lambda je izraz, ki vsebuje (sintaktično zgoščeno) implementacijo abstraktne metode funkcijskega vmesnika. Lambda ustvari nov objekt funkcijskega vmesnika, njena vrednost pa je kazalec na ta objekt.
- Funkcijski vmesnik, ki ga implementira lambda, se določi iz konteksta, denimo na podlagi leve strani prireditvenega stavka (če lambda priredimo spremenljivki) ali pa na podlagi tipa pripadajočega parametra metode (če lambda posredujemo metodi kot argument).
- V lambdi lahko uporabljamo lokalne spremenljivke iz okolja, a le tiste, ki so deklarirane kot konstante, in tiste, ki jih uporabljamo, kot da bi bile konstante (to pomeni, da jim po inicializaciji več ne prirejamo vrednosti).

### *Iz profesorjevega kabineta*

»Μῆγιν ᾿αεῖδε, θεᾶ, Πηληιάδεω Ἀχιλῆος ...«

»Kaj naj bi to bilo, profesor Doberšek?«

»Homerjeva Iliada. No, njen prvi stih. Veš, Jože, jaz sem še iz tistih časov, ko se je za izobraženca skorajda pričakovalo, da bo znal latinsko, nekateri pa smo se učili

tudi starogrško. Današnja mladina pa, saj jih poslušam, lambda sem, lambda tja, a se sprašujem, koliko bi jih to črko sploh znalo napisati ...»

»Pa menite, da bi stari Grki napadli Trojo, če bi znali programirati?«

»Prmejduš ne vem. Zdi pa se mi, da bi si, Helena gor ali dol, marsikdo od njih pošteno belil glavo s takimi rečmi:

```
public class LeniSeznam {
    private int glava;
    private Supplier<LeniSeznam> leniRep;

    public LeniSeznam(int glava, Supplier<LeniSeznam> leniRep) {
        this.glava = glava;
        this.leniRep = leniRep;
    }

    public LeniSeznam rep() {
        return this.leniRep.get();
    }

    public LeniSeznam plus(LeniSeznam drugi) {
        return new LeniSeznam(
            this.glava + drugi.glava,
            () -> this.rep().plus(drugi.rep()));
    }

    public List<Integer> prvihN(int n) {
        List<Integer> rezultat = new ArrayList<>();
        LeniSeznam p = this;
        for (int i = 0; i < n; i++) {
            rezultat.add(p.glava);
            p = p.rep();
        }
        return rezultat;
    }
}

public class TestLS {
    public static void main(String[] args) {
        System.out.println(f().prvihN(10));
    }

    public static LeniSeznam f() {
```

```

        return new LeniSeznam(
            0, () -> new LeniSeznam(
                1, () -> f().plus(f().rep())));
    }
}

```

Tole mi je namreč nek študent poslal, ko sem ga povprašal glede neskončnih zaporedij v javi. Tuhtam in tuhtam, pa mi ni nič jasno. Jože, se ti kaj sanja? Genovefa?»

»Uf, tole pa bo kar zalogaj,« odvrne asistent Slapšak.

»Leni seznam, kot vidim,« prida docentka Javornik. »Tak seznam si lahko predstavljata kot povezani seznam, v katerem je vsako vozlišče sestavljeno iz povsem običajnega elementa (v našem primeru je to kar celo število) in lambde, ki ustvari naslednje vozlišče, ko njeno telo izvršimo z metodo `get`. Na začetku se ustvari samo prvo vozlišče. »Leno« v računalništvu pomeni, da se izračuna šele takrat, ko je res nujno, do takrat pa obstaja zgolj kot obljuba. Takšno obljubo lahko v javi predstavimo z `lambda`.«

»Kaj pa izpiše ta program?«

»Moram vzeti v roke papir in svinčnik, ne bo druge. Metoda `f` je kratka, a sladka. Ravno pravšnja za dostojno slovo ...«



