PREVOD DRUGOG IZDANJA

Java programerski problemi

276 rešenih zadataka koji će vam pomoći u svakodnevnim izazovima

ANGEL LEONARD

kompjuter biblioteka

<packt>

PREVOD DRUGOG IZDANJA

Skenirajte QR kod, registrujte knjigu i osvojite nagradu

Java programerski problemi

Kroz 276 rešenih problema naučićete nove načine da se nosite sa svakodnevnim izazovima i značete odgovore na najčešće postavljana pitanja.

Java programerski problemi predstavlja najnovije funkcionalnosti jezika, ali ne insistira na primeni najnovijih rešenja. Umesto toga, fokus je na otkrivanju kompromisa pri izboru najboljeg rešenja za određeni problem.

Za koga je ova knjiga

Ova knjiga je namenjena Java programerima koji žele da rešavanjem svakodnevnih problema unaprede svoje veštine. Da se u potpunosti iskoristi ova knjiga, potrebno je osnovno praktično poznavanje programskog jezika Java.

Šta ćete naučiti:

- Primenu najnovijih funkcionalnosti JDK 21 verzije: integracija novih mogućnosti u vaše aplikacije.
- Rad sa zapisima: istraživanje, obrasci, serijalizacija zapisa i ostalo.
- Rad sa zatvorenim klasama i interfejsima: značaj zatvorenih klasa i interfejsa za povećanje enkapsulacije.
- Rešavanje problema sa kolekcijama i ezoteričnim strukturama podataka: koncepti i primene naprednih rešenja.
- Filtere za deserijalizaciju specifične za kontekst: kako se bezbedno upravlja podacima.
- Napredne tehnike proširivanja funkcionalnosti Java API interfejsa: metodi za efikasnije upravljanje podacima.
- Novi Socket API interfejs i Simple Web Server alat: savremene mogućnosti rada sa aplikacijama na mreži.
- Savremene sakupljače otpada i dinamičke CDS arhive: moderni alati za upravljanje memorijom.

Budite spremni za poslovni intervju

Uz zbirku svakodnevnih Java problema, brzo ćete steći uvid u širok spektar izazova, od osnovnih do naprednih, s kojima je Java programer suočen u svakodnevnom radu. Značete odgovore čak i na najteža pitanja na poslovnim intervjuima i dobićete posao koji želite.

ISBN: 978-86-7310-608-3

Angel Leonard

Java programerski problemi

276 rešenih zadataka koji će vam pomoći u svakodnevnim izazovima

kompjuter biblioteka

Packt›

Izdavač:

kompjuter biblioteka

Obalskih radnika 4a
Beograd, Srbija
Tel: 011/2520272
e-pošta: kombib@gmail.com
veb-sajt: www.kombib.rs

Za izdavača:
Mihailo J. Šolajić, urednik

Autor: Angel Leonard

Prevod: Nemanja Lukić
Recezent: Miroslav Ristić
Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Apollo plus“,
Beograd
Tiraž: 500
Godina izdanja: 2025.

Broj knjige: 585
Izdanje: Prvo
ISBN: 978-86-7310-608-3

Naslov originala:

Java Coding Problems
Second Edition
ISBN 978-1-83763-394-4

Packt Publishing Ltd.
Birmingham, UK, packt.com

Java programerski problemi
Autorizovani prevod sa engleskog jezika.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.438JAVA(075.8)

ЛЕОНАРД, Ејнџел

Java programerski problemi : 276 rešenih zadataka koji će vam pomoći u svakodnevnim izazovima / Angel Leonard ; [prevod Nemanja Lukić]. - Izd. 1. - Beograd: Kompjuter biblioteka, 2025 (Beograd: Apollo plus). - XVI, 742 str. : ilustr. ; 24 cm. - (Kompjuter biblioteka; br. knj. 585)

Prevod dela: Java Coding Problems. - Tiraž 500. - O autoru: str. III.

ISBN 978-86-7310-608-3

а) Програмски језик „Java“

COBISS.SR-ID 161261833

SARADNICI

O AUTORU

Angel Leonard je glavni tehnološki strateg s više od 20 godina iskustva u Java ekosistemu. U svom svakodnevnom radu fokusira se na projektovanje i razvoj distribuiranih Java aplikacija koje omogućavaju robusne arhitekture, čist kod i visoke performanse. Strastveno se bavi podučavanjem, mentorstvom i tehničkim liderstvom.

Autor je nekoliko knjiga, video materijala i desetina članaka vezanih za Java tehnologije.

O RECENZENTIMA

Džordž Adams je viši softverski inženjer u kompaniji Microsoft, Java Champion i predsedavajući upravnog odbora radne grupe Eclipse Adoptium. Bio je jedan od osnivača projekta AdoptOpenJDK 2016. godine i od tada se nalazi na čelu zajednice. Odigrao je ključnu ulogu u premeštanju projekta u Eclipse Foundation. Džordž takođe doprinosi projektima Homebrew i Node.js Foundation, gde je glavni saradnik i aktivan član nekoliko radnih grupa.

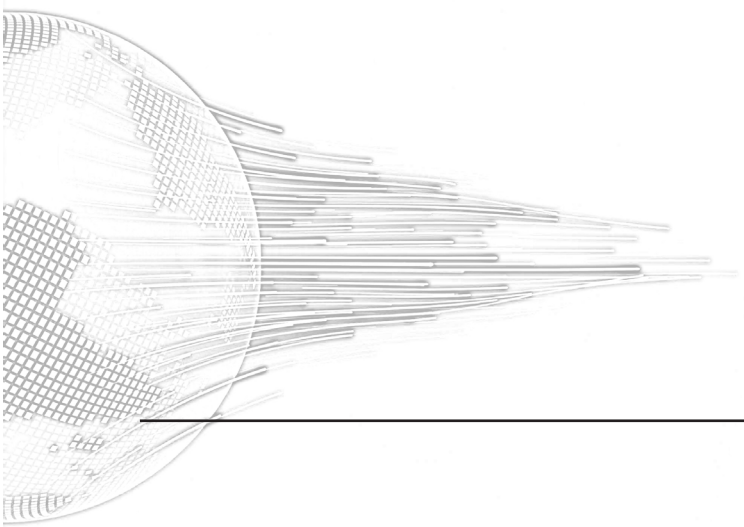
Ivar Grimstad se zalaže za Jakarta EE razvoj u organizaciji Eclipse Foundation. Java Champion i lider JUG zajednice sa sedištem u Švedskoj, doprinosi Jakarta EE specifikacijama i vodeći je PMC za Eclipse Enterprise for Java (EE4J). Takođe je vođa specifikacije za Jakarta MVC i predstavlja Eclipse Foundation u Izvršnom odboru JCP tela. Ivar je uključen u brojne projekte otvorenog koda i zajednice i redovan je govornik na međunarodnim programerskim konferencijama.

David Vlijmincx je Oracle ACE i veoma iskusan Java programer sa više od 8 godina iskustva u oblasti. Njegov glavni fokus je izgradnja skalabilnih Java aplikacija visokog kvaliteta. Strastven je u deljenju znanja, napisao je knjigu o migraciji Jakarta EE u oblak, redovno piše blogove i, takođe, govori na konferencijama.

O RECENZENTU ZA SRPSKO IZDANJE

Miroslav Ristić je redovni profesor na Prirodno-matematičkom fakultetu Univerziteta u Nišu, sa preko 25 godina iskustva u razvoju statističkog softvera. Posebno se ističe njegov rad na razvoju grafičkog korisničkog interfejsa R Commander za programski jezik R. Dugi niz godina recenzirao je značajan broj knjiga za izdavačku kuću Springer i časopis Journal of Applied Statistics. Od 2023. godine aktivno recenzira najaktuelnija izdanja izdavačke kuće "Kompjuter biblioteka". Nakon prevođenja, svako izdanje prolazi kroz njegovo stručno vrednovanje i recenziju prevoda, sa ciljem da se osigura da prevodi budu ne samo jasni, precizni i prilagođeni čitaocima, već i da održe visok kvalitet i stručnu relevantnost knjiga.

Kratak sadržaj



Sadržaj

POGLAVLJE 4

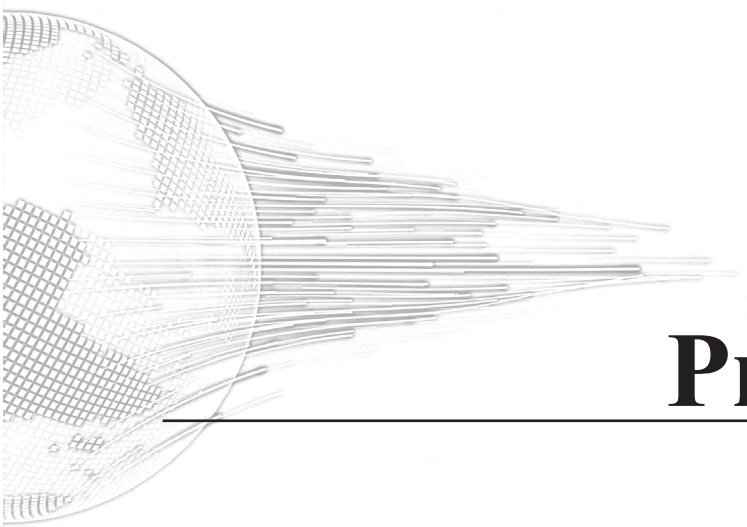
Zapisi i obrasci zapisa 197

POGLAVLJE 8

POGLAVLJE 9

POGLAVLJE 13

Socket API interfejs i Simple Web Server 679



Predgovor

Vrtoglavi razvoj između JDK verzija 12 i 21 donosi dve važne stvari: prvo, Java sada ima niz moćnih novih funkcionalnosti koje možete koristiti za rešavanje raznovrsnih savremenih problema; drugo, učenje savremenog jezika Java postaje sve kompleksnije.

Ova knjiga vam omogućava da objektivno pristupite rešavanju uobičajenih problema i objašnjava ispravne prakse i odluke koje treba da donesete u pogledu složenosti, performansi, čitljivosti i drugih aspekata.

Java programerski problemi, drugo izdanje, pomoći će vam da rešite svakodnevne zadatke i ispoštujete rokove dok, istovremeno, postajete veštiji i samostalniji Java programer. Možete se osloniti na više od 270 potpuno novih problema u ovom izdanju, koji pokrivaju najaktuelnije i najvažnije oblasti: rad sa niskama, brojevima, nizovima, kolekcijama, Foreign Function and Memory API interfejsom, strukturama podataka, rad s datumom i vremenom, usklađivanje obrazaca, skrivene i zatvorene klase, funkcionalno programiranje, virtuelne niti, strukturiranu konkurentnost, sakupljače otpada, dinamičke CDS arhive, Socket API interfejs i Simple Web Server.

Usavršite svoje veštine rešavanjem problema pažljivo osmišljenih da istaknu i prenesu osnovna znanja neophodna za svakodnevni rad. Drugim rečima, bilo da je vaš zadatak jednostavan, srednje složen ili veoma kompleksan, posedovanje ovog znanja nije opcija, već obaveza.

Do kraja knjige steći ćete snažno razumevanje Java koncepta i samopouzdanje da razvijate i birate prava rešenja za sve Java probleme s kojima se suočavate.

Iako je ova knjiga potpuno samostalna i nije vam potrebna dodatna literatura da biste izvukli maksimum iz nje, mnoge teme obrađene u ovoj knjizi istražene su i u *prvom izdanju* knjige *Java programerski problemi*. Ako ga još uvek niste pročitali, a želite dodatnu praksu, razmislite o tome da nabavite i to izdanje, jer donosi potpuno drugačiji skup Java problema.

Za koga je ova knjiga

Java programerski problemi, drugo izdanje, posebno je korisna programerima na granici početnog i srednjeg nivoa, koji žele da unaprede svoje znanje rešavanjem pravih problema. Međutim, problemi obrađeni u ovoj knjizi deo su svakodnevnice svakog Java programera, od početnika do iskusnog stručnjaka.

