Универзитет у Београду
Математички факултет

Објектно оријентисано програмирање

Програмски језик Јава - 1

Александар Картељ, Владимир Филиповић, Душан Тошић

Београд, 2022.

Предговор

Ова књига је, првенствено, намењена студентима Математичког факултета у Београду и требало би да послужи као уџбеник за предмет Објекно оријентисано програмирање. Међутим, књига може бити од користи свакоме ко жели да научи програмски језик Јаву и да се упозна са принципима програмирања. У књизи је покривена верзија Јава 17LTS.

На српском говорном подручју постоји велики број књига које се односе на програмски језик Јаву и пратеће технологије. Ова књига је специфична због повезаности са предметом Објектно оријентисано програмирање и усклађена је са програмом овог предмета. Књига представља свеобухватни водич за програмски језик Јава, али не и за све његове пратеће библиотеке. Опис коришћења свих кључних библотека језика Јава превазилази обим ове књиге. Пожељно је да читалац познаје програмски језик C, али није неопходно.

За опис конструкција Јаве коришћена је Бекусова нотација. Скоро сваки формални опис синтаксе Јава-конструкција пропраћен је примерима. Поред овога, већина поглавља садржи већи број комплетно урађених задатака. Кроз решења задатака демонстрирају се најважније могућности језика Јава и принципи програмирања. За највећи број решења наведени су тест примери да би се видело како је организован улаз и у каквом облику се добијају излазни подаци. Решени примери из уџбеника доступни су и на GitHub страни посвећеној уџбенику https://github.com/matf-oop-java/tom-1. Упутство за преузимање примера и њихову интеграцију у оквиру неких развојних окружења дато је у додатку Б. На крају сваког поглавља су наведена питања и задаци за вежбање. Ако неко учи било који програмски језик, па самим тим и програмски језик Јаву, пожељно је да самостално уради што већи број задатака за вежбање.

Аутори су настојали да свуда, где је то могуће, користе ћирилично писмо. Коментари у програмима и излазни резултати који садрже текст, исписани су ћирилицом.

Књига се састоји од 15 поглавља, два додатка и литературе.

С обзиром на то да је ово прво издање књиге, грешке су могуће. Штампарске грешке у рукопису су скоро неизбежне, иако су текст пажљиво прочитали аутори и већи број колега. Међу њима велику захвалност дугујемо колегиници др Милани Грбић са Универзитета у Бањој Луци, која је, поред пажљивог читања, помогла и у конципирању питања и задатака на крајевима поглавља.

Унапред се захваљујемо свима на добронамерним сугестијама и предлозима за отклањање уочених грешака, као и побољшања целокупног садржаја књиге.

Посебну захвалност дугујемо рецензентима ..... Они су пажљиво прочитали цео рукопис и корисним сугестијама допринели да ова књига буде квалитетнија.

Аутори

Решавање проблема помоћу рачунара

Скуп програма рачунара чини софтвер (или програмску подршку) тог рачунара. Креирање софтвера је сложен процес и данас постоји посебна инжењерска дисциплина (софтверско инжењерство) која се бави овом проблематиком. Савремени софтверски производи треба да испуњавају низ захтева, као што су: ефикасност, поузданост, корисност, изменљивост, преносивост итд. Софтверско инжењерство има за циљ економичан развој високо-квалитетног софтвера.

1.1. Опис поступка решавања проблема помоћу рачунара

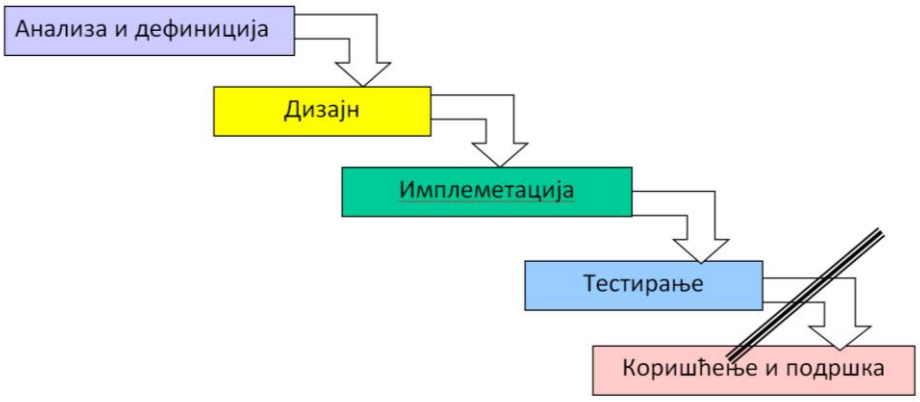
Постоји више модела преко којих се може описати процес креирања софтвера, тј. поступак решавања проблема помоћу рачунара. Овде ће бити обрађена два модела: водопадни и спирални.

1.1.1. Водопадни модел

Један од најстаријих и најчешће коришћених модела је тзв. водопадни модел у којем се издвајају четири основне фазе у креирању софтвера:

1. анализа и дефиниција проблема,
2. дизајн,
3. имплементација (кодирање) и
4. тестирање.

Поред ове четири фазе, често се у процес креирања софтвера убраја и фаза коришћења и подршке. Резултат сваке од фаза у креирању софтвера су одређени артефакти, који представљају улазне елементе за наредну фазу: резултат фазе анализе су документи о архитектури софтвера, резултат фазе дизајна су документи о детаљном дизајну система, резултат фазе имплементације је сам софтвер, а резултат фазе тестирања су извештаји о исходу тестирања.
Ове фазе могу се представити у облику водопада (као на следећем дијаграму) па отуд назив водопадни модел.



Редослед фаза у креирању софтвера по водопадном моделу

Слично као што вода у водопаду увек тече наниже, тако и ток активности креирања софтвера у водопадном моделу увек иде у једном смеру. Према водопадном моделу, пре почетка нове фазе рада морају бити успешно окончане све активности претходне фазе – фазе се не преклапају.
Дакле, тек по завршетку комплетне анализе прелази се на дизајн, по завршетку дизајна комплетног система на његову имлементацију, а тестирање почиње тек кад систем буде комплетно завршен. Водопадни модел може се представити и пирамидом.

Водопадни модел

Фаза анализе и дефиниције проблема

У фази анализе и дефиниције анализира се проблем и постављају захтеви које мора задовољавати новокреирани софтвер. Ово је значајна фаза јер од постављених захтева зависи како ће изгледати крајњи производ. Уколико захтеви нису прецизно наведени, може се десити да коначан производ не испуњава очекивања будућих корисника. У фази анализе и дефиниција обично се разликују две подфазе:

1. планирања и
2. дефинисања.

Током планирања прави се студија изводљивости, дефинише се речник (програмер често није упознат са терминологијом области за коју се софтвер креира па је потребно објаснити стручне термине), процена цене, прелиминарна спецификација и израда пројектног плана. У току дефинисања врши се детаљна спецификација захтева, дефинише се кориснички интерфејс, као и упутство за кориснике. Резултат фазе анализе и дефинисања треба да буде неколико докумената, као што су: структура пројекта, основа уговора итд. У примерима, које ћемо користити у овој књизи, захтеви ће бити дефинисани постављањем задатка. Приликом објашњавања решења врши се анализа. Када се учи програмирање, користе се једноставнији програми за демонстрирање својстава програмског језика па је ова фаза знатно једноставнија, него ли приликом професионалне израде софтвера.

Фаза пројектовања (дизајна) софтвера

У овој фази врши се пројектовање производа, тј. будући софтверски производ се моделира на одређен начин — коришћењем погодног алата. Као средство за моделирање све више се користи UML (енг. Unified Modeling Language), али и разни псеудојезици, као и друга средства. Ова фаза се, често, своди на глобални опис алгоритма постављеног задатка. У оквиру ове фазе треба изабрати метод пројектовања, водити рачуна о архитектури софтвера (да ли треба да буде: слојевита,

клијент-сервер, серверски оријентисана архитектура итд.) Дизајн софтверског производа мора бити урађен у складу са постављеним захтевима у фази анализе.

Фаза импементације

Фаза имплементације састоји се од кодирања (писања програма) коришћењем неког конкретног програмског језика, а на основу модела направљеног у претходној фази. Фаза имплементације сматра се рутинском за искусног програмера. Наравно, овде се претпоставља добро познавање програмског језика у којем се врши кодирање, као и програмских библиотека и развојних окружења за тај програмски језик. Међутим, може се десити да се имплементација реализује коришћењем више програмских језика и онда ова фаза постаје компликованија, јер модуле креиране у различитим програмским језицима треба повезати у једну целину.

У овој књизи се изучавају методе програмирања презентујући својства програмског језика Јава. Да би се стекло одређено искуство, морају се добро упознати могућности програмског језика. То се постиже у фази имплементације и стога је она у фази учења програмирања веома значајна. Преко навођења једноставних примера програма добро се могу описати карактеристике програмског језика.

Фаза тестирања софтвера

У фази тестирања врши се испитивање да ли софтвер испуњава раније постављене захтеве. Тестирање се реализује тако што се изаберу подаци за које се унапред зна коректан одговор, па се за те податке проверави да ли софтвер који се тестира даје коректан резултат. Понекад је могуће закључити да ће програм радити коректно ако даје исправне резултате за неке кључне податке. Од искуства особе која врши тестирања (тестер софтвера) зависи избор ових података. Фаза тестирања служи да се открију евентуалне грешке у програму. Уколико постоје грешке, уследила би корекција. Тестирање софтвера може бити врло компликовано, посебно када се не могу наћи подаци за које је унапред познат резултат рада програма. Стога постоје разне методологије тестирања и различити алати који се могу користити при тестирању софтвера. Овде се нећемо њима бавити.