## Literatura

- Douglas Adams. *Štoparski vodnik po Galaksiji: trilogija v štirih knjigah (prevedel Alojz Kodre)*. Tehniška založba Slovenije, 1996. ISBN 86-365-0189-X.
- Joshua Bloch. *Effective Java, 3rd Edition*. Addison-Wesley, 2017. ISBN 978-0134685991.
- Barry Burd. *Beginning Programming with Java For Dummies*. For Dummies, 2017. ISBN 978-1119235538.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, in Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN 978-0-262-03384-8.
- Allen B. Downey in Chris Mayfield. *Think Java: How to Think Like a Computer Scientist, 2nd Edition*. O'Reilly, 2019. ISBN 978-1492072508.
- Benjamin Evans, James Gough, in Chris Newland. *Optimizing Java: Practical Techniques for Improving JVM Application Performance*. O'Reilly, 2018. ISBN 978-1492025795.
- David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 2020. ISBN 978-1491952023.
- Andrej Florjančič. *Uvod v programiranje s programskim jezikom Java*. Mehatrona, 2017. ISBN 978-9619420904.
- Douglas Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979. ISBN 978-0-465-02656-2.
- Brian W. Kernighan in Dennis M. Ritchie. *The C Programming Language, 2nd Edition*. Pearson, 1988. ISBN 978-0131103627.
- Donald E. Knuth. *The TeXbook*. Addison-Wesley, 1986. ISBN 0-201-13447-0.
- David Kopec. *Classic Computer Science Problems in Java*. Manning, 2021. ISBN 978-1617297601.

Anghel Leonard. *Java Coding Problems: Improve your Java Programming skills by solving real-world coding challenges*. Packt, 2019. ISBN 978-1789801415.

Mark Lutz. *Learning Python, 5th Edition*. O'Reilly, 2013. ISBN 978-1449355739.

Viljan Mahnič, Luka Fürst, in Igor Rožanc. *Java skozi primere*. Bi-tim, 2008. ISBN 978-9616046107.

Uroš Mesojedec in Borut Fabjan. *Java 2, temelji programiranja*. Pasadena, 2004. ISBN 978-9616361309.

Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, in Chris Rowley. *The L<sup>A</sup>T<sub>E</sub>X Companion*. Addison-Wesley, 2004. ISBN 978-0201362992.

Maurice Naftalin in Philip Wadler. *Java Generics and Collections*. O'Reilly, 2006. ISBN 978-0596527754.

Herbert Schildt. *Java: The Complete Reference, 11th Edition*. McGraw-Hill, 2018. ISBN 978-1260440232.

William Shotts. *The Linux Command Line, 2nd Edition: A Complete Introduction*. No Starch Press, 2019. ISBN 978-1593279523.

Kathy Sierra in Bert Bates. *Head First Java, 2nd Edition*. O'Reilly, 2005. ISBN 978-0596009205.

Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley, 2013. ISBN 978-0321563842.

Raoul-Gabriel Urma, Mario Fusco, in Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and functional-style programming*. Manning, 2014. ISBN 978-1617291999.



## *Stvarno kazalo*

- !, 50
- !=, 49
- &&, 50
- \*, 19
- \*=, 61
- +, 19
- ++, 61
- +=, 61
- , 19
- =, 61
- , 61
- /, 19
- /=, 61
- <, 49
- <=, 49
- <<, 97
- =, 49
- ==, 49
- >, 49
- >=, 49
- >>, 97
- >>>, 97
- ?, 282
- ?:, 95
- %, 19
- %=, 61
- &, 97
- ^, 97
- ~, 97
- |, 97
- ||, 50
- abstract, 227
- algoritem
  - Dijkstrov, 161
  - Eratostenovo sito, 135
  - Floyd-Warshallov, 159
  - navadno vstavljanje, 141
- argument, 107
  - tipni, 270
  - ukazne vrstice, 380
- ArrayIndexOutOfBoundsException, 126
- ArrayList, 325
- ArrayStoreException, 240
- ASCII, 27
- atribut, 168
  - statični, 190
- BiConsumer, 369
- BiFunction, 369
- BinaryOperator, 369
- BiPredicate, 369
- bit, 96
- bitna predstavitev, 96
- blok, 31
- bločna zgradba, 11, 31
- Boolean, 246
- boolean, 49
- break, 80
- Byte, 246
- byte, 23
- case, 91
- catch, 384
- char, 27
- Character, 246