Da biste maksimalno iskoristili ovu knjigu potrebno je osnovno poznavanje Java jezika i osnove rada sa njim.

Šta sadrži ova knjiga

*Poglavlje 1: Tekstualni blokovi, lokalizacija, brojevi i matematika, o*buhvata 37 problema koji se odnose na 4 glavne teme: tekstualne blokove, lokalizaciju, brojeve i matematičke operacije.

*Poglavlje 2: Objekti, nepromenljivost, switch izrazi i usklađivanje obrazaca, s*adrži 30 problema koji obrađuju, između ostalog, neke manje poznate funkcionalnosti klase `java.util.Objects`, zanimljive aspekte nepromenljivosti, najnovije funkcionalnosti `switch` izraza, kao i detaljnu analizu izraza usklađivanja obrazaca: `instanceof` i `switch`.

*Poglavlje 3: Rad sa datumom i vremenom, s*adrži 20 problema koji pokrivaju različite teme vezane za rad sa datumom i vremenom. Problemi su uglavnom fokusirani na Calendar API interfejs i JDK 8 Date/Time API interfejs. Kod ovog poslednjeg obrađujemo manje poznate API interfejse, poput `ChronoUnit`, `ChronoField`, `IsoFields` i `TemporalAdjusters`.

Poglavlje 4: Zapisi i obrasci zapisa, obuhvata 19 problema koji detaljno pokrivaju zapise jezika Java uvedene u JDK 16 verziju (JEP 395) i obrasce zapisa, predstavljene kao probna funkcionalnost u JDK 19 verziji (JEP 405), kao druga probna funkcionalnost u JDK 20 verziji (JEP 432) i kao konačna funkcionalnost u JDK 21 verziji (JEP 440).

Poglavlje 5: Nizovi, kolekcije i strukture podataka, sadrži 24 problema koji pokrivaju tri glavne teme. Počinje problemima vezanim za novi Vector API interfejs posvećen paralelnoj obradi podataka. Nastavlja se strukturama podataka kao što su uže, preskačuća lista, K-dimenzionalno stablo, ziper, binomni hip, Fibonači hip, Hafmanovo stablo kodiranja i druge. Na kraju su razmotrena tri najpopularnija algoritma za spajanje.

Poglavlje 6: Ulazno-izlazne Java operacije: Filteri za deserijalizaciju specifični za kontekst, obuhvata 13 problema vezanih za procese serijalizacije/deserijalizacije u jeziku Java. Počinje klasičnim problemima serijalizacije/deserijalizacije objekata u formate `byte[]`, `String` i `XML`. Nastavlja se filterima za deserijalizaciju iz JDK 9 verzije za sprečavanje ranjivosti u deserijalizaciji, a završava filterima za deserijalizaciju specifičnim za kontekst iz JDK 17 verzije.

Poglavlje 7: Foreign (Function) Memory API interfejs, sadrži 28 problema koji pokrivaju Foreign Function Memory API interfejs i Foreign Linker API interfejs. Počinje klasičnim pristupima pozivanju stranih funkcija pomoću JNI API interfejsa i JNA/JNR biblioteka otvorenog koda. Nastavlja sa novim pristupima u okviru Projekta Panama, analizom API interfejsa kao što su *Arena*, *MemorySegment* i *MemoryLayout*. Na kraju obrađuje Foreign Function Memory API interfejs i alat Jextract za pozivanje stranih funkcija koje imaju različite tipove prototipova, uključujući funkcije povratnih poziva.

Poglavlje 8: Zatvorene i skrivene klase, obuhvata 13 problema. Prvih 11 pokrivaju zatvorene klase, uvedene u JDK 17 verziju za podršku zatvorenim hijerarhijama. Poslednja dva problema bave se skrivenim klasama, funkcionalnošću JDK 15 verzije koja omogućava radnim okvirima da kreiraju i koriste izvršne (dinamičke) klase skrivene of Java virtuelne mašine.

Poglavlje 9: Funkcionalno programiranje – proširivanje API interfejsa, obuhvata 24 problema vezana za funkcionalno programiranje. Počinje uvodom u metod `mapMulti()` iz JDK 16 verzije i nastavlja radom na predikatima, funkcijama i sakupljačima.

Poglavlje 10: Konkurentnost – virtuelne niti i strukturirana konkurentnost, sadrži 16 problema koji pružaju osnovni uvod u *virtuelne niti* i *strukturiranu konkurentnost*.

Poglavlje 11: Konkurentnost – virtuelne niti i strukturirana konkurentnost: detaljnije razmatranje, sadrži 18 problema koji detaljno obrađuju način rada virtuelnih niti i strukturirane konkurentnosti, kao i kako ih treba koristiti u vašim aplikacijama.

Poglavlje 12: Sakupljači smeća i dinamičke CDS arhive, sadrži 15 problema koji pokrivaju sakupljače otpada i deljenje podataka o klasama aplikacija (AppCDS).

Poglavlje 13: Socket API interfejs i Simple Web Server alat, obuhvata 11 problema vezanih za Socket API interfejs i 8 problema vezanih za Simple Web Server iz JDK 18 verzije. Prvih 11 problema obrađuje implementaciju aplikacija zasnovanih na priključcima (blokirajuće/neblokirajuće aplikacije, datagram aplikacije i aplikacije višesmernog emitovanja). Drugi deo poglavlja obrađuje Simple Web Server kao alat komandne linije.

Kako da najbolje iskoristite ovu knjigu

Treba da imate osnovno poznavanje jezika Java. Takođe, trebalo bi da instalirate sledeće:

- neko integrisano razvojno okruženje (preporučeno, ali ne i obavezno, izbor je Apache NetBeans 20.x: `(https://netbeans.apache.org/front/main/)`
- JDK 21 i najnoviju vwerziju alata Maven
- Dodatne biblioteke treće strane koje će biti potrebne za neke probleme i poglavlja (ništa previše zahtevno ni specijalno).

Preuzmite datoteke sa primerima koda

Imamo pakete kodova iz našeg bogatog kataloga knjiga i video materijala dostupnih na `https://github.com/PackPublishing/`. Proverite!

Preuzmite slike u boji

PDF dokument sa slikama u boji i snimcima ekrana/dijagramima koji su korišćeni u ovoj knjizi možete preuzeti sa adrese: `https://packt.link/gbp/9781837633944`.

Konvencije korišćene u ovoj knjizi

U ovoj knjizi korišćene su razne tekstualne konvencije.

`KodUTekstu` označava kod u tekstu, imena tabela u bazi podataka, imena direktorijuma, imena datoteka, ekstenzije datoteka, putanje, lažne URL adrese, unos korisnika i Twitter naloge.

Primer: „Na primer, dodajmo zapise `Patient` i `Appointment`."

Blok koda izgleda ovako:

```
if (osoba instanceof Stanar(String ime, Doktor dr)) {
  return "Kabinet " + dr.specijalnost() + ". Doktor: "
                    + dr.ime() + ", Stanar: " + ime;
}
```

Kada želimo da skrenemo pažnju na određeni deo koda, relevantne linije ili stavke su naglašene:

```
if (osoba instanceof Stanar(String ime, Doktor dr)) {
  return "Kabinet " + dr.specijalnost() + ". Doktor: "
                    + dr.ime() + ", Stanar: " + ime;
}
```

Svaki unos ili izlaz u komandnoj liniji se piše na sledeći način:

```
2023-02-07T05:26:17.374159500Z
2023-02-07T05:26:17.384811300Z
2023-02-07T05:26:17.384811300Z
```

Podebljan tekst označava novi termin, važnu reč ili pojmove koje vidite na ekranu. Na primer, reči u menijima ili dijalozima se pojavljuju u tekstu na sledeći način. Evo primera: „Kompajler prepoznaje **Java zapis** preko ključne reči **record**“.

> **Upozorenja** ili **važna obaveštenja**
> pojavljuju se ovako.

> **Saveti i** trikovi
> pojavljuju se ovako.

Kontaktirajte nas

Vaše povratne informacije su uvek dobrodošle.

Opšti komentari: pošaljite imejl na adresu `feedback@packtpub.com` i navedite naziv knjige u predmetu poruke. Ako imate pitanja o bilo kom aspektu ove knjige, pišite na adresu `questions@packtpub.com`.

Greške: iako smo se trudili da obezbedimo tačnost sadržaja, greške su moguće. Ako pronađete grešku u knjizi, bićemo vam zahvalni ako je prijavite. Posetite adresu `http://www.packtpub.com/submit-errata`, odaberite knjigu, kliknite na vezu za prijavu greške i unesite detalje.

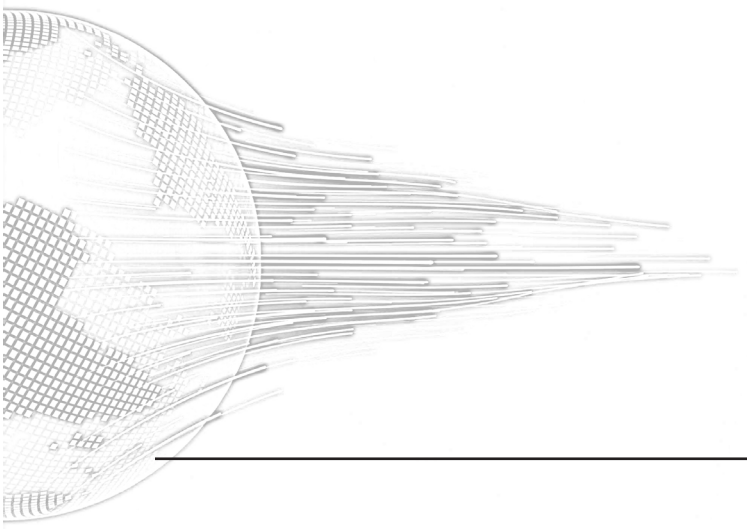
Piraterija: ako nađete nelegalne kopije naših dela u bilo kom obliku na internetu, bićemo vam zahvalni ako nam date adresu lokacije ili naziv veb sajta. Kontaktirajte nas na adresi `copyright@packtpub.com` sa vezom do spornog materijala.

Ako želite da budete autor: ako ste stručnjak za neku oblast i želite da pišete ili da date doprinos knjizi, posetite adresu `http://authors.packtpub.com`.

Postanite član Kompjuter biblioteke

Kupovinom jedne naše knjige stekli ste pravo da postanete član Kompjuter biblioteke. Kao član možete da kupujete knjige u pretplati sa 40% popusta i učestvujete u akcijama kada ostvarujete popuste na sva naša izdanja. Potrebno je samo da se prijavite preko formulara na našem sajtu.
Link za prijavu: kombib.rs/kblista.php

Skenirajte QR kod
registrujte knjigu
i osvojite nagradu



1

Tekstualni blokovi, lokalizacija, brojevi i matematika

Ovo poglavlje sadrži 37 problema koji se tiču 4 glavne teme: tekstualni blokovi, lokalizacija, brojevi i matematičke operacije. Počećemo od tekstualnih blokova (elegantnim višelinijskim niskama uvedenim u JDK 13 verziju (JEP 355, probna)/JDK 15 verziju (JEP 378, konačna), zatim, prelazimo na probleme za kreiranje Java lokalizacija, uključujući lokalizovane lokalitete (`ofLocalizedPattern()` iz JDK 19 verzije), a završavamo problemima o brojevima i matematičkim operacijama, kao što su Vavilonska metoda za izračunavanje kvadratnog korena i različiti ekstremni slučajevi prekoračenja rezultata. Poslednji deo poglavlja posvećen je novom API interfejsu za generatore pseudoslučajnih brojeva u JDK 17 verziji (JEP 356, konačna).

Do kraja ovog poglavlja, bićete upoznati sa svim novim i interesantnim JDK funkcionalnostima koje su dodate i koje se odnose na ove četiri teme.