У задацима који следе, фаза тестирања биће демонстрирана навођењем неких тест-примера. Број тест-примера је ограничен због недостатка простора, а код једноставних задатака тест-примери ће бити изостављени. Међутим, код сложенијих задатака препоручује се читаоцу детаљније испитивање програма за већи број карактеристичних случајева.

Фаза коришћења и подршке

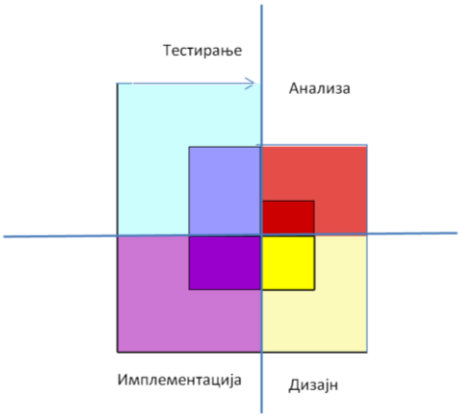
Фаза коришћења и подршке позната је још и под именом фаза експлоатације. Уколико је софтверски производ успешно прошао фазу тестирања, иде у фазу коришћења. Наиме, софтвер се испоручује корисницима са потребном документацијом. У току коришћења софтвера понекад је потребно извршити одређене измене и прилагођавања конкретним потребама, тј. потребно је одржавати софтвер. Стога се ова фаза назива фаза коришћења и одржавања (или експлоатације и подршке).

Фаза експлоатације и одржавања овде практично неће бити заступљена јер су наведени крађи програми преко којих се учи програмирање, а ти програми, најчешће, немају неку значајну примену у пракси.

1.1.2. Спирални модел креирања софтвера

Ако се приликом тестирања софтвера закључи да софтвер не испуњава постављене захтеве, потребно је преиспитати претходне фазе, тј. практично поново проћи кроз све претходне фазе и поново извршити тестирање. Код модела водопада, то значи да се претходне фазе преиспитују за цео систем, што је превише ригидно у многим ситуацијама. Честе су ситуације у којима би се боље показао модел где се софтвер развија еволутивно и где се у постепено додају нове функциоиналности у већ развијен софтвер.

Такав, веома популарни модел за развој софтвера је спирални модел.
Спирални модел садржи исте фазе као водопадни модел, али је ток активности другачији. Код овог модела, уместо тока наниже (са евентуалним повратком на почетак у случају да је уочен дефект) који је одлика водопадног модела, активности имају ток у облику спирале.
Спирални модел се може представити следећим дијаграмом:
Један пролаз кроз све четири фазе у спиралном моделу назива се итерација.



Спирални модел

једна итреација се односи на реализацију групе функционалности у систему, а не система као целине. Јасно је да у току креирања сложеног софтвера постоји више итерација. После тестирања у једној итерацији, уколико се тестирањем потврди да су испуњени захтеви из фазе анализе и дефиниција, иде се у нову итерацију. Ако ти захтеви нису испуњени, врши се повратак на почетак текуће итерације. Након успеха при тестирању у последњој итерацији (када се закључи да су испуњени затеви из фазе анализе), иде се у фазу коришћења. Спирални модел је тежи за вођење, троши се више времена приликом његове примене, али многи сматрају да он више одговара стварном стању ствари приликом развоја софтвера. Из изложеног је јасно да спирални модел представља модификацију водопадног модела.

Са друге стране, водопадни модел се може посматрати као спирални модел са једном итерацијом.
Спирални модел можемо изразити помоћу пирамиде као и водопадни модел.
На следећем дијаграму, који предстаља спирални модел, делови пирамиде издвојени испрекиданим или ивичним линијама представљају једну итерацију спиралног модела.

Спирални модел представљен преко пирамиде

Према неким статистичким подацима, дистрибуција утрошеног времена по фазама (за четири основне фазе) је следећа:

анализа и дефиниција……..20%,
дизајн……15%,
имлементација (кодирање) …….20%,
тестирање……45%.

Фазу експлоатације се овде не разматра јер је она другачије природе и може да траје годинама.
Из наведеног се види да фаза тестирања најдуже траје. Овде треба имати у виду да је реч о сложеним софтверским производима који могу имати неколико хиљада (чак и неколико стотина хиљада) линија кода. Такви програми имају огроман број могућности. Да би се истестирале све те могућности, потребно је много времена (код сложених програма дешавало се да програм успешно прође фазу тестирања, користи се неколико година и након тога се открије да садржи грешку).
Процес учења програмирања разликује се од процеса развоја софтвера. Већ је речено да све ове фазе постоје и у процесу учења прављења програма, али неке су знатно поједностављене. Тако се фаза анализе и дефиниција обично своди на формулацију задатка. У опису решења заступљена је и фаза дизајна, где се помоћну говорног језика, често уз коришћење математичке симболике, описује модел.

1.2. Језички процесори

Под језичким процесором се подразумева програм за обраду језика. Могу се вршити разне врсте обраде (процесирања) над разним врстама језика. На пример, може се вршити превођење са једног говорног језика на други или са једног програмског језика на други. Како савремени рачунари могу да извршавају само програме изражене помоћу бинарне азбуке, тј. помоћу симбола 0 и 1, то сваки другачије написан програм, мора бити преведен на језик изражен симболима 0 и 1, тј. на машински језик. Постоје разне врсте програмских језика – неки су блиски машинском језику (тзв. машински оријентисани језици), док су други ближи говорним језицима човека и њих називамо вишим програмским језицима. Језички процесори, који врше превођење са машински оријентисаних језика на машински, у приципу су једноставнији од процесора који врше превођење са виших пограмских језика на машински.
Језички процесори за превођење са машински оријентисаних језика називају се асемблери, а сам процес превођења, назива се асемблирање. Процес асемблирања можемо да прикажемо следећом сликом.

Процес асемблирања

Изворни програм је написан на машински оријентисаном језику и након асемблирања, преводи се у језик над бинарном азбуком, тзв. објектни програм. Објектни програм је на машинском језику, али није спреман за извршавање. Понекад је потребно повезати више независно преведених објектних програма у једну целину, што ради повезивач

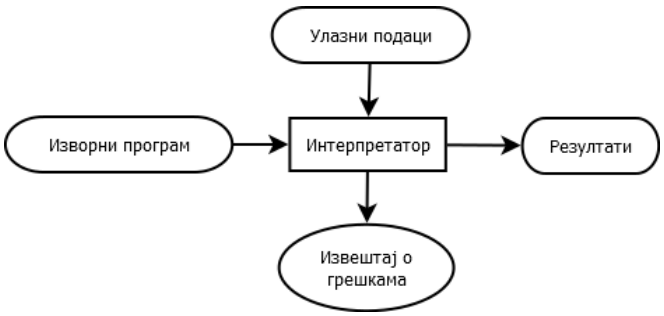
(није приказано на дијаграму). Стога се објектни програм обрађује посебним програмом (тзв. пунилац) који га смешта на одређено место у меморији и омогућава да буде извршни. Постоје два начина за превођење изворних програма са виших програмских језика на машински језик. То су:

1. компилација и
2. интерпретација.

Ако су процес превођења и процес извршавања програма временски раздвојени, врши се компилација (компајлирање) програма. Компилацију врше процесори који се називају компилатори или компајлери. Процес компилације можемо да прикажемо следећом сликом.

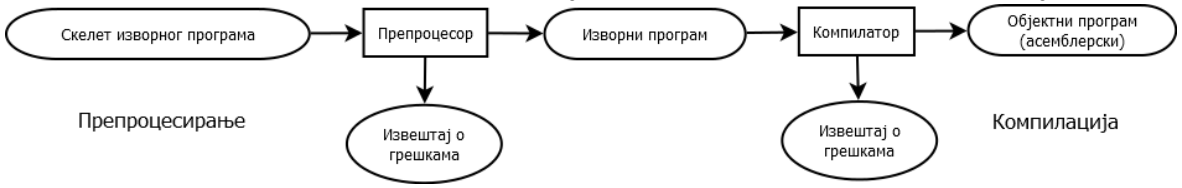
Процес компилације

Ако су процес превођења и процес извршавања програма временски повезани (преведе се део наредби па се изврши, затим се преведе друга део наредби па се изврши итд.), врши се интерпретација програма. Процесори који врше интерпретацију програма називају се интерпретатори или интерпретери. Процес интерпретације можемо да прикажемо следећом сликом.



Процес интерпретације

Понекад се у програму могу наћи конструкције које не припадају програмском језику, али се могу додатном обрадом свести на конструкције тог језика. Та додатна обрада се примењује приликом компилације и назива се препроцесирање, а сам обрађивач, назива се препроцесор. Препроцесирање је карактеристично за програмски језик C.



Процес компилације са препроцесирањем

За један програмски језик могу се креирати и компилатор и интерпретатор. Међутим, најчешће се за конкретан програмски језик већ у самом његовом дизајну одлучује да буде или погодан за интерпретацију или за компилацију. Постоје и језици код којих се комбинују компилација и интерпретација и такав је програмски језик Јава.

1.3. Резиме

Развој софтвера је изазован и комплексан процес за који су потребне различите вештине. Разнородност потребних вештина је допринела диверсификацији знања и улога учесника у развоју софтвера. Тако поред програмера који развијају нови кôд, постоје и програмери специјализовани за проверу квалитета софтвера (тестери), искусни програмери који су у стању да размишљају на високом нивоу (систем архитекте) итд. Одржавање и функционисање овако комплексног система људи и производа (софтвера) захтева добру организацију људских ресурса, дисциплину у дефинисању захтева, писању кода, документовању кода итд. Развој софтвера такође захтева постојање разноврсних софтверских алата. Један од најбитнијих је језички процесор за виши програмски језик. Он омогућава брзо и прецизно формулисање идеја и алгоритама. и њихово даље извршавање на конкретном рачунару.