- class, 168
- ClassCastException, 237
- Collection, 321
- Collections, 357
- Comparable, 292
- Comparator, 296
- Consumer, 369
- continue, 83
- datoteka, 379
  - odpiranje, 383
- dedovanje, 211
  - in tabele, 223
  - večkratno, 289
- default, 91
- deklaracija, 20
- disjunkcija, 50
- do, 61
- Double, 246
- double, 20, 23
- dvojiški komplement, 97
- dvojiško iskalno drevo, 333
- dvojiško iskanje, 138
- else, 43
- else if, 47
- enum, 91
- equals, 242
- extends, 211, 274, 282, 290
- false, 49
- Fibonaccijevo zaporedje, 121
- final, 199, 313
- Float, 246
- float, 23
- for, 67
- Function, 369
- generiki, 269
  - metode, 273
  - nadomestni znak, 282
  - omejitve, 276
  - razredi, 272
- graf, 157
- hashCode, 244
- HashMap, 345
- HashSet, 330
- hrošč, 15
- if, 43
- implementacija, 174
- implements, 290
- import, 34
- inicializacija, 21
- instanceof, 235
- int, 20, 23
- Integer, 246
- interface, 290
- Iterable, 307
- Iterator, 299
- iterator, 299
  - nad slovarjem, 303
  - nad vektorjem, 300
- izhod, 32
- izjema, 14
  - lovljenje, 383
- izraz, 19
  - aritmetični, 19
  - logični, 49
  - pogojni, 95
  - primerjalni, 49
  - vrednost, 19
- izvajalnik, 9, 220
- izvorna koda, 9
- java, 2
  - namestitev, 2
- kazalec, 146
- koda, 9
- komentar, 13
  - bločni, 13
  - vrstični, 13
  - za javadoc, 14
- konjunkcija, 50

- konstanta, 199, 313
- konstruktor, 175
  - podrazreda, 214
  - privzeti, 195
  - več konstruktorjev, 194
- konzola, 17
- kovariantnost, 239, 282
- kratkostično izvajanje, 51
- lambda, 361
  - in lokalne spremenljivke, 372
  - in zbirke, 373
  - kontekst, 364
  - poenostavitve, 368
  - tip, 364
  - vrednost, 364
- LinkedList, 325
- List, 324
- Long, 246
- long, 23
- lupina, 3
  - skripta, 38
- Map, 343
- memoizacija, 142
- metoda, 11, 104, 177
  - abstraktna, 227
  - argument, 12, 107
  - generična, 273
  - getter, 181
  - implementacija, 227
  - izhodni tip, 113
  - klic, 12, 31, 105
  - klic v hierarhiji, 233
  - nestatična, 178
  - parameter, 107
  - podpis, 193
  - redefinicija, 212, 216
  - rekurzivni klic, 119
  - setter, 182
  - statična, 178
  - več istoimenskih, 193
  - vračanje vrednosti, 112
- množica, 329
  - uporaba, 338
- nadrazred, 212
- nadtip, 212
- nadvmesnik, 289
- napaka, 14
- navidezni stroj, 2
- negacija, 50
- new, 127, 175
- nit, 367
- niz, 12, 29, 195
- NoSuchElementException, 301
- NullPointerException, 249
- Object, 241
  - in tabele, 246
- objekt, 167
- operator
  - aritmetični, 18
  - asociativnost, 99
  - bitni, 97
  - desnoasociativni, 99
  - levoasociativni, 99
  - logični, 50
  - pogojni, 95
  - postfiksni ++, 61
  - prednostni nivo, 99
  - prefiksni ++, 61
  - primerjalni, 49
  - prireditveni, 49
  - sestavljani prireditveni, 60
- paket, 34
- parameter, 107
  - tipni, 270
- podrazred, 212
- podtip, 212
- podvmesnik, 289
- povezani seznam, 257
  - dvosmerni, 328
- Predicate, 369

prevajalnik, 2, 220  
 prireditev, 21  
 private, 174  
 program  
     avtomatizacija preverjanja, 38  
     dokazovanje pravilnosti, 40  
     preverjanje, 35  
     samostojen, 17  
 protected, 174  
 public, 174  
  