Napomena

U ovoj knjizi nailazićete na reference na prvo izdanje. Uloga tih referenci je da vam pruže najbolje izvore za dalju literaturu o određenim temama. Ovo izdanje možete uspešno pratiti čak i ako niste pročitali prvo izdanje niti planirate da ga čitate.

Problemi

Iskoristite sledeće probleme da testirate svoje veštine rada sa niskama, Java lokalizacijama i programiranja u matematičkim ekstremnim slučajevima. Toplo preporučujem da pokušate da rešite svaki zadatak pre nego što se okrenete rešenjima i preuzmete primere programa:

1. **Kreiranje višelinijske SQL, JSON i HTML niske**: napišite program koji deklariše višelinijsku nisku (na primer, SQL, JSON i HTML niske).
2. **Primer upotrebe graničnika u tekstualnim blokovima**: napišite program koji korak po korak pokazuje kako graničnici u tekstualnim blokovima utiču na rezultujuću nisku.

3. **Rad sa uvlačenjem u tekstualnim blokovima**: napišite program koji prikazuje različite tehnike za uvlačenje teksta u tekstualnim blokovima. Objasnite značenje *nepotrebnih* i *neophodnih* praznina.
4. **Uklanjanje nepotrebnih praznina u tekstualnim blokovima**: istaknite glavne korake algoritma koji kompajler koristi za uklanjanje *nepotrebnih* praznina iz tekstualnog bloka.
5. **Upotreba tekstualnih blokova radi bolje čitljivosti**: napišite program koji kreira nisku koja izgleda kao tekstualni blok (višelinijska niska) ali se ponaša kao jednolinijska konstantna niska.
6. **Izbegavanje navodnika i završetaka linija u tekstualnim blokovima**: napišite program koji pokazuje kako se rukuje Java specijalnim sekvencama karaktera (uključujući znakove navoda, `\"` i završetke linija `\n` i `\r`) u tekstualnim blokovima.
7. **Programsko prevođenje specijalnih sekvenci karaktera**: napišite program koji omogućava programsko prevođenje specijalnih sekvenci u tekstualnim blokovima. Pretpostavite da imate tekstualni blok koji sadrži specijalne sekvence i da morate da ga prosledite funkciji koja zahteva nisku bez takvih sekvenci.
8. **Formatiranje tekstualnih blokova sa promenljivama i izrazima**: napišite program koji prikazuje nekoliko tehnika formatiranja tekstualnih blokova pomoću promenljivih i izraza. Komentarišite svaku tehniku sa aspekta čitljivosti i pružite upoređivanja pomoću alata **Java Microbenchmark Harness** (**JMH**).
9. **Dodavanje komentara u tekstualne blokove**: objasnite kako se komentari dodaju u tekstualni blok.
10. **Mešanje običnih konstantnih niski sa tekstualnim blokovima**: napišite program koji kombinuje obične konstantne niske sa tekstualnim blokovima, na primer, spajanjem. Takođe, proverite da li su obična konstantna niska i tekstualni blok jednaki ako imaju isti sadržaj.
11. **Mešanje regularnih izraza sa tekstualnim blokovima**: napišite primer koji meša regularne izraze sa imenovanim grupama i tekstualne blokove.
12. **Provera da li su dva tekstualna bloka izomorfna**: napišite program koji proverava da li su dva tekstualna bloka izomorfna. Dve niske su izomorfne ako se svaki karakter prve niske može jednoznačno preslikati na svaki karakter druge niske (na primer, „xxyznnxiz" i „aavurraqu" su izomorfne).
13. **Spajanje niski naspram klase StringBuilder**: napravite JMH upoređivanje za poređenje spajanja niski (putem operatora „+") sa pristupom u kom se koristi `StringBuilder`.
14. **Konvertovanje celobrojne vrednosti u nisku**: napišite program koji nudi nekoliko uobičajenih tehnika za pretvaranje celobrojne vrednosti u nisku. Takođe, za predložena rešenja obezbedite JMH upoređivanje.
15. **Predstavljanje šablona niski**: objasnite i ilustrujte upotrebu funkcionalnosti šablona niski uvedenih u JDK 21 verziju (JEP 430, probna).
16. **Pisanje prilagođenog procesora šablona**: predstavite API interfejs za pisanje korisnički definisanog procesora šablona. Zatim, obezbedite nekoliko primera prilagođenih procesora šablona.
17. **Kreiranje lokalizacije**: napišite program koji prikazuje različite pristupe kreiranju lokalizacija. Takođe, kreirajte *jezičke opsege* i *liste prioriteta jezika*.

18. **Prilagođavanje lokalizovanih formata datuma i vremena**: napišite program koji ilustruje upotrebu prilagođenih lokalizovanih formata datuma i vremena.
19. **Vraćanje uvek-strogih semantika za pokretni zarez**: objasnite šta je `strictfp` modifikator i kako/gde se koristi u Java aplikacijama.
20. **Izračunavanje apsolutne vrednosti za int/long tipove i prekoračenje rezultata**: napišite program koji ilustruje ekstremni slučaj gde primena apsolutne vrednosti na `int`/`long` tipovima rezultira prekoračenjem rezultata. Takođe, pronađite rešenje za ovaj problem.
21. **Izračunavanje količnika argumenata i prekoračenje rezultata**: napišite program koji ilustruje ekstremni slučaj u kom izračunavanje količnika argumenata dovodi do prekoračenja rezultata. Takođe, nađite rešenje za ovaj problem.
22. I**zračunavanje najveće/najmanje vrednosti koja je manja/veća ili jednaka algebarskom količniku**: napišite program koji koristi metode iz `java.lang.Math` klase za izračunavanje najveće/najmanje vrednosti manje/veće ili jednake algebarskom količniku. Ne zaboravite da pokrijete i slučaj prekoračenja rezultata u ekstremnom slučaju.
23. **Dobijanje celobrojnog i decimalnog dela iz vrednosti tipa double**: napišite program koji prikazuje nekoliko tehnika za dobijanje celobrojnog i decimalnog dela broja tipa `double`.
24. **Provera da li je broj tipa double ceo broj**: napišite program koji prikazuje nekoliko pristupa za proveru da li je broj tipa `double` ceo broj. Pored toga, odredite JMH upoređivanje za predložena rešenja.
25. **Rad sa (ne)označenim celobrojnim vrednostima u kratkim crtama**: objasnite i prikažite u kodu upotrebu označenih/neoznačenih celih brojeva u jeziku Java.
26. **Vraćanje modula sa zaokruživanjem prema dole/gore**: definišite modul sa zaokruživanjem prema dole/gore na osnovu operacija `floor/ceil`, i prikažite rezultat u linijama koda.
27. **Prikupljanje svih prostih faktora datog broja**: prost broj je broj deljiv samim sobom i jedinicom (na primer, 2, 3 i 5 su prosti brojevi). Napišite program koji prikuplja sve proste činioce datog pozitivnog broja.
28. **Izračunavanje kvadratnog korena broja Vavilonskim metodom**: objasnite Vavilonski metod za izračunavanje kvadratnog korena, razradite algoritam za ovaj metod i napišite kod zasnovan na ovom algoritmu.
29. **Zaokruživanje broja tipa float na određen broj decimala**: napišite program koji sadrži nekoliko pristupa za zaokruživanje broja tipa `float` na određen broj decimala.
30. **Ograničavanje vrednosti između minimalne i maksimalne**: pružite rešenje za ograničavanje zadate vrednosti između date minimalne i maksimalne vrednosti.
31. **Množenje dva cela broja bez petlji, operacija množenja i deljenja, bitovskih operacija i operatora**: napišite program koji množi dva cela broja bez upotrebe petlji, množenja, bitovskih operacija, deljenja i operatora. Na primer, počnite od *specijalne formule za binomni proizvod*.
32. **TAU vrednost**: objasnite značenje TAU vrednosti u geometriji/trigonometriji i napišite program koji rešava sledeći problem: obim kruga je 21,33 cm. Koliki je poluprečnik kruga?
33. **Izbor generatora pseudoslučajnih brojeva**: pružite kratko razmatranje o novom API interfejsu za generisanje pseudoslučajnih brojeva uvedenom u JDK 17

verziju (JEP 356, konačna). Takođe, prikažite različite tehnike za izbor generatora pseudoslučajnih brojeva.

34. **Popunjavanje niza tipa long pseudoslučajnim brojevima**: napišite program koji popunjava niz tipa `long` pseudoslučajnim brojevima na paralelan i ne paralelan način.
35. **Kreiranje toka generatora pseudoslučajnih brojeva**: napišite program koji kreira tok pseudoslučajnih brojeva i tok generatora pseudoslučajnih brojeva.
36. **Dobijanje nasleđenih generatora pseudoslučajnih brojeva iz novih u JDK 17 verziji**: napišite program koji instancira stari generator pseudoslučajnih brojeva (na primer, `Random`) koji može da delegira pozive metoda novim generatorom `RandomGenerator` u JDK 17 verziji.
37. **Korišćenje generatora pseudoslučajnih brojeva u višenitnim okruženjima**: objasnite i prikažite upotrebu generatora pseudoslučajnih brojeva u višenitnom okruženju (na primer, pomoću interfejsa `ExecutorService`).

Sledeći odeljci opisuju rešenja za prethodne probleme. Zapamtite da obično ne postoji samo jedan ispravan način za rešavanje određenog problema. Takođe, imajte na umu da objašnjenja obuhvataju samo najzanimljivije i najvažnije detalje potrebne za rešavanje problema. Preuzmite primere rešenja kako biste videli dodatne detalje i eksperimentisali sa programima na adresi *https://github.com/PacktPublishing/Java-Coding-Problems-Second-Edition/tree/main/Chapter01)*.

Programi i slike su prilagođeni našem govornom području kako bi bili razumljiviji i lakši za upotrebu čitaocima u našem regionu, kao i kako bi se olakšalo razumevanje i primena materijala.

1. Kreiranje višelinijske SQL, JSON i HTML niske

Hajde da razmotrimo sledeću višelinijsku SQL nisku:

```
UPDATE "šema"."kancelarija"
SET ("adresa_prva", "adresa_druga", "telefon") =
  (SELECT "šema"."zaposleni"."ime",
          "šema"."zaposleni"."prezime", ?
   FROM "šema"."zaposleni"
   WHERE "šema"."zaposleni"."radno_mesto" = ?
```

Kao što je poznato, pre JDK 8 verzije mogli smo da SQL upit predstavimo kao Java nisku (konstantnu nisku) na nekoliko načina.

Pre JDK 8 verzije

Verovatno najčešći pristup oslanja se na jednostavno spajanje uz poznati operator „+". Na taj način dobijamo prikaz niske u više linija, kao što sledi:

```
String sql =
 "UPDATE \"šema\".\"kancelarija\"\n"
+ "SET (\"adresa_prva\", \"adresa_druga\", \"telefon\") =\n"
+ "  (SELECT \"šema\".\"zaposleni\".\"ime\",\n"
```

```
+ "          \"šema\".\"zaposleni\".\"prezime\", ?\n"
+ "   FROM \"šema\".\"zaposleni\"\n"
+ "   WHERE \"šema\".\"zaposleni\".\"radno_mesto\" = ?";
```

Kompajler bi trebalo da je (a uglavnom jeste) dovoljno pametan da interno transformiše operacije „+" u instancu klase `StringBuilder` ili `StringBuffer` i da koristi metod `append()` za kreiranje konačne niske. Međutim, možemo direktno da koristimo klasu `StringBuilder` (nije bezbedna u višenitnom okruženju) ili `StringBuffer` (bezbedna u višenitnom okruženju), kao u sledećem primeru:

```
StringBuilder sql = new StringBuilder();
sql.append("UPDATE \"šema\".\"kancelarija\"\n")
   .append("SET ...\n")
   .append("  (SELECT...\n")
   ...
```

Drugi pristup (obično manje popularan od prethodna dva) koristi metod `String.concat()`. Ovo je nepromenljiva operacija koja u suštini dodaje datu nisku na kraj trenutne. Na kraju vraća novu kombinovanu nisku. Pokušaj dodavanja vrednosti `null` rezultira izuzetkom `NullPointerException` (u prethodna dva primera, `null` vrednosti mogu biti dodate bez generisanja izuzetaka). Lančani pozivi metoda `concat()` omogućavaju predstavljanje niski u više linija, kao u sledećem primeru:

```
String sql = "UPDATE \"šema\".\"kancelarija\"\n"
  .concat("SET...\n")
  .concat("  (SELECT...\n")
  ...
```

Pored toga, imamo i metod `String.format()`. Jednostavnom upotrebom specifikatora formata `%s` možemo da spojimo više niski (uključujući i `null` vrednosti) u višelinijsku nisku, kao što sledi:

```
String sql = String.format("%s%s%s%s%s%s",
  "UPDATE \"šema\".\"kancelarija\"\n",
  "SET ...\n",
  "  (SELECT ...\n",
  ...
```

Iako su ovi pristupi i dalje popularni, pogledajmo šta JDK 8 verzija nudi po ovom pitanju.