1.4. Питања и задаци

1. Упоредити водопадни и спирални модел креирања софтвера. По чему су слични, по чему се разликују?
2. Упоредити процес компилације програма и интерпретације програма. Истражити који програмски језици врше компилацију, а који интерпретацију.
3. Шта се подразумева под процесом препроцесирања?
4. Описати процес асемблирања.

2. Објектно оријентисано програмирање

Објектно оријентисано програмирање је програмска парадигма заснована на скупу објеката који остварују међусобну интеракцију. Главне интеракције везане су за манпулисање објектима и размену информација између њих. Решавање проблема се релизује преко објеката који дејствују међусобно. Како је једна од најважнијих особина Јаве објектна оријентисаност, посветићемо више пажње објектно оријентисаној парадигми.

2.1. Каракеристике објектно оријентисаног програмирања

Теоријска основа објектно оријентисане парадигме је филозофски правац према којем је свет састављен из објеката помоћу којих се моделирају одређени феномени.
Објектно оријентисана парадигма обезебеђује механизме за напредну репрезентацију елемената проблема који се решава – такву да репрезентација буде довољно општа и да не буде ограничена конкретним типом проблема. Репрезентације елемената проблема се називају објекти и они омогућују да се проблем који се решава описује помоћу појмова истог проблемског домена, а не помоћу „чисто“ рачунарских појмова.
Објектно оријентисана парадигма има следеће карактеристике:

1. Основни градивни елементи, објекти, се могу посматрати као ентитет који садржи податке, који може проследити захтеве другим објектима и који и сам може реализовати операције.
2. Програм се може схватити као скупина објеката који (преко порука) шаљу захтеве једни другима. Порука је захтев да се позове метод који припада датом објекту.
3. Сваки објекат поседује сопствену меморију у којој се могу наћи и други објекти.
4. Сваки објекат има свој тип. Сваки од објеката је примерак неког типа.
5. Сви објекти датог типа могу прихватати и процесирати исте поруке.

Још сажетије исказано, објекти имају своје стање (обезбеђено преко унутрашњих података датог објекта), своје понашање (обезбеђено преко метода датог објекта) и свој идентитет (дефинисан јединственом адресом датог обекта у меморији).

2.2. Историјат и развој објектно оријентисаног програмирања

Први објектно оријентисани језик био је Simula 67, настао 1967. године, али није стекао велику популарност. Главни промотери објектно оријентисане парадигме били су творци програмског језика SmallTalk (група програмера под руководством Алена Кеја) – они су агресивним рекламирањем свог производа битно утицали на развој и на популарност објектно оријентисане парадигме.
Године 1983. појављује се језик C++ који је представљао објектно оријентисану надградњу тада популарног процедуралног језика C. Језик C++ убрзо постаје један од најпопуларнијих програмских језика, тако да и он доприноси популаризацији објектно оријентисаног стила програмирања.
Појављују се и нови, моћни програмски језици (Java, C#, ...), који подржавају објектно оријентисану парадигму, а конструкције за подршку ове прадигме се укључују и у већ постојеће популарне програмске језике (Fortran, Python, ...). Период настанка ове

парадигме карктерише постојање огромне продукције софтвера, таквог да се једном написан софтвер тешко могао поново користити у некој другој ситуацији. Објектно оријентисан приступ омогућава (преко објеката, класа и наслеђивања) лако поновно коришћење већ написаног (eng. reusability) софтвера, али и читав низ нових концепата.

2.3. Основни појмови објектно оријентисаног програмирања

Као што је већ истакнуто, подаци и функције (процедуре) за рад са њима се учаурују у објекте. На тај начин, подаци се скривају да би се заштитила унутрашња својства објеката. Они подаци којиме треба да да се приступи из спољашњости су изложени као атрибути датог објекта. Објекти интерагују међусобно разменом порука. Овде се запажа читав низ нових појмова, као што су: објекат, атрибут, порука итд.

2.3.1. Објекат, атрибут, метод

Објекат је интегрална целина података и функција (процедура) за рад са њима. Функције (процедуре) се, под утицајем SmallTalk-а, називају **методи**. У објекту су **учаурени** (енкапсулирани) подаци које тај објекат садржи и методи за рад са њима. Због присуства метода у објектима, објекти имају могућност да самостално делују, тј. постају динамички. Објекти се користе и у другим парадигмама, али су статички и са њима се оперише спољашњим средствима. Подаци унутар објекта представљају **атрибуте** (особине) објекта. Атрибути се још називају и **пољима**. Вредности атрибута чине (унутрашње) **стање** објекта.
Пожељно је да се промена унутрашњег стања објекта реализује само преко метода смештених унутар тог објекта.

Метод је функција (процедура) која је саставни део објекта, тј. поступак којим се реализује порука упућена објекту. Методи одређују понашање објекта.
Порука је скуп информација који се шаље објекту. Састоји се из адресе (објекта примаоца поруке) и саопштења (које казује шта треба да се уради). Поруком се преноси информација са очекивањем да ће уследити активност на објекту који прима поруку. Могу се разликовати три врсте порука: информативне, упитне и командне.

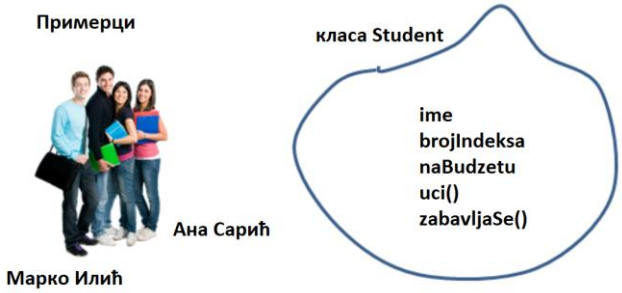
Илустрација објекта, атрибута и метода

На овој слици, објекат је човек. Као што се може видети, атрибути су `ime` и `starost`, а методи су: `razmislja()`, `kreceSe()` и `radi()`. Поред метода и атрибута, објекат нужно има и идентитет, тј. вредност која га идентификује – што у пракси обично бива информација изведена из адресе меморијске локације у којој се налази тај објекат.
Препоручује се да се, приликом објектно оријентисане анализе и дизајна, објекти посматрају као ентитети који обезбеђују услуге. Дакле, да би програм који се развија могао да пружи услугу крајњем кориснику, потребно је да се размотри како ће тај програм да користи услуге које му пружају други објекти. Објекте који пружају потребне

услуге програмер може или сам да дизајнира или да искористи већ готове објекте у расположивим библиотекама (што је у принципу боље решење, када је оно могуће).
Посматрање објеката као пружалаца услуга представља добар начин за упрошћавање, што помаже у анализи, у лакшем разумевању постојећег кода и у поновном коришћењу објеката. Ако се вредност објеката разматра на основу услуга које он пружа, онда се тако разматрани објекти лакше могу укључити у програм који се развија.
Посматрањем објеката као пружаоца услуга, постиже се, осим обезбеђења јасне поделе одговорности међу објектима, висок ниво кохезије. Висок ниво кохезије значи да се различити аспекти објекта међусобно добро уклапају тј. да чине складну целину. Код квалитетног објектно оријентисаног дизајна, сваки објекат једну ствар ради добро, али не покушава да уради превише. На тај начин, побољшава се поновна искористивост (енг. reusability) – не само што је олакшано тражење већ развијених објеката са потребним карактеристикама, већ је омогућено и креирање нових објеката који касније лако могу бити поново коришћени.

2.3.2. Класе и инстанце

Иако код неких популарних програмских језика (на пример, JavaScript) није тако, код највећег броја објектно оријентисаних програмских језика су објекти реализовани помоћу класе. Класе описују структуру и понашање (функционалност) објеката.
Увођењем класа се на елегантан начин омогућује да бројни објекти имају заједничке карактеристике. На пример сваки студент учи, сваки студент полаже испите, сваки факултет организује испите итд. Са друге стране, сваки од студената има своје име и презиме, као и сопствени број индекса, а сваки од факултета има своју адресу, свој акредитовани план и програм студија итд. Стога сваки од ентитета који представља студенте, факултете, професоре и слично може бити представљен једним елементом рачунарског програма. Сваки од тих ентитета је објекат, при чему сваки од објеката припада конкретној класи.
Другим речима, класа представља шаблон за креирање датих објеката – примерака (инстанци) те класе. Класе су организоване хијерархијски или мрежно и повезане међусобно механизмом наслеђивања.
Класа дефинише шаблон за креирање конкретних објеката (инстанци или примерака) и описује структуру и функционалности објекта. Појам класе је веома битан у програмском језику Јава — сви методи, па и главни метод којим се покреће програм (`main()`) су увек смештени у некој класи.
Инстанца (примерак) класе је конкретан објекат дате класе. Објекат је потпуно одређен својим атрибутима и понашањем. У даљем тексту ће се термини примерак, инстанца или објекат класе сматрати синонимима.
Пример 1. Следећа слика илуструје концепте објекта и класе, као и однос међу њима . □



Илустрација објекта и класе

Јава програм, који приказује податке о студентима са претходне слике, би обухватао две датотеке. Прва датотека (названа `Student.java`) би садржавала Јава кôд који описује својства студента.

```
class Student {
    String ime;
    int brojIndeksa;
    boolean naBudzetu;
    void stampajPodatke() {
        System.out.println("Име студента је: " + ime
             + ", број индекса је: " + brojIndeksa
             + " На буџету: " + naBudzetu + ".");
    }
}
```

Конкретни објекти, који представљају студенте, креирају се помоћу оператора `new` и њихови атрибути се подешавају у другој Јава датотеци `StudentPokretanje`. Ако се атрибути не промене експлицитно, они ће бити постављени на неке подразумеване вредности карактеристичне за сваки тип података. У овој класи се такође налази и почетни метод рада програма (`main()`). Овај метод може да буде убачен и у саму класу `Student`.

```
class StudentPokretanje {
    public static void main(String[] args) {
        Student prvi = new Student();
        prvi.ime = "Марко Илић";
        prvi.brojIndeksa = 243;
        prvi.naBudzetu = false;

        Student drugi;
        drugi = new Student();
        drugi.ime = "Ана Сарић";
        drugi.brojIndeksa = 25;
        drugi.naBudzetu = true;

        prvi.stampajPodatke();
        drugi.stampajPodatke();
    }
}
```

Класом је дефинисан тип објекта, а за сваки објекат примерак, тј. инстанцу, инстанцна променљива има конкретну вредност атрибута. Испис који се добија приликом извршавања овог програма је дат испод.

```
Име студента је: Марко Илић, број индекса је: 243 На буџету: false.
Име студента је: Ана Сарић, број индекса је: 25 На буџету: true.■
```

2.3.3. Наслеђивање и композиција

Потклаcа класе A је класа B уколико све инстанце класе B имају исте атрибуте и методе као инстанце класе A. Поткласе обично настају додавањем нових атрибута и метода постојећој класи.
Наткласа класе B је класа A уколико је B потклаcа класе A.