 Random, 65  
 razhroščevanje, 15  
 razred, 11, 168  
     abstrakten, 227  
     anonimni notranji, 312  
     dostopno določilo, 173  
     generični, 272  
     implementira vmesnik, 289  
     nespremenljiv, 183  
     nestatični notranji, 310  
     razširja razred, 212  
     statični notranji, 258, 309  
     testni, 170  
 rekurzija, 119  
 return, 112  
 rezervirana beseda, 10  
  
 Scanner, 33  
 Set, 330  
 seznam, 324  
 Short, 246  
 short, 23  
 sklad, 120  
 Slovar, 256, 280  
 slovar, 254, 342  
     ključ, 254  
     naivna implementacija, 255  
     sprehod, 303  
     uporaba, 348  
     vrednost, 254  
     z zgoščeno tabelo, 256

smetar, 197  
 sprehod  
     naravni, 309  
     po slovarju, 303  
     po tabeli, 128  
     po vektorju, 300  
 spremenljivka, 20  
     doseg, 31  
     ime, 20  
     lokalna, 107  
     nedefinirana, 21  
     prireditev vrednosti, 22  
     tip, 20  
 spremenljivka okolja, 7  
 standardni izhod, 32  
     preusmeritev, 37  
     za napake, 385  
 standardni vhod, 33  
     preusmeritev, 37  
 static, 178, 190  
 statični inicializator, 191  
 stavek, 12, 30  
     break, 80  
     continue, 83  
     do, 61  
     for, 67  
     if, 43  
     if-else, 43  
     pogojni, 43  
     prireditveni, 21  
     return, 112  
     switch, 90  
     veriga pogojnih, 47  
     while, 52  
     zaporedje, 43  
 String, 195  
 String.format, 185  
 super, 214, 216, 282  
 Supplier, 369  
 switch, 90  
 System.out.printf, 76  
 števec, 57

- tabela, 125
  - dolžina, 126
  - dvodimenzionalna, 150
  - dvojiško iskanje, 138
  - element, 125
  - indeks, 125
  - iskanje, 137
  - izdelava, 127
  - sprehod, 128
  - tridimenzionalna, 161
  - urejanje, 140
  - velikost, 126
- terminal, 3
- testni primer, 38
- this, 176, 188, 195
- throw, 301
- throws, 384
- tip, 18, 23
  - celoštevilski, 18
  - izrecna pretvorba, 25, 235
  - kazalca in objekta, 219
  - navzgor omejen, 274
  - ovojni, 246
  - parametriziran, 270
  - primitivni, 146
  - prirecljivost, 30
  - privzeta vrednost, 127
  - realnoštevilski, 18
  - referenčni, 146
  - samodejno pretvarjanje, 247
  - številski, 23
- toString, 184, 202, 242
- TreeMap, 345
- TreeSet, 333
- true, 49
- try, 384
- UnaryOperator, 369
- UnsupportedOperationException, 324
- urejanje, 387
  - navadno vstavljanje, 141
  - stabilno, 141
- urejenost
  - alternativna, 296
  - naravna, 292
- Vektor, 251, 277
- vektor, 198
  - sprehod, 300
- VektorInt, 198
- VektorString, 203
- vhod, 32
- vmesnik, 289
  - funkcijski, 364
  - implementacija, 289
  - razširja vmesnik, 289
- void, 112
- vrsta, 329
- vsebovalnik, 299, 319
  - hierarhija, 320
- while, 52
- zanka
  - do, 61
  - for, 67
  - for-each, 129, 154, 308
  - neskončna, 55
  - while, 52
- zbirka, 320
- zgoščena tabela, 256
- znak, 27
  - prelom vrstice, 28