Od JDK 8 verzije

Od JDK 8 verzije, možemo da koristimo metod `String.join()` za predstavljanje višelinijskih niski. Ovaj metod je takođe specijalizovan za spajanje niski i omogućava nam laku čitljivost primera. Kako? Ovaj metod za prvi argument prihvata graničnik i koristi ga između niski koje će biti spajane. Dakle, ako smatramo da je `\n` naš graničnik linije, možemo ga navesti samo jednom, kao što sledi:

```
String sql = String.join("\n"
 ,"UPDATE \"šema\".\"kancelarija\""
 ,"SET (\"adresa_prva\", \"adresa_druga\", \"telefon\") ="
 ,"  (SELECT \"šema\".\"zaposleni\".\"ime\","
 ,"          \"šema\".\"zaposleni\".\"prezime\", ?"
```

```
 ,"   FROM \"šema\".\"zaposleni\""
 ,"   WHERE \"šema\".\"zaposleni\".\"radno_mesto\" = ?;");
```

Pored metoda `String.join()`, JDK 8 verzija takođe poseduje klasu `java.util.StringJoiner`. Klasa `StringJoiner` podržava graničnik (kao i `String.join()`), ali takođe podržava prefiks i sufiks. Predstavljanje naše višelinijske SQL niske ne zahteva prefiks/sufiks; stoga graničnik ostaje omiljena funkcionalnost:

```
StringJoiner sql = new StringJoiner("\n");
sql.add("UPDATE \"šema\".\"kancelarija\"")
   .add("SET (\"adresa_prva\", ..., \"telefon\") =")
   .add("  (SELECT \"šema\".\"zaposleni\".\"ime\",")
   ...
```

Na kraju, ne možemo govoriti o JDK 8 verziji, a da ne pomenemo moćni Stream API interfejs. Konkretno, zanima nas sakupljač `Collectors.joining()`. Ovaj sakupljač funkcioniše kao `String.join()`, a u našem slučaju izgleda ovako:

```
String sql = Stream.of(
 "UPDATE \"šema\".\"kancelarija\"",
 "SET (\"adresa_prva\", \"adresa_druga\", \"telefon\") =",
 "  (SELECT \"šema\".\"zaposleni\".\"ime\",",
 "          \"šema\".\"zaposleni\".\"prezime\", ?",
 "   FROM \"šema\".\"zaposleni\"",
 "   WHERE \"šema\".\"zaposleni\".\"radno_mesto\" = ?;")
 .collect(Collectors.joining(String.valueOf("\n")));
```

Svi prethodni primeri imaju nekoliko zajedničkih nedostataka. Najvažniji od njih je taj što nijedan od ovih primera ne predstavlja pravu višelinijsku konstantnu nisku, a čitljivost ozbiljno ispašta zbog specijalnih sekvenci karaktera i dodatnih navodnika potrebnih za svaku liniju. Srećom, počevši od JDK 13 verzije (kao buduća probna funkcionalnost) i zaključno s JDK 15 verzijom (kao konačna funkcionalnost), novi tekstualni blokovi postaju standard za predstavljanje višelinijskih niski. Pogledajmo kako.

Predstavljanje tekstualnih blokova (JDK 13/15)

JDK 13 verzija (JEP 355) uvodi probnu funkcionalnost koja ima za cilj podršku za višelinijske konstantne niske. Tokom dve verzije, u JDK 15 verziji (JEP 378), funkcionalnost tekstualnih blokova postaje konačna i trajna za upotrebu. Ali, to je dovoljno istorije; brzo pogledajmo kako tekstualni blokovi oblikuju našu višelinijsku SQL nisku:

```
String sql="""
           UPDATE "šema"."kancelarija"
           SET ("adresa_prva", "adresa_druga", "telefon") =
             (SELECT "šema"."zaposleni"."ime",
                     "šema"."zaposleni"."prezime", ?
              FROM "šema"."zaposleni"
              WHERE "šema"."zaposleni"."radno_mesto" = ?""";
```

Ovo je zaista sjajno, zar ne? Odmah vidimo da je čitljivost našeg SQL upita vraćena, i to bez komplikacija sa graničnicima, završecima linija i spajanjima. Tekstualni blok je koncizan, jednostavan za ažuriranje i lako razumljiv. Dodatan kod u našoj SQL niski praktično

ne postoji, a Java kompajler će se pobrinuti da niske bude kreirana na najpredvidljiviji način. Evo još jednog primera koji sadrži deo JSON informacija:

```
String json = """
              {
                "objekat": {
                  "praćenje_grešaka": "uključeno",
                  "prozor": {
                    "naslov": "Objekat 1",
                    "ime": "pozadinski_prozor"
                  },
                  "slika": {
                    "izvor": "slike\\sw.png"
                  },
                  "tekst": {
                    "podaci": "Klikni ovde",
                    "veličina": 39
                  }
                }
              }""";
```

Šta kažete na predstavljanje dela HTML koda kao tekstualnog bloka? Naravno, evo ga:

```
String html = """
              <table>
                <tr>
                  <thcolspan="2">Ime i prezime</th>
                  <th>Broj godina</th>
                </tr>
                <tr>
                  <td>Petar</td>
                  <td>Petrović</td>
                  <td>22</td>
                </tr>
              <table>""";
```

Kakva je sintaksa tekstualnih blokova?

Upotreba sintakse tekstualnih blokova

Sintaksa tekstualnih blokova je veoma jednostavna. Bez komplikacija, bez suvišnih elemenata – samo dve stvari treba imati na umu:

- Tekstualni blok mora početi sa """ (tri znaka dvostrukih navodnika) i novom linijom. Tu konstrukciju nazivamo *početni graničnik*.
- Tekstualni blok mora se završiti sa """ (tri znaka dvostrukih navodnika). Graničnik """ može biti na sopstvenoj liniji (kao nova linija) ili na kraju poslednje linije teksta (kao u našem primeru). Tu konstrukciju nazivamo *završni graničnik*. Međutim, postoji semantička razlika između ova dva pristupa (koja će biti razmotrena u narednom problemu).

U ovom kontekstu, sledeći primeri su sintaktički ispravni:

```
String tb = """
            Ja sam tekstualni blok""";
String tb = """
            Ja sam tekstualni blok
            """;

String tb = """

            Ja sam tekstualni blok""";

String tb = """

            Ja sam tekstualni blok
            """;

String tb = """

            Ja sam tekstualni blok

            """;
```

S druge strane, sledeći primeri su neispravni i izazivaju greške prilikom kompajliranja:

```
String tb = """Ja sam tekstualni blok""";

String tb = "Ja sam tekstualni blok""";

String tb = """Ja sam tekstualni blok";

String tb = ""Ja sam tekstualni blok""";

String tb = """Ja sam tekstualni blok"";

String tb = ""Ja sam tekstualni blok
            """;
```

U svakom slučaju, imajte na umu sledeću najbolju praksu.

Važna napomena

Iz prethodnih primera koda možemo izvući najbolju praksu za tekstualne blokove: koristite tekstualne blokove samo kada imate višelinijsku nisku; ako niska može da stane u jednu liniju koda (kao u prethodnim kodovima), onda koristite običnu konstantnu nisku, jer tekstualni blokovi u tom slučaju ne donose značajnu prednost.

U priloženom kodu možete vežbati sve primere ovog problema na SQL, JSON i HTML primerima.

> **Važna napomena**
>
> Za podršku biblioteka treće strane, razmotrite Apache Commons, `StringUtils.join()` i `Joiner.on()` biblioteke Guava.

U nastavku ćemo se fokusirati na rad sa graničnicima tekstualnih blokova.

2. Primer upotrebe graničnika u tekstualnim blokovima

Setite se iz prethodnog problema, *Kreiranje višelinijske SQL, JSON i HTML niske*, da je tekstualni blok sintaktički ograničen početnim i završnim graničnicima, predstavljenim sa tri znaka dvostrukih navodnika, `"""`.

Najbolji prikaz upotrebe ovih graničnika sastoji se iz tri jednostavna koraka: uzeti primer, pregledati rezultat i doneti zaključak. S tim na umu, počnimo sa primerom koji oponaša neke od primera iz JEP dokumenta:

```
String sql= """
            UPDATE "šema"."kancelarija"
            SET ("adresa_prva", "adresa_druga", "telefon") =
              (SELECT "šema"."zaposleni"."ime",
                      "šema"."zaposleni"."prezime", ?
               FROM "šema"."zaposleni"
               WHERE "šema"."zaposleni"."radno_mesto" = ?)""";
```

Prateći te primere, sadržaj treba poravnati sa početnim graničnikom. Verovatno je da ovaj stil poravnanja nije dosledan ostatku našeg koda i možda nije dobra praksa. Šta će se dogoditi sa sadržajem tekstualnog bloka ako promenimo ime promenljive `sql` u, recimo, `azurirajSql`, `azurirajKancelarijuRadnimMestomZaposlenog` ili nešto slično? Očigledno je da će, radi očuvanja poravnanja, sadržaj biti još više pomeren udesno. Srećom, sadržaj možemo pomeriti ulevo bez uticaja na krajnji rezultat, na sledeći način:

```
String sql = """
  UPDATE "šema"."kancelarija"
  SET ("adresa_prva", "adresa_druga", "telefon") =
    (SELECT "šema"."zaposleni"."ime",
            "šema"."zaposleni"."prezime", ?
     FROM "šema"."zaposleni"
     WHERE "šema"."zaposleni"."radno_mesto" = ?""";
```

Pomeranje početnih/završnih graničnika udesno neće uticati na rezultujuću nisku. Malo je verovatno da ćete imati dobar razlog da to uradite, ali radi kompletnosti, sledeći primer daje isti rezultat kao prethodna dva primera:

```
String sql =                                              """
  UPDATE "šema"."kancelarija"
  SET ("adresa_prva", "adresa_druga", "telefon") =
    (SELECT "šema"."zaposleni"."ime",
            "šema"."zaposleni"."prezime", ?
     FROM "šema"."zaposleni"
     WHERE "šema"."zaposleni"."radno_mesto" = ?       """;
```

Sada, hajde da pogledamo nešto zanimljivije. Početni graničnik ne prihvata sadržaj na istoj liniji, dok završni graničnik stoji desno, na kraju sadržaja. Međutim, šta će se dogoditi ako pomerimo završni graničnik na sopstvenu liniju, kao u sledeća dva primera?

```
String sql= """
            UPDATE "šema"."kancelarija"
            SET ("adresa_prva", "adresa_druga", "telefon") =
              (SELECT "šema"."zaposleni"."ime",
                      "šema"."zaposleni"."prezime", ?
               FROM "šema"."zaposleni"
               WHERE "šema"."zaposleni"."radno_mesto" = ?
            """;
```

```
String sql= """
  UPDATE "šema"."kancelarija"
  SET ("adresa_prva", "adresa_druga", "telefon") =
    (SELECT "šema"."zaposleni"."ime",
            "šema"."zaposleni"."prezime", ?
     FROM "šema"."zaposleni"
     WHERE "šema"."zaposleni"."radno_mesto" = ?
  """;
```