Наслеђивање је механизам за крерање нових класа из постојећих. Наслеђивањем се формирају релације између једне класе и више других. Наслеђивање описује однос који може бити именован на два начина:

- «јесте конкретизација» када се посматра са позиције потклсасе (на пример, «Студент је конкретизација класе Човек»).
- «јесте уопштење» када се посматра са позиције наткласе (на пример, «Човек је уопштење класе Студент»).

Наслеђивање омогућава да се методи из неке класе могу користити у њеним потклсасама. На тај начин је омогућено да једном написан софтвер не мора да се пише поново, већ само да се наследи из наткласе.

Пример 2. Треба дефинишисати класу `Poslediplomac` која је потклсаса класе `Student` и која додатно садржи: логичко поље `zaposlen`, целобројно `brojIspita` и још два метода, један за постављање броја испита, а други за враћање његове вредности. Потклсаса се генерише помоћу кључне речи `extends`.□

```
class Poslediplomac extends Student {
	boolean zaposlen;
	private int brojIspita;

	public void postavibrojIspita(int b) { brojIspita = b; }

	public int uzmibrojIspita() { return brojIspita; }
}
```

Инстанца класе `Poslediplomac` има сва својства класе `Student`, као и нека додатна. Наслеђене су променљиве: `ime`, `brojIndeksa`, `naBudzetu` и није потребно да се поново декларишу у класи `Poslediplomac`. То исто важи и за метод `stampajPodatke()`.

На основу програмског кода претходно развијене класе `StudentPokretanje` развијена је класа `StudentPosllediplomacPokretanje`, додајући податке о последипломцу. Понашање објекта одређено је методима у класи. Методи могу дејствовати на дати објекат и својим дејством мењати његово унутрашње стање. Ово се види код употребе метода `postaviBrojIspita()` у објекту класе `Poslediplomac`. Уместо пређашње измене атрибута директно над објектом (на пример, `brojIndeksa` код објекта класе `Student`), сада се то ради посредно кроз позивање метода.

```
class StudentPosllediplomacPokretanje {
	public static void main(String[] args) {
		int n = 7;
		Student prvi = new Student();
		prvi.ime = "Марко Илић";
		prvi.brojIndeksa = 243;
		prvi.naBudzetu = false;

		Student drugi;
		drugi = new Student();
		drugi.ime = "Ана Сарић";
		drugi.brojIndeksa = 25;
		drugi.naBudzetu = true;

		prvi.stampajPodatke();
		drugi.stampajPodatke();

		Poslediplomac novi = new Poslediplomac();
```

```
                novi.postavibrojIspita(n);
                novi.ime = "Петар Перић";
                novi.brojIndeksa = 4;
                novi.naBudzetu = true;
                novi.stampajPodatke();
                System.out.println(novi.ime + " је положио "
                  + novi.uzmibrojIspita() + " испита.");
        }
}
```

Покретањем овог програма добија се следећи резултат.

```
Име студента је: Марко Илић, број индекса је: 243 На буџету: false.
Име студента је: Ана Сарић, број индекса је: 25 На буџету: true.
Име студента је: Петар Перић, број индекса је: 4 На буџету: true.
Петар Перић је положио 7 испита.■
```

У горњем примеру, инстанца класе `Student` садржи у себи (изложен као атрибут `ime`) инстанцу класе `String`. Такав однос се назива **садржавање**. Садржавање је једна врста асоцијације међу објектима и она описује однос који може бити именован на два начина:

- «јесте део» када се посматра са позиције дела (на пример, «мотор је део аутомобила»).
- «садржи» када се посматра са позиције целине (на пример, «аутомобил садржи мотор»).

Код односа садржавања се може рећи да је целина састављена (тј. компонована) од делова, па се у литератури оваква врста односа још назива и **композиција**.
У домену објектно оријентисаног програмирања, аутори предлажу да се приликом креирања класа, фаворизује композиција у односу на наслеђивање.

2.3.4. Динамичко (касно) везивање

Динамичко (касно) везивање података је процес у којем се током извршавања програма, на основу поруке коју је примио објекат и информација о типу објекта, одређује која операција ће се извршити и која ће меморијска локација бити резервисана за објекат. У програмском језику C се процес динамичког (касног) везивања назива динамичка алокација меморије (функције `malloc()`, `realloc()`, `calloc()`).
Насупрот динамичком везивању је **статичко (рано) везивање** у којем се везивање података за меморијску локацију, као и начин оперисања са њима, одређују у фази превођења програма (ранија фаза обраде програма). У програмском језику C се процес статичког (раног) везивања назива статичка алокација меморије.
Динамичко везивање је карактеристично за објектно оријентисано програмирање. Треба напоменути да динамичко везивање утиче на нешто спорије извршавање програма.
Као што се може уочити у примеру 2, у класи `StudentPoslediplomacPokretanje` променљива `n` се везује за локацију у фази превођења, што значи да се врши статичко везивање. Инстанце `prvi`, `drugi` и `novi` се креирају у фази извршавања програма, дакле, на њих се примењује динамичко везивање. Ово је могуће (и рано и касно везивање у једном програму), јер Јава није чисто објектно оријентисан језик.

2.3.5. Учауривање

Учауривањем се скривају подаци унутар објекта и онемогућава неконтролисан приступ овим подацима. На тај начин спречава се тзв. домино-ефекат (енг. side effect) који може довести до некоректног рада програма због отворене могућности да се из спољашњости могу мењати подаци дати унутар неке целине.
Учауривањем се може постићи и боља расподела функционалности међу класама. На тај начин свака класа, односно њени објекти, могу да располажу само одређеним подацима (атрибутима) који су им природно додељени. Ово онемогућава мешање објекта једне класе у функционалности објеката друге класе, јер се функционалности реализују тако да модификују стање објекта, односно доступне податке.
Пример 3. Нека су дате две класе `Krug` и `Crtez`, при чему су на цртежу нацртана два круга. Природно је да `Krug` садржи учаурене податке који га описују, на пример центар круга и полупречник. `Crtez` садржи, као атрибуте, два круга. Објекат класе `Crtez` не треба да има директан приступ подацима који описују круг, већ да им приступа или манипулише њима искључиво позивањем метода над објектима класе `Krug`, на пример, метод `transliraj()`. `Crtez` има могућност транслирања свих кругова за дати вектор помераја. □

```
class Krug {
    int cx, cy;
    int r;

    Krug(int cx, int cy, int r) {
        this.cx = cx;
        this.cy = cy;
        this.r = r;
    }

    void transliraj(int dx, int dy) {
        cx += dx;
        cy += dy;
    }

    void prikazi() {
        System.out.println("Круг C = (" + cx + ", " + cy + ") и r = " + r);
    }
}
```

Као што се види испод, класа `Crtez` не задире у унутрашње стање круга директним приступом атрибутима `cx`, `cy` или `r`, већ то ради индиректно путем метода `transliraj()`.

```
class Crtez {
    Krug krug1;
    Krug krug2;

    public Crtez() {
        krug1 = new Krug(10, 20, 11);
        krug2 = new Krug(17, 20, 4);
    }

    void translirajSve(int dx, int dy) {
        krug1.transliraj(dx, dy);
```

```
        krug2.transliraj(dx, dy);
    }

    void prikazi() {
        krug1.prikazi();
        krug2.prikazi();
    }
}
```

Приметити да је учауривање функционалности присутно и код метода `prikazi()`. Метод `prikazi()` у класи `Crtez` има за циљ текстуални приказ садржаја цртежа, док је код класе `Krug` циљ приказ информација о конкретном кругу. Оваква подела одговорности поједностављује кôд и побољшава његову прегледност – посебно у ситуацијама када постоје врло дубоки објекти, тј. објекти који имају своје подобјекте, а њихови подобјекти своје подобјекте итд.

```
class CrtezPokretanje {
    public static void main(String[] args) {
        Crtez crtez = new Crtez();
        System.out.println("Пре транслације:");
        crtez.prikazi();
        crtez.translirajSve(11, -3);
        System.out.println("После транслације:");
        crtez.prikaziSadrzaj();
    }
}
```

Извршавањем овог програма добија се резултат приказан испод.

```
Пре транслације:
Круг C = (10, 20) и r = 11
Круг C = (17, 20) и r = 4
После транслације:
Круг C = (21, 17) и r = 11
Круг C = (28, 17) и r = 4■
```

Пример 4. Написати Јава програм који одређује НЗД и НЗС за три броја. Рачунање НЗД и НЗС треба да буде учаурено унутар објекта.□

Класа у којој су учаурене атрибут вредности броја, методи за рачунање НЗД и НЗС, као и метод за приказ броја, налазе се у датотеци `CeoBroj.java`.

```
class CeoBroj {
    int vrednost;

    CeoBroj(int vrednostBroja) { vrednost = vrednostBroja; }

    void prikazi() {
        System.out.print(vrednost);
        System.out.println();
    }

    CeoBroj NZD(CeoBroj drugi) {
        int prvaVrednost = vrednost;
        int drugaVrednost = drugi.vrednost;
        while (true)
            if (prvaVrednost > drugaVrednost) {
                if (prvaVrednost % drugaVrednost == 0)
```

```
                    return new CeoBroj(drugaVrednost);
                prvaVrednost %= drugaVrednost;
            } else {
                if (drugaVrednost % prvaVrednost == 0)
                    return new CeoBroj(prvaVrednost);
                drugaVrednost %= prvaVrednost;
            }
    }

    CeoBroj NZS(CeoBroj drugi) {
        CeoBroj nzd = this.NZD(drugi);
        int nzs = (vrednost * drugi.vrednost) / nzd.vrednost;
        return new CeoBroj(nzs);
    }
}
```

Решење за НЗД је засновано на модификованом Еуклидовом алгоритму, према ком је НЗД за два броја једнак НЗД мањег од та два и остатка при дељењу већег мањим. НЗС се рачуна по познатој формули $\text{НЗС}(m,n) = \frac{m \times n}{\text{НЗД}(m,n)}$.