Ovog puta, dobijena niska sadrži novi red na kraju sadržaja. Pogledajte sledeću sliku (tekst -- `PRE TEKSTUALNOG BLOKA` -- i -- `POSLE TEKSTUALNOG BLOKA` -- samo su vodiči dodati pomoću metoda `System.out.println()` kako bi se pomoglo u razgraničavanju samog tekstualnog bloka; oni nisu neophodni i nisu deo tekstualnog bloka):

```
-- PRE TEKSTUALNOG BLOKA --
UPDATE "šema"."kancelarija"
SET ("adresa_prva", "adresa_druga", "telefon") =
  (SELECT "šema"."zaposleni"."ime",
          "šema"."zaposleni"."prezime", ?
   FROM "šema"."zaposleni"
   WHERE "šema"."zaposleni"."radno_mesto" = ?
-- POSLE TEKSTUALNOG BLOKA --
```

A

```
-- PRE TEKSTUALNOG BLOKA --
UPDATE "šema"."kancelarija"
SET ("adresa_prva", "adresa_druga", "telefon") =
  (SELECT "šema"."zaposleni"."ime",
          "šema"."zaposleni"."prezime", ?
   FROM "šema"."zaposleni"
   WHERE "šema"."zaposleni"."radno_mesto" = ?

-- POSLE TEKSTUALNOG BLOKA --
```

B

Slika 1.1: Pomeranje završnog graničnika na sopstvenu liniju, vertikalno poravnatu sa početnim graničnikom

Na levoj slici (*A*), završni graničnik je na kraju sadržaja. Međutim, na desnoj slici (*B*), pomerili smo završni graničnik na sopstvenu liniju, i kao što možete videti, dobijena niska je obogaćena novim redom na kraju.

> **Važna napomena**
>
> Postavljanje završnog graničnika na sopstvenu liniju dodaje novi red u dobijenu nisku. Takođe, obratite pažnju da vertikalno poravnanje početnog graničnika, sadržaja i završnog graničnika uz levu marginu može kasnije dovesti do dodatnog posla. Ako se ime promenljive promeni, biće potrebna ručna promena poravnanja kako bi se održalo ovo poravnanje.

Stoga, obratite pažnju na način postavljanja završnog graničnika.

Da li vam je ovo čudno? Pa, to nije sve! U prethodnom primeru, završni graničnik je bio postavljen na sopstvenu liniju, ali vertikalno poravnat sa početnim graničnikom. Hajde da napravimo korak dalje i pomerimo završni graničnik ulevo, kao u sledećem primeru:

```
String sql= """
            UPDATE "šema"."kancelarija"
            SET ("adresa_prva", "adresa_druga", "telefon") =
              (SELECT "šema"."zaposleni"."ime",
                      "šema"."zaposleni"."prezime", ?
               FROM "šema"."zaposleni"
               WHERE "šema"."zaposleni"."radno_mesto" = ?
""";
```

Sledeća slika otkriva efekat ove akcije:

```
-- PRE TEKSTUALNOG BLOKA --
UPDATE "šema"."kancelarija"
SET ("adresa_prva", "adresa_druga", "telefon") =
  (SELECT "šema"."zaposleni"."ime",
          "šema"."zaposleni"."prezime", ?
   FROM "šema"."zaposleni"
   WHERE "šema"."zaposleni"."radno_mesto" = ?

-- POSLE TEKSTUALNOG BLOKA --
```

A

```
-- PRE TEKSTUALNOG BLOKA --
            UPDATE "šema"."kancelarija"
            SET ("adresa_prva", "adresa_druga", "telefon") =
              (SELECT "šema"."zaposleni"."ime",
                      "šema"."zaposleni"."prezime", ?
               FROM "šema"."zaposleni"
               WHERE "šema"."zaposleni"."radno_mesto" = ?

-- POSLE TEKSTUALNOG BLOKA --
```

B

Slika 1.2: Pomeranje završnog graničnika na sopstvenu liniju i njegovo pomeranje ulevo

Na levoj slici (*A*), završni graničnik se nalazi na sopstvenoj liniji i poravnat je sa početnim graničnikom. Na desnoj slici (*B*), imamo efekat prethodnog koda. Pomeranje završnog graničnika ulevo rezultira dodatnim uvlačenjem sadržaja udesno. Dodatno uvlačenje zavisi od toga koliko pomerimo završni graničnik ulevo.

> **Važna napomena**
>
> Postavljanje završnog graničnika na sopstvenu liniju i njegovo pomeranje ulevo dodaće novi red i dodatno uvlačenje u krajnju nisku.

S druge strane, ako završni graničnik pomerimo na sopstvenu liniju i pomerimo ga udesno, to neće uticati na krajnju nisku:

```
String sql= """
            UPDATE "šema"."kancelarija"
            SET ("adresa_prva", "adresa_druga", "telefon") =
              (SELECT "šema"."zaposleni"."ime",
                      "šema"."zaposleni"."prezime", ?
               FROM "šema"."zaposleni"
               WHERE "šema"."zaposleni"."radno_mesto" = ?
                                              """;
```

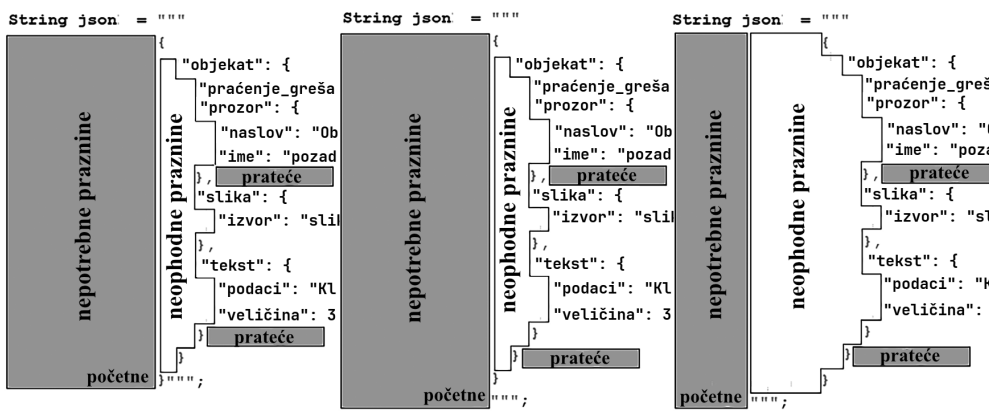
Ovaj kod dodaje novi red u krajnju nisku, ali ne utiče na uvlačenje. Da biste bolje razumeli ponašanje početnih/završnih graničnika, morate istražiti sledeći problem.

3. Rad sa uvlačenjem u tekstualnim blokovima

Uvlačenje u tekstualnim blokovima je lako razumeti ako jasno razlikujemo dva pojma:

- *Nepotrebne* (ili *nebitne*) praznine – predstavljaju praznine bez značenja, koji nastaju usled formatiranja koda (*početne* praznine koje obično dodaje integrisano razvojno okruženje) ili koje su namerno/slučajno dodate na kraju teksta (*prateće* praznine).
- *Neophodne* praznine – predstavljaju praznine koje eksplicitno dodajemo i koje imaju značenje za krajnju nisku.

Na *slici 1.3* možete videti razliku između nepotrebnih i neophodnih praznina u JSON tekstualnom bloku:



Slika 1.3: Razlika između nepotrebnih i neophodnih praznina u JSON tekstualnom bloku

Levo na slici, prikazana je razlika između nepotrebnih i neophodnih praznina kada je završni graničnik postavljen na kraju sadržaja. Na srednjoj slici, završni graničnik je pomeren na sopstvenu liniju, dok je desno na slici dodatno pomeren ulevo.

Nepotrebne (nebitne) praznine automatski uklanja Java kompajler. Kompajler uklanja sve nepotrebne prateće praznine (kako bi osigurao isti prikaz u različitim uređivačima teksta, koji mogu automatski ukloniti prateće praznine) i koristi poseban interni algoritam (koji je detaljno razmotren u sledećem problemu) za identifikaciju i uklanjanje početnih nepotrebnih praznina. Takođe, važno je napomenuti da linija koja sadrži završni graničnik uvek ulazi u ovu proveru (što je poznato kao *politika značajne prateće linije*).

Neophodne praznine se čuvaju u krajnjoj niski. U suštini, kao što možete zaključiti iz prethodnih ilustracija, nepotrebne praznine se mogu dodati na dva načina:

- Pomeranjem završnog graničnika ulevo (kada je ovaj graničnik na sopstvenoj liniji).
- Pomeranjem sadržaja udesno (eksplicitnim dodavanjem praznina ili pomoćnim metodima za kontrolu uvlačenja).

Pomeranje završnog graničnika i/ili sadržaja

Počnimo sledećim kodom:

```
String json = """
--------------{
--------------++"objekat": {
--------------++++"praćenje_grešaka": "uključeno",
--------------++++"prozor": {
--------------++++++"naslov": "Objekat 1",
--------------++++++"ime": "pozadinski_prozor"
--------------++++},
--------------++++"slika": {
--------------++++++"izvor": "slike\\sw.png"
--------------++++},
--------------++++"tekst": {
--------------++++++"podaci": "Klikni ovde",
--------------++++++"veličina": 39
--------------++++}
--------------++}
--------------}""";
```

Praznine označene znakom „–” predstavljaju nepotrebne početne praznine (nema nepotrebnih pratećih praznina), dok praznine označene znakom „+” predstavljaju neophodne praznine koje će se videti u krajnjoj niski. Ako pomerimo ceo sadržaj udesno dok je završni graničnik na kraju sadržaja, eksplicitno dodate praznine se smatraju nepotrebnim i kompajler ih uklanja:

```
String json = """
----------------------{
----------------------++"objekat": {
----------------------++++"praćenje_grešaka": "uključeno",
----------------------++++"prozor": {
```

```
----------------------++++++"naslov": "Objekat 1",
----------------------++++++"ime": "pozadinski_prozor"
----------------------++++},
----------------------++++"slika": {
----------------------++++++"izvor": "slike\\sw.png"
----------------------++++},
----------------------++++"tekst": {
----------------------++++++"podaci": "Klikni ovde",
----------------------++++++"veličina": 39
----------------------++++}
----------------------++}
----------------------}""";
```

Međutim, ako završni graničnik pomerimo na sopstvenu liniju (vertikalno poravnatu sa početnim graničnikom) i samo sadržaj pomerimo udesno, dobijamo neophodne praznine koje ostaju u krajnjoj niski:

```
String json = """
--------------++++++++{
--------------+++++++++"objekat": {
--------------++++++++++"praćenje_grešaka": "uključeno",
--------------++++++++++"prozor": {
--------------+++++++++++"naslov": "Objekat 1",
--------------+++++++++++"ime": "pozadinski_prozor"
--------------++++++++++},
--------------++++++++++"slika": {
--------------+++++++++++"izvor": "slike\\sw.png"
--------------++++++++++},
--------------++++++++++"tekst": {
--------------+++++++++++"podaci": "Klikni ovde",
--------------+++++++++++"veličina": 39
--------------++++++++++}
--------------+++++++++}
--------------++++++++}
              """;
```

Naravno, iste neophodne praznine možemo dodati i pomeranjem završnog graničnika ulevo:

```
String json = """
-------+++++++{
-------+++++++++"objekat": {
-------++++++++++"praćenje_grešaka": "uključeno",
-------++++++++++"prozor": {
-------+++++++++++"naslov": "Objekat 1",
-------+++++++++++"ime": "pozadinski_prozor"
```

```
------++++++++++++},
------++++++++++++"slika": {
------++++++++++++++"izvor": "slike\\sw.png"
------++++++++++++},
------++++++++++++"tekst": {
------++++++++++++++"podaci": "Klikni ovde",
------++++++++++++++"veličina": 39
------++++++++++++}
------++++++++++}
------++++++++}
       """;
```

Štaviše, možemo prilagoditi svaku liniju teksta ručnim dodavanjem praznina, kao u sledećem primeru:

```
String json = """
--------------{
--------------++++"objekat": {
--------------++++++++"praćenje_grešaka": "uključeno",
--------------++++++++"prozor": {
--------------++++++++++++++++++++"naslov": "Objekat 1",
--------------++++++++++++++++++++"ime": "pozadinski_prozor"
--------------++++++++},
--------------++++++++"slika":  {
--------------++++++++++++++++++++"izvor": "slike\\sw.png"
--------------++++++++},
--------------++++++++"tekst":   {
--------------++++++++++++++++++++"podaci": "Klikni ovde",
--------------++++++++++++++++++++"veličina": 39
--------------++++++++}
--------------++++}
--------------}""";
```

Sledeće, razmotrićemo neke pomoćne metode svrsishodne za uvlačenje.