С обзиром да су сад и подаци и понашање учаурени у објекту, позивање НЗД и НЗС је такво да се над првим бројем позива метод и прослеђује други број, а потом на резултат тог извршавања, што је такође објекат класе `CeoBroj`, позива поново метод и прослеђује трећи број. Програмски кôд за покретање се налази у засебној датотеци `ObjektnoNzdNzsPokretanje.java`.

```
class ObjektnoNzdNzsPokretanje {
    public static void main(String[] args) {
        CeoBroj prvi = new CeoBroj(48);
        CeoBroj drugi = new CeoBroj(120);
        CeoBroj treci = new CeoBroj(56);
        System.out.print("Први број је ");
        prvi.prikazi();
        System.out.print("Други број је ");
        drugi.prikazi();
        System.out.print("Трећи број је ");
        treci.prikazi();
        CeoBroj nzd = prvi.NZD(drugi).NZD(treci);
        CeoBroj nzs = prvi.NZS(drugi).NZS(treci);
        System.out.print("НЗД ова три броја је ");
        nzd.prikazi();
        System.out.print("НЗС ова три броја је ");
        nzs.prikazi();
    }
}
```

Извршавањем овог програма добија се резултат приказан испод.

```
Први број је 48
Други број је 120
Трећи број је 56
НЗД ова три броја је 8
НЗС ова три броја је 1680■
```

2.4. Предности и мане објектно оријентисаног програмирања

Парадигму објектно оријентисаног програмирања карактеришу многобројне предности:

- Олакшана анализа, пројектовање и програмирање – претходно описане карактеристике омогућују да се приликом анализе елементи домена пословања и њихови односи лако моделирају преко објеката, да се при пројектовању могу апстраховати одређени елементи и концепти, а да се при програмирању може креирати једноставан и чист програмски кôд.
- Олакшано одржавање софтвера – по правилу, једноставан и чист програмски кôд, у коме су подржани пословни процеси описани интеракцијом између објеката бива много погоднији за одржавање и евентуалну даљу надоградњу .
- Омогућава лако и једноставно уклапање модула – како објекти дају јасну подршку за учаурење, то се и организовање кода по модулима природно може реализовати ослањајући се на то како су организоване класе, а тако дефинисани модули се једноставно и лако уклапају .
- Поновна искористивост софтвера – једноставан и чист програмски кôд, који најчешће бива резултат правилне примене принципа и пракси објектно оријентисаног програмирања се лако може поновно искористити – по правилу, много лакше него кôд креиран помоћу других парадигми програмирања.
- Најпогоднији за симулирање догађаја – посматрање пословних процеса као интеракције међу објектима се изузетно добро уклапа у ситуације када треба моделирати догађаје.
- Олакшава паралелни рад више програмера на истом пројекту – када се дефинише начин интеркације међу објектима, одвојени делови програмерског тима паралелно могу радити на реализацији различитих објаката, не сметајући једни другима.

Иако правилно примењена парадигма објектно оријентисаног програмирања пружа значајне преднсти, уочени су и одређени њени недостаци:

- Може да доведе до дуплирања кода.
- Због динамичког везивања, може доћу до споријег извршавања програма.
- Није погодна за све типове проблема (на пример, проблеми у којима су непходна интезивна нумеричка израчунавања обично немају користи од примене објектно оријентисане парадигме).
- Програмирање засновано на интеракцији објеката може бити компликованије, јер се може догодити да се из програмског кода не види директно след интеракција међу објектима.

2.5. Резиме

Објектно оријентисано програмирање омогућава формулисање интуитивне и прегледне структуре решење посматраног проблема. Помоћу ООП, ентитети и интеракције међу ентитетима могу се пренети из реалног света у програмски кôд.

Помоћу наслеђивања могуће је избећи вишеструко дефинисање кода који извршава исту или сличну функционалност.

ООП тренутно представља најзаступљенију програмску парадигму у контексту апликативног софтвера, а велика је вероватноћа да ће тако остати и у будућности.

Евентуални недостаци, на пример нешто спорије извршавање програма, заобилазе се применом различитих програмских технологија за различите делове софтвера, и њиховим каснијим уклапањем у целину.
У последње време заступљено је и комбиновање ООП језика са другим језицима. Најбољи пример је комбиновање ООП језика са функционалним језиком зарад омогућавања комбинованог модела императивно-декларативног програмирања у ООП језику.

2.6. Питања и задаци

1. Која је основна идеја објектно оријентисане парадигме?
2. Кратко описати историјат објектно оријентисаног програмирања.
3. Шта је класа, а шта је инстанца/конкретан објекат класе?
4. Објаснити механизам наслеђивања и навести конкретне примере.
5. Упоредити динамичко и статичко везивање.
6. Шта се подразумева под учаурењем објекта?
7. Које су предности, а које недостаци објектно оријентисаног програмирања?

3. Неке програмске парадигме

Са развојем рачунарства појавили су се и различити начини програмирања. Фундаментални стил програмирања, који се користи, назива се програмска парадигма. Назив потиче од грчке речи парадигма коју можемо превести као: образац, шаблон, пример за углед, … До сада је настао већи број програмских парадигми.
Иако је у центру проучавања овог уџбеника објектно оријентисана програмска парадигма, биће кратко приказане и друге програмске парадигме (императивна, структурна, модуларна и функционална). Решавајући исти задатак коришћењем различитих парадигми, биће демостриране неке карактеристике тих парадигми и приказано како се те парадигме могу користити у програмском језику Јава.
Задатак: Написати Јава програм који одређује највећи заједнички делилац (НЗД) за три броја.□
У свим решењима, која ће бити дата у наставку, кључну улогу има начин на који се одређује НЗД за два броја (претпостављамо да су улазни подаци природни бројеви). Тај начин је битно скопчан са парадигмом која се користи. Када се одреди НЗД за два броја, одређивање НЗД за три броја је мање-више исто у свим парадигмама.

3.1. Императивно програмирање

Основни појам на којем се заснива императивна парадигма је команда или наредба и отуд потиче њен назив — програму се директно наређује шта треба да ради. Поред тога, битан је и редослед команди, које се сврставају у процедуре, па се ова парадигма још назива и процедурална у неким поделама (према неким другим, процедурална се сматра варијантом императивне). За класично императивно програмирање (које се понекад још назива и операционо) карактеристично је да се `goto`-наредба може користити без ограничења.
Решење 1. Користећи императивну парадигму**,** програм ће бити реализован као монолитна целину. НЗД за три броја се одређује тако што се прво одреди НЗД за прва два броја, а коначни резултат се добија тако што се одреди НЗД за претходно одређени НЗД прва два броја и за трећи број. Прво одређивање НЗД је реализовано коришћењем Еуклидовог алгоритма, који се ослања на тврдњу да је НЗД два различита броја исти као НЗД мањег од та два броја и њихове разлике. Друго одређивање НЗД је реализовано коришћењем модификованог Еуклидовог алгоритма, који се од оригиналног разликује по томе што се уместо разлике рачуна остатак при дељењу - чиме се постиже бржа конвергенција.
Програм је реализован монолитно, тј. алгоритам је реализован као јединствена целина, у датотеци `ImperativnoNzd.java`. С обзиром да програмски језик Јава не подржава `goto` наредбу, уместо те наредбе је коришћена обележена `break` наредба. Следи програмски кôд.

```
class ImperativnoNzd{
   public static void main(String[] args) {
        int prviBroj = 48;
        int drugiBroj = 120;
        int treciBroj = 56;
        System.out.println("Први број је " + prviBroj);
        System.out.println("Други број је " + drugiBroj);
        System.out.println("Трећи број је " + treciBroj);
```

```
        nzdPrviDrugi:
        for (; ; ) {
            if (prviBroj == drugiBroj) break nzdPrviDrugi;
            if (prviBroj > drugiBroj) {
                int privremeni = prviBroj;
                prviBroj = drugiBroj;
                drugiBroj = privremeni;
            }
            drugiBroj = drugiBroj - prviBroj;
        }
        nzdNadPrvaDvaTreci:
        for (; ; ) {
            if (prviBroj == treciBroj) break nzdNadPrvaDvaTreci;
            if (prviBroj > treciBroj) {
                int privremeni = prviBroj;
                prviBroj = treciBroj;
                treciBroj = privremeni;
            }
            if( treciBroj % prviBroj == 0 ) break nzdNadPrvaDvaTreci;
            treciBroj = treciBroj % prviBroj;
        }
        System.out.println("НЗД ова три броја је " + prviBroj);
    }
}
```

Прва `for` петља је посвећена тражењу НЗД за прва два броја, док је друга посвећена тражењу НЗД за сва три броја (односно претходно одређеном НЗД за прва два броја и трећем броју). У оба случаја се петља завршава када су бројеви исти (обележена `break` наредба). Уколико то није случај, итерација петље се извршава тако што се већи од два броја замењује разликом већег и мањег (или остатком при дељењу већег са мањим).
Приликом извршавања овог Јава програма добија се следећи испис:

```
Први број је 48
Други број је 120
Трећи број је 56
НЗД ова три броја је 8■
```

3.1.1. Структурно програмирање

Структурно програмирање је настало из процедуралног тако што је ограничена (или потпуно онемогућена) употреба `goto`-наредбе. Дозвољена је употреба ограниченог броја управљачких структура попут гранања, циклуса (петљи) и процедура са циљем побољшања јасноће и квалитета програма, као и смањења времена развоја и каснијег одржавања. Структурна парадигма се може третирати као потпарадигма процедуралне парадигме.
Решење 2. Претходно постављени задатак се може решити применом структурног програмирања. НЗД се одређује коришћењем модификованог Еуклидовог алгоритма, описаног у претходном решењу. Рачунање НЗД за два и за три броја је издвојено у посебне методе, чиме је постигнуто да програмски кôд буде прегледнији, читљивији и лакши за одржавање и надоградњу.
Програмски кôд је дат у датотеци `StrukturnoNzd.java`.

```
class StrukturnoNzd {
    static int nzd2(int prvi, int drugi) {
```

```
        while (true)
            if (prvi > drugi) {
                if (prvi % drugi == 0) return drugi;
                prvi %= drugi;
            } else {
               if (drugi % prvi == 0) return prvi;
                drugi %= prvi;
            }
    }

    static int nzd3(int prvi, int drugi, int treci) {
        return nzd2(nzd2(prvi, drugi), treci);
    }

    public static void main(String[] args) {
        int prviBroj = 48;
        int drugiBroj = 120;
        int treciBroj = 56;
        System.out.println("Први број је " + prviBroj);
        System.out.println("Други број је " + drugiBroj);
        System.out.println("Трећи број је " + treciBroj);
        int nzd = nzd3(prviBroj, drugiBroj, treciBroj);
        System.out.println("НЗД ова три броја је " + nzd);
    }
}
```

Примећује се да је ово решење краће од претходног. Главни разлог је елиминација дуплираног (или бар довољно сличног) кода.
Приликом извршавања горњег програма добија се исти резултат који је добијен у решењу који се односи на императивно програмирање.■

3.1.2. Модуларно програмирање

Модуларно програмирање се може третирати као потпарадигма процедуралне парадигме. Ова парадигма је усвојила све добре стране структурне и заснива се на интензивном коришћењу мањих програмских целина — модула. Модули се користе за апстракцију (уопштавање) појма податак.
Решење 3. Постављени задатак може се решити применом модуларне парадигме. НЗД се одређује по истом алгоритму као у решењу 1. Рачунање НЗД за два броја реализује се преко метода `nzd2()`, који се налази у модулу у датотеци `ModulNzd.java`. У истом модулу се налази и метод за рачунање НЗД за три броја, под називом `nzd3()`.

```
class ModulNzd {
    static int nzd2(int prvi, int drugi) {
        while (true)
            if (prvi > drugi) {
                if (prvi % drugi == 0) return drugi;
                prvi %= drugi;
            } else {
                if (drugi % prvi == 0) return prvi;
                drugi %= prvi;
            }
    }

    static int nzd3(int prvi, int drugi, int treci) {
```

```
        return nzd2(nzd2(prvi, drugi), treci);
    }
}
```

Главни програм је другом модулу, тј. у датотеци `ModulNzdPokretanje.java`.

```
class ModulNzdPokretanje {
    public static void main(String[] args) {
        int prviBroj = 48;
        int drugiBroj = 120;
        int treciBroj = 56;
        System.out.println("Први број је " + prviBroj);
        System.out.println("Други број је " + drugiBroj);
        System.out.println("Трећи број је " + treciBroj);
        int nzd = ModulNzd.nzd3(prviBroj, drugiBroj, treciBroj);
        System.out.println("НЗД ова три броја је " + nzd);
    }
}
```

Приликом извршавања овог програма, добија се исти резултат као у решењу 1.
Напомена: програмски језик Јава је реализован тако да се свака датотека са изворним кодом независно преводи – самим тим, она представља модул у „класичном“ смислу[1].
■

3.2. Декларативно програмирање

Са развојем рачунарства софтвер постаје све комплекснији. Нове програмске парадигме појављују се уз тежњу да се олакша процес програмирања и креирање софтвера учини ефикаснијим. У претходно наведеним парадигмама рачунару се, преко програма, саопштава **како** се проблем решава.
Да ли је могуће, преко програма, описати (декларисати) проблем како би, на основу тог описа, рачунар сам генерисао решење?
Овакав приступ програмирању подржан је преко тзв. декларативних (описних) парадигми. У деклативне парадигме спадају функционална и логичка. За ове парадигме карактеристично је да се преко програма саопштава **шта** се решава, а програм уз подршку програмског језика, треба да има механизам за налажење решења. Понављање операција је веома битно за електонске рачунаре. Код императивних парадигми природан начин за понављање операција је итерација, док је код деклартивних парадигми то рекурзија.

3.2.1. Логичко програмирање

Логичка парадигма заснована је на аутоматском доказивању теорема (једној области вештачке интелигенције). Основни појмови логичке парадигме су: аксиоме (чињенице), правила закључивања и упити. Логичка парадигма омогућава екстракцију нових знања из чињеница и правила закључивања. Програм се реализује постављањем упита и притом се аутоматски претражује скуп чињеница уз коришћење одређених правила закључивања. Због овог специфичног начина решавања проблема логичку парадигму је најтеже симулирати у Јави.

[1] Појам модула дефинисан је у верзији Јава 9, 2017. године и односи се на паковање извршног кода. Овако дефинисан појам модула нема везе са концептом модуларног програмирања.

Решење 4. Користиће се рекурзивна верзија модификованог Еуклидовог алгоритма, где се уместо разлике рачуна остатак при дељењу и одређује НЗД за тај остатак и мањи од аргумената. Правила из логичког програмирања се симулирају преко метода. Ако, краће, са `m` и `n` означимо дате бројеве онда преко Јава метода записујемо одговарајућа логичка правила закључивања:

$$m \bmod n = 0 \Rightarrow nzd(m,n) = n$$
$$n \bmod m = 0 \Rightarrow nzd(m,n) = m$$
$$m > n \Rightarrow nzd(m,n) = nzd(m \bmod n, n)$$
$$m < n \Rightarrow nzd(m,n) = nzd(m, n \bmod m).$$

Цео програм налази се у датотеци `LogickoNzd.java`.

```
class LogickoNzd {
    static int nzd2(int prvi, int drugi) {
        if (prvi % drugi == 0) return prvi;
        if (drugi % prvi == 0) return drugi;
        if (prvi > drugi) return nzd2(prvi % drugi, drugi);
        return nzd2(prvi, drugi % prvi);
    }

    static int nzd3(int prvi, int drugi, int treci) {
        return nzd2(nzd2(prvi, drugi), treci);
    }

    public static void main(String[] args) {
        int prviBroj = 48;
        int drugiBroj = 120;
        int treciBroj = 56;
        System.out.println("Први број је " + prviBroj);
        System.out.println("Други број је " + drugiBroj);
        System.out.println("Трећи број је " + treciBroj);
        int nzd = nzd3(prviBroj, drugiBroj, treciBroj);
        System.out.println("НЗД ова три броја је " + nzd);
    }
}
```

Примећује се да Јава метод `nzd2()` симулира горе наведена четири правила закључивања. Тражење НЗД за три броја је, као и раније, реализовано двоструким позивом метода за тражење НЗД за два броја – ово је природно и у логичкој парадигми. Приликом тестирања наведеног програма добија се исти резулат као и у претходним решењима.■

3.2.2. Фунционално програмирање

Функционална парадигма је заснована на математичком појму „функција“ и отуд потиче њен назив. Може се окарактерисати реченицом: „Израчунај вредност израза и користи је за нешто“. Овде је функција представљена изразом и реализује се израчунавањем израза. Сва израчунавања у програму испуњавају се применом (позивом) функција. Како се методи у неким објектно оријентисаним језицима називају функције, преко метода у Јави се могу симулирати функције из функционалне парадигме. У Јави постоји могућност коришћења ламбда-функције која је каратеристична за функционални стил програмирања.

Решење 5. Претходно постављени задатак биће решен применом функционалног стила програмирања. Користиће се рекурзивна дефиниција НЗД за два природна броја, која гласи:

$$nzd(n,m) = n, \qquad za\ m = n$$
$$nzd(n,m) = nzd(m-n,n), \quad za\ m > n$$
$$nzd(n,m) = nzd(n-m,m), \quad za\ m < n$$

Претходна дефиниција се користи за креирање функције смештене у променљиву `nzd2`. Овде је функција „грађанин првог реда“ — са њом се може радити слично како се ради са другим објектима (слично показивачима на функције у C-у). Цео програм налази се у датотеци `FunkcionalnoNzd.java`.

```
class FunkcionalnoNzd {
    static java.util.function.BiFunction<Integer, Integer, Integer> nzd2;
    static {
        nzd2 = (prvi, drugi) -> (prvi.compareTo(drugi) == 0)
                ? prvi
                : ( (prvi.compareTo(drugi) < 0)
                        ? nzd2.apply(drugi - prvi, prvi)
                        : nzd2.apply(prvi - drugi, drugi) );
    }

    public static void main(String[] args) {
        int prviBroj = 48;
        int drugiBroj = 120;
        int treciBroj = 56;
        System.out.println("Први број је " + prviBroj);
        System.out.println("Други број је " + drugiBroj);
        System.out.println("Трећи број је " + treciBroj);
        int nzd = nzd2.apply( prviBroj, nzd2.apply(drugiBroj, treciBroj) );
        System.out.println("НЗД ова три броја је " + nzd);
    }
}
```

Тестирањем наведеног програма добија се исти резулат као и у претходним решењима.■

3.2.3. Објектно оријентисано програмирање

На крају другог поглавља дато је решење постављеног задатка применом објектно оријентисаног програмирања – пре свега је демонстриран концепт учауривања.
Међутим, у методу за налажење НЗД тада су били коришћени примитивни типови Јаве (слични основним типовима који постоје у C-у).