Upotreba metoda za uvlačenje

Počevši od JDK 12 verzije, možemo dodati neophodne praznine konstantnoj niski pomoću metoda `String.indent(int n)`, gde `n` predstavlja broj praznina. Ovaj metod se može primeniti i za uvlačenje celog sadržaja tekstualnog bloka, kao u sledećem primeru:

```
String json = """
--------------********{
--------------*********++"objekat": {
--------------**********++++"praćenje_grešaka": "uključeno",
```

```
--------------*********++++"prozor": {
--------------*********++++++"naslov": "Objekat 1",
--------------*********++++++"ime": "pozadinski_prozor"
--------------*********++++},
--------------*********++++"slika": {
--------------*********++++++"izvor": "slike\\sw.png"
--------------*********++++},
--------------*********++++"tekst": {
--------------*********++++++"podaci": "Klikni ovde",
--------------*********++++++"veličina": 39
--------------*********++++}
--------------*********++}
--------------********}""".indent(8);
```

Praznine dodate pomoću metoda `indent()` nisu vidljive u uređivaču koda integrisanog razvojnog okruženja okruženja, ali su ovde označene znakom „*”, samo da bi se ilustrovao efekat na krajnju nisku. Međutim, kada se koristi `indent()`, dodaje se i nova linija, čak i ako je završni graničnik na kraju sadržaja. U ovom kontekstu, pomeranje završnog graničnika na sopstvenu liniju ima isti efekat, tako da ne očekujte da će biti dodate dve nove linije. Naravno, slobodno vežbajte sa priloženim kodom, da bolje razumete.

Metod `indent()` je koristan za poravnanje bloka sadržaja koji sadrži linije teksta na istom nivou uvlačenja, poput sledeće pesme:

```
String pesma = """
               Hteo bih da uspostavim snagu; nalik korenu,
               usidren u nadi solidnosti.

               Ostavite kontaminaciju nestabilnosti.
               Dokaži da sam pesnik svakog stiha proze.""";
```

Ako ručno dodamo praznine ispred svake linije pesme, kompajler će ih ukloniti, tako da nije moguće globalno dodati neophodne praznine. Možemo pomeriti završni graničnik na sopstvenu liniju i pomeriti ga ulevo, ili pomeriti sadržaj udesno da bismo dobili željene neophodne praznine. Međutim, u takvom slučaju, potrebno je ukloniti novu liniju koja je dodata (kao rezultat pomeranja završnog graničnika na sopstvenu liniju). Najlakši način za to je pomoću nove specijalne sekvence iz JDK 14 verzije, `\`. Dodavanjem ove specijalne sekvence na kraj linije, kompajleru se daje instrukcija da suzbije dodavanje novog reda na tu liniju:

```
String pesma = """
               Hteo bih da uspostavim snagu; nalik korenu,
               usidren u nadi solidnosti.
```

```
            Ostavite kontaminaciju nestabilnosti.
            Dokaži da sam pesnik svakog stiha proze.\
  """;
```

Dok je ova specijalna sekvenca (\) detaljno razmotrena u problemu 5, *Upotreba tekstualnih blokova radi bolje čitljivosti*, pogledajmo nekoliko pristupa zasnovanih na `String` API interfejsu.

Pre JDK 11 verzije, možemo ukloniti ovu liniju pomoću jednostavnog regularnog izraza, poput `replaceFirst("\\s++$", "")`, ili se osloniti na pomoćnika treće strane, poput Apache Commons metoda `StringUtils.stripEnd()`. Međutim, od JDK 11 verzije, ovaj cilj možemo postići pomoću metoda `String.stripTrailing()`, kao u sledećem primeru:

```
String pesma = """
            Hteo bih da uspostavim snagu; nalik korenu,
            usidren u nadi solidnosti.

            Ostavite kontaminaciju nestabilnosti.
            Dokaži da sam pesnik svakog stiha proze.
  """.stripTrailing();
```

Sada je blok sadržaja uvučen, kao rezultat pomeranja završnog graničnika ulevo, a automatski dodata nova linija je uklonjena, zahvaljujući metodu `stripTrailing()`.

Važna napomena

Pored metoda `stripTrailing()`, JDK 11 verzija donosi i metode `stripLeading()` i `strip()`. Takođe, od JDK 15 verzije, imamo i metod `stripIndent()`, koji uklanja početne i prateće praznine na isti način kao što to radi kompajler.

Međutim, od JDK 12 verzije, možemo koristiti metod `String.indent(int n)`, čime izbegavamo ručno dodavanje praznina:

```
String pesma = """
            Hteo bih da uspostavim snagu; nalik korenu,
            usidren u nadi solidnosti.

            Ostavite kontaminaciju nestabilnosti.
            Dokaži da sam pesnik svakog stiha proze."""
  .indent(6)
  .stripTrailing();
```

Sada je vreme da detaljno analiziramo algoritam za uklanjanje nepotrebnih praznina.

4. Uklanjanje nepotrebnih praznina u tekstualnim blokovima

Uklanjanje nepotrebnih praznina u tekstualnim blokovima obično je zadatak koji obavlja kompajler, pomoću specijalnog algoritma. Da bismo razumeli osnovne aspekte ovog algoritma, analiziraćemo sledeći primer:

```
String json = """                          |Kompajler:
----{                                      |Linija 01: 4  lws
----++"objekat": {                         |Linija 02: 6  lws
----++++"praćenje_grešaka": "uključeno",   |Linija 03: 8  lws
----++++"prozor": {                        |Linija 04: 8  lws
----++++++"naslov": "Objekat 1",           |Linija 05: 10 lws
----++++++"ime": "pozadinski_prozor"       |Linija 06: 10 lws
----++++},                                 |Linija 07: 8  lws
----++++"slika": {                         |Linija 08: 8  lws
----++++++"izvor": "slike\\sw.png"         |Linija 09: 10 lws
----++++},                                 |Linija 10: 8  lws
----++++"tekst": {                         |Linija 11: 8  lws
----++++++"podaci": "Klikni ovde",         |Linija 12: 10 lws
----++++++"veličina": 39                   |Linija 13: 10 lws
----++++}                                  |Linija 14: 8  lws
----++}                                    |Linija 15: 6  lws
----}                                      |Linija 16: 4  lws
----""";                                   |Linija 17: 4  lws
```

Posebno nas zanima uklanjanje početnih nepotrebnih praznina predstavljenih u prethodnom isečku koda pomoću znaka „-“.

Za uklanjanje početnih nepotrebnih praznina, kompajler mora da analizira sve linije koje nisu prazne (linije koje sadrže samo praznine se zanemaruju). U našem slučaju, kompajler će analizirati 17 linija. To obuhvata 16 linija JSON koda i liniju sa završnim graničnikom.

Kompajler pregleda svaku od ovih 17 linija i prebrojava broj početnih praznina. Karakter koji predstavlja prazninu nije bitan – to može biti običan razmak, tabulator itd. Svi oni imaju istu vrednost 1, tako da je jedna praznina jednaka jednom tabulatoru. To je neophodno jer kompajler ne može znati kako će tabulatori biti prikazani u različitim uređivačima teksta (na primer, tabulator može zauzimati četiri ili osam karaktera). Kada se ovaj korak algoritma završi, kompajler zna tačan broj početnih praznina za svaku analiziranu liniju. Na primer, prva linija ima 4 **početne praznine** (**lws**), druga linija ima 6, treća ima 8 i tako dalje (proverite prethodni kod za tačne vrednosti).

Važna napomena

Hajde da brzo pogledamo još jednu najbolju praksu za korišćenje tekstualnih blokova: nemojte mešati praznine i tabulatore u istom tekstualnom bloku. Na ovaj način osiguravate doslednost uvlačenja i izbegavate potencijalne nepravilnosti.

U ovom trenutku kompajler računa minimalnu vrednost među brojevima početnih praznina. Rezultat (u ovom slučaju, 4) predstavlja broj nepotrebnih početnih praznina koje treba ukloniti iz svake od 17 linija. Na kraju, barem jedna linija neće imati početne praznine. Naravno, neophodne praznine (dodatna uvlačenja predstavljena znakom „+") ostaju netaknute. Na primer, u petoj liniji imamo 10 početnih praznina – 4 nepotrebne = 6 neophodnih praznina koje ostaju.

U priloženom kodu možete pronaći još tri JSON primera kako biste vežbali ovaj algoritam. Sada ćemo preći na aspekte čitljivosti tekstualnih blokova.

5. Upotreba tekstualnih blokova radi bolje čitljivosti

Upotreba tekstualnih blokova radi bolje čitljivosti znači da želimo da niska izgleda kao tekstualni blok, ali da funkcioniše kao jednolinijska konstantna niska. Ovo je posebno korisno za formatiranje dugih linija teksta. Na primer, možemo želeti da SQL niska izgleda kao tekstualni blok (radi čitljivosti), ali da funkcioniše kao jednolinijska konstantna niska (u smislu da je kompaktna kada je prosledimo u bazu podataka):

```
SELECT "šema"."zaposleni"."ime"
FROM "šema"."zaposleni"
WHERE "šema"."zaposleni"."radno_mesto" = ?
```

Od JDK 14 verzije, ovo možemo postići novom specijalnom sekvencom \ (jedna kosa crta unazad). Dodavanjem ove sekvence na kraj linije, govorimo kompajleru da ne dodaje novi red na toj liniji. Tako u našem slučaju možemo SQL upit predstaviti kao jednolinijsku konstantnu nisku, kao što sledi:

```
String sql = """
             SELECT "šema"."zaposleni"."ime" \
             FROM "šema"."zaposleni" \
             WHERE "šema"."zaposleni"."radno_mesto" = ?\
             """;
```

Obratite pažnju na to da ne dodajete praznine iza \, jer će to izazvati grešku.

Ako ovu sekvencu prosledimo u metod `System.out.println()`, dobićemo sledeću jednolinijsku konstantnu nisku:

```
SELECT "šema"."zaposleni"."ime" FROM "šema"."zaposleni" WHERE
"šema"."zaposleni"."radno_mesto" = ?
```

Sledeće, pogledajmo još jedan primer, kako sledi:

```
String sql = """
  UPDATE "šema"."kancelarija" \
  SET ("adresa_prva", "adresa_druga", "telefon") = \
    (SELECT "šema"."zaposleni"."ime", \
            "šema"."zaposleni"."prezime", ? \
     FROM "šema"."zaposleni" \
     WHERE "šema"."zaposleni"."radno_mesto" = ?\
  """;
```

Ovog puta, dobijena niska nije tačno ono što želimo jer su sačuvane neophodne praznine. To znači da je jednolinijska niska isprekidana nizovima razmaka koje bi trebalo smanjiti na jedan razmak. Ovde može pomoći regularni izraz:

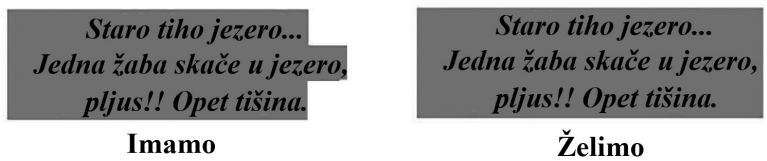
```
sql.trim().replaceAll(" +", " ");
```

Gotovo! Sada imamo jednolinijsku SQL nisku koja izgleda kao tekstualni blok u integrisanom razvojnom okruženju.