У чисто објектно оријентисаним језицима оперише се само са објектима и поставља се питање да ли је могуће релизовати решење постављеног задатка у Јави, али оперишући само са објектима.

Решење 6. Одговор је потврдан и то је урађено помоћу програма у датотеци `ObjektnoNzd.java`.

```
class ObjektnoNzd {
    public static void main(String[] args) {
        Integer prviBroj = new Integer(48);
        Integer drugiBroj = new Integer(120);
        Integer treciBroj = new Integer(56);
```

```
        System.out.println("Први број је " + prviBroj.toString());
        System.out.println("Други број је " + drugiBroj.toString());
        System.out.println("Трећи број је " + treciBroj.toString());
        Integer nzd = nzd3(prviBroj, drugiBroj, treciBroj);
        System.out.println("НЗД ова три броја је " + nzd.toString());
    }

    static Integer nzd2(Integer prvi, Integer drugi) {
        if (prvi.compareTo(drugi) == 0) return prvi;
        else {
               if (prvi.compareTo(drugi) < 0) {
                      Integer novi = new Integer(drugi.intValue()
                         - prvi.intValue());
                      return nzd2(novi, prvi);
               } else {
                      Integer novi = new Integer(prvi.intValue()
                         - drugi.intValue());
                      return nzd2(novi, drugi);
               }
        }
    }

    static Integer nzd3(Integer prvi, Integer drugi, Integer treci) {
        return nzd2(nzd2(prvi, drugi), treci);
    }
}
```

У наведеном решењу се оперише само са објектима, али не може се рећи да је решење елегантно и да се проблем ефикасно решава на овај начин. Генерално, објектно оријентисани језици нису погодни за решавње проблема у којима су потребна интензивна нумеричка израчунавања. Стога популарни пограмски језици, као што су Јава и C++, поред објектне оријентисаности имају и процедуралне карактеристике.

Још једна разлика у односу на решење из претходног поглавља је у томе што се овде не користи у довољној мери концепт учауривања. Дакле, иако ово решење користи само објекте, што је главна претпоставка чистих објектно оријентисаних језика, оно одступа по питању других ООП принципа — методи `nzd2()` и `nzd3()` не оперишу над унутрашњим стањем припадајућег објекта, већ само над аргументима које прихватају.

Тестирањем наведеног програма добија се исти резулат као и у претходним решењима.■

Напомена. Ако се може одредити НЗД за два, тј. за три броја, онда се лако може израчунати и најмањи заједнички садржалац (НЗС) за ова три броја. Израчунавање се слично реализује у свим парадигмама. На пример, у модуларној парадигми израчунавање се може реализовати преко посебног модула.

```
class ModulNzs {
    static int nzs2(int prvi, int drugi) {
        return (prvi * drugi) / ModulNzd.nzd2(prvi, drugi);
    }

    static int nzs3(int prvi, int drugi, int treci) {
        return nzs2(nzs2(prvi, drugi), treci);
    }
}
```

Потребно је у главном програму позвати метод `nzs3()` за три броја и одштампати резултат.

```
class ModulNzdNzsPokretanje {
    public static void main(String[] args) {
        int prviBroj = 48;
        int drugiBroj = 120;
        int treciBroj = 56;
        System.out.println("Први број је " + prviBroj);
        System.out.println("Други број је " + drugiBroj);
        System.out.println("Трећи број је " + treciBroj);
        int nzd = ModulNzd.nzd3(prviBroj, drugiBroj, treciBroj);
        System.out.println("НЗД ова три броја је " + nzd);
        int nzs = ModulNzs.nzs3(prviBroj, drugiBroj, treciBroj);
        System.out.println("НЗС ова три броја је " + nzs);
    }
}
```

Приликом извршавања овог програма, добија се следећи резултат.

```
Први број је 48
Други број је 120
Трећи број је 56
НЗД ова три броја је 8
НЗС ова три броја је 1680
```

3.3. Резиме

У овом поглављу су анализирана два основна приступа решавању проблема. Први је тзв. императивни приступ, у којем је програмер надлежан да прецизира сваки корак који доводи до решавања проблема. Мисаони процес који наводи програмера да реши проблеме овим приступом је сличан оном који користи инжењер – целокупни проблем се подели на мање потпроблеме након чега се они независно решавају и потом уклапају у решење целокупног проблема.

Императивни изворни кôд може имати различите степене организованости и структуираности: почев од данас напуштеног приступа који користи `goto` наредбу до модерног приступа који је врло структиран и модуларан.

Други, фундаментално другачији приступ програмирању, је декларативан. Овај приступ не захтева од програмера прецизирање свих корака, већ детаљан опис проблема – отуда и назив декларативно.

Да би декларативни приступ могао да функционише потребно је да постоји одговарајући механизам, који ће опис проблема да трансформише у низ одговарајућих корака који решавају тај проблем: код логичког програмирања је основни механизам тзв. метод резолуције, док је функционално програмирање засновано на математичком концепту функције, њене композиције и рекурзије. Треба имати у виду да се на модерним рачунарима сваки декларативни кôд на крају своди на императивни, јер је машински језик у основи императиван.

3.4. Питања и задаци

1. Упоредити императивну и декларативну парадигму.
2. Написати програм који испитује да ли су два броја узајамно проста:

   - императивном парадигмом;
   - структурном парадигмом;
   - модуларном парадигмом,
   - логичком парадигмом;
   - функционалном парадигмом.
3. Написати програм који имплементира quick sort алгоритам у:
   - императивној парадигми;
   - функционалној парадигми.

   Анализирати написане програме.
4. За сваку од програмских парадигми, разматраних у овом поглављу, наћи примере програмских језика који се заснивају на њој.
5. Истражити које још програмске парадигме, поред наведених, постоје.

4. Карактеристике програмског језика Јава

У овом поглављу читалац ће се упознати са историјом развоја програмског језика Јава, основним принципима којима се водио њен развој као и инфраструктуром над којом је она заснована.

4.1. Историјат и развој програмског језика и окружења Јава

У овој секцији биће направљен хронолошки преглед Јава технологија, почев од њеног настанка па до актуелне верзије Јаве у моменту припреме овог рукописа. Читаоца не треба да брине евентуално непознавање великог броја термина, јер технолошки оквири Јаве су јако широки и дубоки, намењени разним платформама и потребама. Читалац се на ову секцију може вратити и касније када буде боље упознат са Јава технологијама.

Почетак и рани развој

Јава обухвата програмски језик и платформу за извршавање и развој. Она је производ компаније Sun Microsystems, сада у власништву компаније Oracle. Значајни делови Јаве су јавно доступни као софтвер отвореног кода (енг. open source).

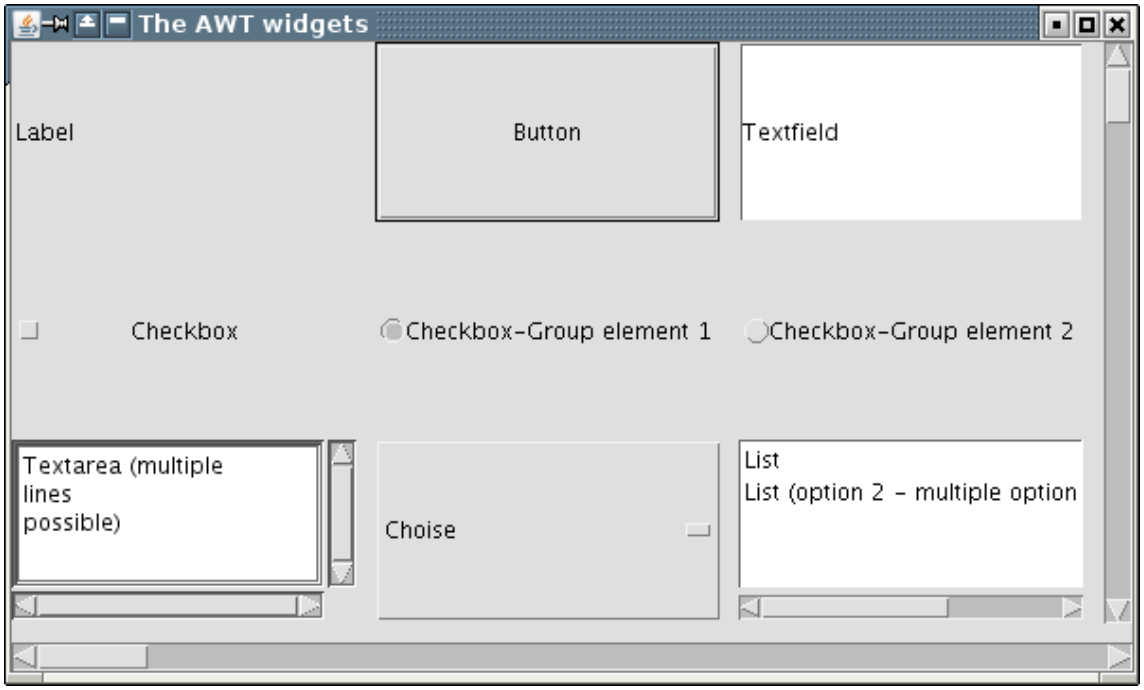
Претеча Јаве настала је 1991. године – она је била намењена мрежном кућном окружењу (пројекат Oak, под руководством Џејмса Гослинга). Сходно трендовима тог времена, развој Јаве се усмерио ка програмирању на Интернету (и дан-данас се Јава успешно користи у том домену). Језик Јава је лансиран 1995. године, на SunWorld конференцији. Одмах је постигао велику популарност: Netscape прегледачи (који су навелико коришћени у том периоду) су почели да користе Јаву, IBM је купио лиценцу, Јаву је користио чак и Microsoft.