Sledeće, pretpostavimo da želimo da ispišemo sledeću pesmu sa lepom pozadinom, upakovanu u tekstualnom bloku:

```
String pesma = """
                  Staro tiho jezero...
               Jedna žaba skače u jezero,
                  pljus!! Opet tišina.
               """;
```

Dodavanje pozadine ovoj pesmi će rezultirati nečim sličnim kao na sledećoj slici:



Slika 1.4: Dodavanje pozadine pesmi

Važna napomena

Obojena pozadina služi samo kao vodič za poravnanje, jer razmak na beloj pozadini ne bi bio vidljiv.

Budući da kompajler uklanja prateće praznine, dobićemo nešto kao na slici levo. Očigledno, želimo nešto kao što je prikazano na slici desno, pa moramo pronaći način da sačuvamo prateće praznine kao neophodne. Od JDK 14 verzije, to možemo učiniti pomoću nove specijalne sekvence, `\s`.

Možemo ponoviti ovu specijalnu sekvencu za svaki razmak na sledeći način (dodajemo tri praznine na prvu liniju i dve praznine na poslednju liniju; na taj način dobijamo simetričan tekstualni blok):

```
String pesma = """
                  Staro tiho jezero...\s\s\s
               Jedna žaba skače u jezero,
                 pljus!! Opet tišina.\s\s
               """;
```

Alternativno, možemo ručno dodati praznine i jednu sekvencu `\s` na kraj linije. To je moguće jer kompajler čuva sve praznine koje se nalaze ispred `\s`:

```
String pesma = """
                  Staro tiho jezero...  \s
               Jedna žaba skače u jezero,
                 pljus!! Opet tišina. \s
               """;
```

Gotovo! Sada su praznine sačuvane, pa kada se primeni boja pozadine, dobićemo nešto kao na desnoj strani *slike 1.4*.

Sledeće, fokusiramo se na izbegavanje karaktera.

6. Izbegavanje navodnika i završetaka linija u tekstualnim blokovima

Izbegavanje sekvence dvostrukih navodnika je potrebno samo kada želimo da u tekstualnom bloku ugradimo niz od tri znaka dvostrukih navodnika (`"""`), na sledeći način:

```
String tekst = """
               Rekla mi je
                      \"""Nemam pojma šta se dešava\"""
               """;
```

Izbegavanje sekvence `"""` može se postići sa `\"""`. Nije potrebno pisati `\"\"\"`.

Rezultujuća niska će izgledati ovako:

```
Rekla mi je
       """Nemam pojma šta se dešava"""
```

Kad god treba da ugradite karaktere `"` ili `""`, jednostavno uradite to na sledeći način:

```
String tekst = """
               Rekla mi je
                       "Nemam pojma šta se dešava"
               """;

String tekst = """
```

```
            Rekla mi je
                   ""Nemam pojma šta se dešava""
            """;
```

Tako da, iako funkcioniše, nemojte ovo raditi jer nije neophodno:

```
String tekst = """
            Rekla mi je
                   \"Nemam pojma šta se dešava\"
            """;

String tekst = """
            Rekla mi je
                   \"\"Nemam pojma šta se dešava\"\"
            """;
```

Međutim, konstrukcija poput """" (gde prvi dvostruki navodnik " predstavlja dvostruke navodnike, a poslednja tri dvostruka navodnika """ predstavljaju završni graničnik tekstualnog bloka) će izazvati grešku. U tom slučaju možete postaviti razmak kao " """ ili izbegavanjem navodnika kao \"""".

Po definiciji, tekstualni blok predstavlja konstantnu nisku koja se prostire na više linija, tako da nema potrebe eksplicitno izbegavati završetke linija (novi redovi), kao što su `\n`, `\r` ili `\f`. Jednostavno dodajte nove linije teksta u tekstualni blok, a kompajler će se pobrinuti za završetke linija. Naravno, to ne znači da njihova upotreba ne funkcioniše. Na primer, tekstualni blok koji ima ispremešane prazne linije može se dobiti pomoću `\n`, na sledeći način:

```
String sql = """
            SELECT "šema"."zaposleni"."ime",\n
                   "šema"."zaposleni"."prezime", ?\n
            FROM "šema"."zaposleni"\n
            WHERE "šema"."zaposleni"."radno_mesto" = ?
            """;
```

Upotreba specijalnih sekvenci (na primer, `\b`, `\t`, `\r`, `\n`, `\f` i slično) u tekstualnim blokovima je ista u starim konstantnim niskama. Na primer, ovde nema ničeg pogrešnog:

```
String tekst = """
              \b\bRekla mi je\n
            \t""Nemam pojma šta se dešava""
            """;
```

Međutim, ista stvar može se postići bez specijalne sekvenci (smatrajte da je `\t` (tabulator) osam praznina):

```
String tekst = """
            Rekla mi je
                   ""Nemam pojma šta se dešava""
            """;
```

Sve ove primere možete vežbati u priloženom kodu.

Važna napomena

Brzo ćemo razmotriti još jednu najbolju praksu za tekstualne blokove: eksplicitno dodavanje specijalnih sekvenci može negativno uticati na čitljivost tekstualnog bloka, pa ih koristite pažljivo i samo kada su stvarno potrebne. Na primer, eksplicitne sekvence `\n` i `\"` su retko potrebne za tekstualne blokove.

Govoreći o `\n` završetku linija (novi red), važno je biti svestan sledeće napomene.

Važna napomena

Verovatno najčešće korišćen završetak linija u jeziku Java je `\n` (Unix, **nova linija** (**LF**)), ali može se koristiti i `\r` (Windows, **povratak na početak linije** (**CR**)) ili `\n\r` (Windows, **povratak na početak nove linije / nova linija** (**CRLF**)). Bez obzira na to koji od ovih preferiramo, Java tekstualni blokovi uvek koriste `\n` (LF). Prvo, kompajler normalizuje sve prekide linija koji nisu eksplicitno dodati putem specijalnih sekvenci u `\n` (LF). Drugo, nakon normalizacije završetka linija i upravljanog uvlačenja, kompajler obrađuje sve eksplicitne specijalne sekvence (`\n` (LF), `\f` (FF), `\r` (CR) i tako dalje), kao u bilo kojoj konstantnoj niski. Praktično, to nam omogućava da u tekstualni blok kopiramo nasleđenu Java nisku koja sadrži specijalne sekvence i da dobijemo očekivani rezultat bez dodatnih izmena.

Ako ikada treba da koristite završetak linija specifičan za vaš operativni sistem, tada morate eksplicitno zameniti završetak linije nakon normalizacije tekstualnog bloka putem metoda `String.replaceAll()`, kao što je `String::replaceAll("\n", System.lineSeparator())`.

Ugrađivanje specijalne sekvence u tekstualni blok može se uraditi na uobičajen način pomoću konstrukcije `\\`. Evo primera ugrađivanja specijalne sekvence `\"` kao `\\\"`:

```
String sql = """
  SELECT \\"šema\\".\\"zaposleni\\".\\"ime\\",
         \\"šema\\".\\"zaposleni\\".\\"prezime\\", ?
  FROM \\"šema\\".\\"zaposleni\\"
  WHERE \\"šema\\".\\"zaposleni\\".\\"radno_mesto\\" = ?
  """;
```

Rezultat možete proveriti u priloženom kodu. Sada, hajde da vidimo kako programski možemo prevesti specijalne sekvence.

7. Programsko prevođenje specijalnih sekvenci karaktera

Već znamo da je kompajler odgovoran za prevođenje specijalnih sekvenci i da, u većini slučajeva, nema potrebe da eksplicitno intervenišemo u ovom procesu. Međutim, postoje slučajevi kada je potrebno programski pristupiti ovom procesu (na primer, da bismo eksplicitno uklonili specijalne sekvence iz niske pre nego što je prosledimo funkciji).

Od JDK 15 verzije, to možemo postići metodom `String.translateEscapes()`, koja je sposobna da ukloni specijalne sekvence poput `\t`, `\n`, `\b` i sl., kao i oktalne brojeve (`\0-\377`). Međutim, ovaj metod ne prevodi Unicode specijalne sekvence (`\uXXXX`).

Možemo izvršiti test jednakosti kako bismo otkrili kako klasa `translateEscapes()` funkcioniše:

```
String novalinija = "\\n".translateEscapes();
System.out.println(("\n".equals(novalinija)) ? "da" : "ne");
```

Kao što možete naslutiti, rezultat je *da*.

Sledeće, pretpostavimo da želimo da koristimo spoljašnju uslugu za štampanje adresa na paketima. Funkcija odgovorna za ovaj zadatak prihvata nisku koja predstavlja adresu bez specijalnih sekvenci. Problem je to što adrese naših klijenata prolaze kroz proces formatiranja koji ih „obogaćuje" specijalnim sekvencama, kao u sledećem primeru:

```
String adresa = """
                JASON MILLER (\\"BIGBOY\\")\\n
                \\tMOUNT INC\\n
                \\t104 SEAL AVE\\n
                \\tMIAMI FL 55334 1200\\n
                \\tUSA
                """;
```

Sledeća slika pokazuje kako će rezultujuća niska izgledati ako ne prevedemo specijalne sekvence iz adrese (leva strana) i kako će izgledati ako to uradimo (desna strana). Naravno, naš cilj je da dobijemo adresu sa desne strane i pošaljemo je na štampanje:

```
JASON MILLER (\"BIGBOY\")\n
\tMOUNT INC\n
\t104 SEAL AVE\n
\tMIAMI FL 55334 1200\n
\tUSA
```

```
JASON MILLER ("BIGBOY")

        MOUNT INC

        104 SEAL AVE

        MIAMI FL 55334 1200

        USA
```

Slika 1.5: Želimo nisku sa desne strane

Prevođenje specijalnih sekvenci može se izvršiti programski, pomoću metoda `String.translateEscapes()`, neposredno pre nego što rezultat pošaljemo spoljašnjoj usluzi. Evo primera koda:

```
String prevedenaAdresa = adresa.translateEscapes();
```

Sada se `prevedenaAdresa` može proslediti spoljašnjoj usluzi za štampanje. Radi vežbe, razmislite o tome kako se koristi ovaj metod za pisanje parsera izvornog koda, ili za Java jezik, ili za drugi programski jezik.

Važna napomena

Slični rezultati (naravno, pročitajte dokumentaciju za detaljne informacije) mogu se postići korišćenjem podrške iz Lang biblioteke treće strane Apache Commons. Razmotrite korišćenje metoda `StringEscapeUtils.unescapeJava(String)`.

Sledeće, govorimo o ugrađivanju izraza u tekstualne blokove.

8. Formatiranje tekstualnih blokova sa promenljivama i izrazima

U jeziku Java je uobičajena praksa da se konstantne niske formatiraju sa promenljivama i izrazima kako bi se dobile dinamičke niske. Na primer, možemo kreirati dinamički deo XML niske pomoću sledeće dobro poznate operacije spajanja:

```
String ime = "Jo";
String prezime = "Kym";

String niska = "<korisnik><ime>" + ime
  + "</ime><prezime>" + prezime + "</prezime></korisnik>";

// izlaz
<korisnik><ime>Jo</ime><prezime>Kym</prezime></korisnik>
```

Naravno, ova jednostavna konstrukcija ima ozbiljne probleme sa čitljivošću. XML kod je čitljiv za ljude samo ako je pravilno formatiran; u suprotnom, veoma je teško pratiti njegovu hijerarhiju. Dakle, možemo li ovaj XML kod predstaviti da izgleda kao na sledećoj slici?

```
<korisnik>
   <ime>Jo</ime>
   <prezime>Kym</prezime>
</korisnik>
```

```
<korisnik>
   <ime>
    Jo
   </ime>
   <prezime>
    Kym
   </prezime>
</korisnik>
```

Slika 1.6: Formatiran XML kod

Naravno da možemo! Specijalnim sekvencama (na primer, `\n`, `\t`, i `\s`), prazninama i sličnim elementima, možemo konstruisati nisku koja izgleda kao na *slici 1.6*. Međutim, bilo bi bolje izraziti ovo spajanje preko tekstualnog bloka. Možda bismo mogli postići istu čitljivost u uređivaču koda integrisanog razvojnog okruženja i u konzoli (tokom izvršavanja programa). Jedan od mogućih pristupa izgleda ovako:

```
String xml = """
             <korisnik>
                <ime>\
             """
         + ime
         + """
              </ime>
                 <prezime>\
              """
          + prezime
          + """
              </prezime>
              </korisnik>
              """;
```

Dakle, možemo spajati tekstualne blokove na isti način kao i konstantne niske pomoću operatora „+”. Sjajno! Izlaz ovog koda je isti kao na levom delu *slike 1.6*. S druge strane, desni deo *slike 1.6* može se dobiti na sledeći način:

```
String xml = """
             <korisnik>
                <ime>
             """
         + ime.indent(4)
         + """
                </ime>
                <prezime>
             """
         + prezime.indent(4)
         + """
                </prezime>
             </korisnik>
             """;
```

Dok rezultujuća niska izgleda dobro u oba slučaja, isto se ne može reći za sam kod. On i dalje ima nizak stepen čitljivosti.

Važna napomena

Pregledom prethodna dva isečka koda lako možemo zaključiti sledeću najbolju praksu za tekstualne blokove: koristite ih samo kada značajno doprinose jasnoći koda i čitljivosti višelinijskih niski. Takođe, izbegavajte deklarisanje tekstualnih blokova u složenim izrazima (na primer, u lambda izrazima), jer to može uticati na čitljivost celog izraza. Bolje je izdvojiti tekstualne blokove kao statičke promenljive i pozivati ih u složenim izrazima.

Pokušajmo drugi pristup. Ovoga puta koristimo metod `StringBuilder` da bismo dobili rezultat kao na levom delu *slike 1.6*:

```
StringBuilder sbXml = new StringBuilder();

sbXml.append("""
            <korisnik>
               <ime>""")
       .append(ime)
       .append("""
               </ime>
                  <prezime>""")
       .append(prezime)
       .append("""
            </prezime>
            </korisnik>""");
```

Rezultat sa desnog dela *slike 1.6* može se dobiti na sledeći način:

```
StringBuilder sbXml = new StringBuilder();

sbXml.append("""
            <korisnik>
               <ime>
            """)
      .append(ime.indent(4))
      .append("""
              </ime>
              <prezime>
           """)
      .append(prezime.indent(4))
      .append("""
              </prezime>
          </korisnik>
          """);
```

Dakle, možemo da koristimo tekstualne blokove u klasama `StringBuilder/StringBuffer` na isti način kao što koristimo konstantne niske. Dok rezultujuća niska odgovara primerima sa *slike 1.6*, sam kod i dalje nije zadovoljavajući, sa stanovišta čitljivosti.

Pokušajmo ponovo, ovog puta metodom `MessageFormat.format()` iz JDK 1.4 verzije. Prvo, oblikujemo primer sa levog dela *slike 1.6*:

```
String xml = MessageFormat.format("""
                          <korisnik>
                              <ime>{0}</ime>
                              <prezime>{1}</prezime>
                          </korisnik>
                          """, ime, prezime);
```

Rezultat sa desnog dela *slike 1.6* može se dobiti na sledeći način:

```
String xml = MessageFormat.format("""
                          <korisnik>
                              <ime>
                               {0}
                              </ime>
                              <prezime>
                               {1}
                              </prezime>
                          </korisnik>
                          """, ime, prezime);
```

Kombinacija tekstualnih blokova i metoda `MessageFormat.format()` je dobitna kombinacija. Čitljivost koda je očigledno bolja. Međutim, idemo dalje da isprobamo metod `String.format()` iz JDK 5 verzije. Kao i obično, prvo obrađujemo levi deo *slike 1.6*:

```
String xml = String.format("""
                          <korisnik>
                              <ime>%s</ime>
                              <prezime>%s</prezime>
                          </korisnik>
                          """, ime, prezime);
```

Dobijanje rezultata sa desnog dela slike 1.6 može se postići na sledeći način:

```
String xml = String.format("""
                          <korisnik>
                              <ime>
                               %s
                              </ime>
                              <prezime>
                               %s
                              </prezime>
                          </korisnik>
                          """, ime, prezime);
```

Kombinacija tekstualnih blokova i metoda `String.format()` je još jedna dobitna kombinacija, ali to nije najnovija funkcionalnost koju možemo koristiti. Od JDK 15 verzije, metod `String.format()` ima praktičniji metod pod nazivom `formatted()`. Evo kako funkcioniše metod `String.formatted()` za reprodukciju levog dela *slike 1.6*:

```
String xml = """
             <korisnik>
                 <ime>%s</ime>
                 <prezime>%s</prezime>
             </korisnik>
             """.formatted(ime, prezime);
```

Rezultat sa desnog dela *slike 1.6* može se dobiti na sledeći način:

```
String xml = """
             <korisnik>
                 <ime>
                  %s
                 </ime>
                 <prezime>
                  %s
                 </prezime>
             </korisnik>
             """.formatted(ime, prezime);
```

To je najbolje što možemo. Uspeli smo da postignemo isti nivo čitljivosti u uređivaču koda integrisanog razvojnog okruženja i tokom izvršavanja programa za tekstualni blok koji sadrži dinamičke delove (promenljive). Sjajno, zar ne?!

Sa stanovišta performansi, možete pronaći uporedne vrednosti ovih pristupa u priloženom kodu. Na sledećoj slici možete videti rezultate ovog testiranja na računaru sa Intel® Core™ i7-3612QM CPU @ 2.10GHz i operativnim sistemom Windows 10, ali slobodno ih testirajte na različitim sistemima, jer rezultati u velikoj meri zavise od konfiguracije.

Testirani metodi	Način	Broj	Ocena		Greška	Jedinice
Main.concatenation	avgt	5	0.670	±	0.004	ns/op
Main.messageFormat	avgt	5	3199.252	±	116.859	ns/op
Main.stringBuilder	avgt	5	318.641	±	9.198	ns/op
Main.stringFormat	avgt	5	1416.266	±	18.024	ns/op
Main.stringFormatted	avgt	5	1409.030	±	42.547	ns/op

Slika 1.7: Rezultati testiranja performansi

U skladu sa ovim rezultatima, spajanje putem operatora „+" je najbrži, dok je `MessageFormat.format()` najsporiji pristup.

9. Dodavanje komentara u tekstualne blokove

Pitanje: Da li možemo dodavati komentare u tekstualne blokove?

Zvaničan odgovor (prema specifikaciji jezika Java): leksička gramatika implicira da komentari ne mogu postojati unutar literala karaktera, konstantnih niski ili tekstualnih blokova.

Možda ćete biti u iskušenju da pokušate nešto ovako, misleći da je to lak trik, ali ja to ne preporučujem:

```
String tekst = """
               foo  /* komentar */
               buzz //drugi komentar
               """.replace("neki_regularni_izraz","");
```

Kratak odgovor: Ne, komentari ne mogu biti unutar tekstualnih blokova.

Pređimo sada na temu mešanja običnih konstantnih niski sa tekstualnim blokovima.

10. Mešanje običnih konstantnih niski sa tekstualnim blokovima

Pre nego što krenemo sa mešanjem običnih konstantnih niski i tekstualnih blokova, razmotrimo sledeće: koliko se obična konstantna niska razlikuje od tekstualnog bloka? Na ovo pitanje možemo odgovoriti sledećim isečkom koda:

```
String niska = "Volim jezik Java!";
String tekst = """
               Volim jezik Java!""";

System.out.println(niska == tekst);      // true
System.out.println(niska.equals(tekst)); // true
```

Vau! Dakle, naš isečak koda dva puta ispisuje `true`. To znači da su obična konstantna niska i tekstualni blok slični u toku izvršavanja. Tekstualne blokove možemo definisati kao konstantne niske koje se prostiru kroz više linija teksta i koriste trostruke navodnike kao svoje početne i završne graničnike. Kako je to moguće? Prvo, instanca proizvedena iz obične konstantne niske i tekstualnog bloka je tipa `java.lang.String`. Drugo, moramo se osvrnuti na unutrašnji rad kompajlera. U osnovi, kompajler dodaje niske u poseban keširani prostor pod nazivom **Prostor konstantnih niski** (**SCP**) (više detalja dostupno je u knjizi *Java Coding Problems*, prvo izdanje, problem 48, *Immutable string*) radi optimizacije memorije, a od JDK 13 verzije, tekstualni blokovi se nalaze u istom prostoru kao i niske.

Sada kada znamo da nema većih razlika u načinu na koji se obične konstantne niske i tekstualni blokovi tretiraju interno, možemo ih s poverenjem mešati u jednostavnoj operaciji spajanja (u suštini, tekstualni blok se može koristiti svuda gde se može koristiti obična konstantna niska):

```
String tom = "Tom";
String dzeri = """
               Džeri""";

System.out.println(tom + " i " + dzeri); // Tom i Džeri
```

Štaviše, pošto tekstualni blok vraća nisku, možemo koristiti čitav arsenal metoda koje koristimo za obične konstantne niske. Evo primera:

```
System.out.println(tom.toUpperCase() + " I "
  + dzeri.toUpperCase()); // TOM I DŽERI
```

Takođe, kao što ste već videli u *problemu 8*, *Formatiranje tekstualnih blokova sa promenljivama i izrazima*, tekstualni blokovi se mogu koristiti i mešati sa običnim konstantnim niskama u klasama `StringBuilder/StringBuffer`, i metodima `MessageFormat.format()`, `String.format()` i `String.formatted()`.

11. Mešanje regularnih izraza sa tekstualnim blokovima

Regularni izrazi mogu se koristiti sa tekstualnim blokovima. Razmotrimo jednostavnu nisku, kao što je sledeća:

```
String imeIAdresa
  = "Mark Janson;243 West Main St;Louisville;40202;USA";
```

Ovde imamo ime (Mark Janson) i neke detalje o njegovoj adresi, razdvojene tačkom i zarezom (;). Često je potrebno provući takve niske kroz regularne izraze i izdvojiti informacije kao imenovane grupe. U ovom primeru možemo razmotriti pet imenovanih grupa:

- `ime`: treba da sadrži ime osobe (Mark Janson)
- `adresa`: treba da sadrži informacije o adresi (243, West Main St)
- `grad`: treba da sadrži ime grada (Louisville)
- `pb`: treba da sadrži poštanski broj grada (40202)
- `drzava`: treba da sadrži naziv države (USA)

Regularan izraz koji odgovara ovim imenovanim grupama može da izgleda ovako:

```
(?<ime>[ a-zA-Z]+);(?<adresa>[ 0-9a-zA-Z]+);(?<grad>[ a-zA-Z]+);(?<pb>[\\
d]+);(?<drzava>[ a-zA-Z]+)$
```

Ovo je jednolinijska niska, pa je možemo koristiti putem Pattern API interfejsa, kao što sledi:

```
Pattern sablon = Pattern.compile("(?<ime>[ a-zA-Z]+);(?<adresa>[ 0-9a-zA-
Z]+);(?<grad>[ a-zA-Z]+);(?<pb>[\\d]+);(?<drzava>[ a-zA-Z]+)$");
```

Međutim, kao što vidite, pisanje našeg regularnog izraza na ovaj način ozbiljno utiče na čitljivost. Srećom, možemo koristiti tekstualne blokove kako bismo rešili ovaj problem, kao što sledi:

```
Pattern sablon = Pattern.compile("""
         (?<ime>[ a-zA-Z]+);\
```