Лого Јаве

Компанија Sun, која је произвела и поседовала Јаву, године 1997. развила је **JDK 1.0** (кодно име Oak).

Тако су се уз Јава платформу испоручивали и Јава алати за развој (енг. Java Development Kit — JDK), тј. скуп алата и библиотека који се користи за развој Јава апликација. Изворни кôд испоручених библиотека је био и остао доступан за преглед Јава програмерима.

Године 1997. појавио се **JDK 1.1**, који је садржао JavaBeans, унутрашње класе, JDBC, RMI, рефлексију, а прерађене су класе и алати за програмирање графичког корисничког интерфејса AWT – Abstract Window Toolkit. Ова библиотека садржи и подршку за интернационаизацију, тј. за Unicode знакове.



Илустрација AWT корисничког интерфејса

Нови JDK, означен са **J2SE 1.2** (кодно име Playground), испоручен је 1998. године. Поред стандардне верзије 1.2 тј. по новој номенклатури J2SE 1.2, програмерима су дате на располагање и J2EE 1.2 (компонете и спецификације које олакшавају програмирање сложених пословних апликација), као и J2ME 1.2 (за развој мобилних апликација). Обим је утростручен у односу на претходну верзију, па је J2SE 1.2 садржала Swing библиотеку за програмирање графичког корисничког интерфејса, JIT преводилац за JVM, Java IDL за CORBA и колекције (листе, речнике, скупови).

J2SE 1.3 (кодно име Kestrel) се појављује 2000. године. Ова верзија је пружила подршку за рад у дистрибуираном окружењу (више рачунара комуницирају) RMI за CORBA, рад са звуком — JavaSound, JNDI, JPDA и синтетичке класе-заступнике.

Године 2002. појављује се **J2SE 1.4** (кодно име Merlin), који садржи оператор `assert`, уланчавање изузетака, подршку за IPv6, NIO, logging API, рад са сликама преко Image I/O API, XML и XSLT процесор тј. JAXP, JCE криптографију и Java Web Start.

Верзија Јаве **J2SE 5.0** (кодно име Tiger), иницијално означена бројем 1.5 (што и надаље представља интерну ознаку ове верзије) се јавља 2004. године. Нове могућности, које је донела ова верзија, су: подршка за генеричке типове (што омогућава, на пример, да се алгоритми за различите типове података пишу само једанпут), сигурност типова за колекције, анотације, аутоматско паковање/распакивање типа, набројиви (енумерисани) типови, променљиви аргументи, колекцијски `for` циклус, статички увоз, аутоматска подршка за удаљено позивање метода, подршка за паралелно програмирање, класе за парсирање улаза. Творци Јаве претендују да је ово верзија у којој су први пут настале значајније промене у самом језику и да је изостала подршка неким застарелим концептима.

Године 2006. Sun објављује **Java SE 6** (кодно име Mustang), интерна ознака 1.6.0. Промене укључују убрзање перформанси језгра и Swing графичке библиотеке, побољшане веб сервисе коришћењем JAX-WS, побољшан рад при повезивању са базама података — подршка за JDBC 4.0, укључен API за Јава превођење, додату Јава архитектуру за XML повезивање (JAXB), усавршен GUI и побољшану JVM.

Исте 2006. године, велики део Јаве постаје слободан, отворен и доступан је под GPL лиценцом.

Јава и Oracle

Компанија Oracle 2010. године купује Јаву од од Sun-а — па је од тада Java постала власништво Oracle-а.

Java SE 7 (кодно име Dolphin) појављује се 2011. године. Новине су: JVM подршка за динамичке језике, компресовани 64-битни показивачи, нова I/O библиотека са подршком за мета-податке, XRender ток за Java 2D који убрзава цртање модерним GPU, знаковне ниске као обележја у `switch` наредби, аутоматско управљање ресурсима код `try-catch`, простије декларисање метода са променљивим параметрима, литерали који представљају бинарне бројеве, подвлаке у нумеричким литералима, симултано хватање више врста изузетака и избацивање изузетака уз побољшану контролу типа.

Године 2014. Појављује се **Java SE 8** (кодно име Spider). Промене: подршка за рад са ламбда изразима (аспекти функционалног програмирања), побољшани рад са временом и календарима, ефикаснији Nashorn JavaScript модул, побољшања на пољу сигурности.

У септембру 2017. године појавила се **Java SE 9**. Највећа новост код ове верзије Јаве је платформа система модула (позната и као пројекат Jigsaw). Овакав приступ даје слободу комбиновања програмских модула у разноврсне конфигурације, зависно од потреба. Описане конфигурације су нарочито корисне за креирање, али и за испоруку скалабилних Јава апликација. Дата конфигурација може садржати један мањи скуп модула, а не мора садржавати целокупну Јава платформу као монолитни систем, што је био случај у ранијим верзијама Јаве. На тај начин се постижу боље перформансе, лакше одржавање и сигурност Јава апликације. Модуларна структура је довела до увођења нове URI схеме за именовање модула, класа и ресурса који се смештају у извршним датотекама, чиме је принцип учаурења (енкапсулације) подигнут на нови, виши, ниво. Једноставно исказано, све Јава библиотеке и цела Јава платформа су пребачене на повишен ниво апстракције.

Почев од верзије 9, компанија Oracle је почела са дитрибуирањем OpenJDK као софтвера отвореног кода (са условима лиценцирања сличним као што су код Linux оперативног система). Идеја је да оба производа: Oracle JDK и OpenJDK могу да замене један другог, чиме се пружа могућност програмерима и организацијама које не желе или не могу себи да приуште комерцијалну продршку, нити коришћење сложених алата за управљање.

Јава 10 се појављује у марту 2018. године. Ова верзија Јаве је због промене политике испоруке која се заснива на времену (JEP 322) и кратких циклуса испоруке донела веома мало новина. Међу новоуведеним карактерстикама верзије 10 се издвајају: подршка за одређивање типа локалне променљиве (JEP 286), и унапређење скупљача отпадака (JEP 307). Почев од ове верзије Јаве, Java EE се развија независно, вођена од стране Eclipse Foundation, под именом Jacarta EE.

Нова верзија Јаве, тј. **Јава 11 LTS** се појављује у септембру 2018. године, и то је прва верзија Јаве после верзије 8 која ће бити дугорочно подржана (енг. long term support – LTS). Промене код Јаве 11: уведен је нови скупљач отпадака, тзв. Epsilon garbage collector; уведена је синтакса локалних променљивих за параметре ламбда израза; промењен је формат датотека са бајт-кодом; уклоњени су Java EE и CORBA модули из

Java SE платформе и из JDK; JavaFX је постао софтвер отвореног кода, који се испоручује као самостална библиотека и више није укључен у Oracle JDK.
Мења се начин појаве нових верзија Јаве — одлучено је да ће појава нових верзија бити чешћа, а само неке међу њима ће бити дугорочно подржане (LTS), док ће остале верзије бити подржаване само до изласка нове верзије.
Јава 12 се појављује у марту 2019. године. Неке од промена су: експериментални скупљач отпадака са кратком паузом, switch наредба добија могућност да чини део израза, унапређење процеса компилације JDK, итд.
Јава 13 је због убрзане динамике појаве нових верзија имала врло ситне измене. Она се појавила само неколико месеци након Јаве 12, у септембру 2019. године тако да су веће измене заправо биле поправке претходне верзије. Уведени су текстуални блокови који омогућавају лакши рад са вишелинијским текстом при писању кода.
Јава 14 је донела доста унапређења и поправки, појавила се у марту 2020. године. Неке промене су: интеграција регуларних израза у `instanceof` наредбу, нови алат за паковање (Incubator), информативнији изузеци у случају реферисања `null` референце, прилагођавање ZGC скупљача отпадака за рад под macOS и Windows оперативним системима итд.
Јава 15 се појавила у септембру 2020. године и најинтересантнија промена је увођење запечаћених (`sealed`) класа и интерфејса, које постављају ограничења на то које друге класе или интерфејси их могу наслеђивати.
Јава 16 из марта 2021. године доноси још неке новине попут: Vector API за унапређена векторска израчунавања, миграција OpenJDK пројекта на Git систем за верзионисање и управљање кодом и ресурсима итд.
Јава 17 LTS из септембра 2021. године је друга по реду LTS верзија после верзије 11 из септембра 2018. године. Од побољшања и новина могу се издвојити: побољшани рад са генераторима псеудослучјаних бројева, ојачавање енкапсулације интерних JDK библиотека, подршка за рад са регуларним изразима у оквиру `switch` наредбе итд.

4.2. Постављени циљеви приликом развоја програмског језика и окружења Јава

По речима креатора Јаве, Џејмса Гослинга, на почетку развоја Јаве, пред сам језик и окружење су се поставили разнородни циљеви које је требало испунити.

Највише пажње је посвећено испуњењу следећих циљева.
Једноставност, објектна оријентисаност, и фамилијарност: Јава окружење ће садржавати готове библиотеке за најразличитије намене; језик ће бити објектно орјентисан од самог почетка; синтакса језика ће бити слична синтакси програмских језика C/C++.
Робусност и сигурност: биће омогућено креирање веома поузданог софтвера; биће реализована интензивна провера, како приликом компилације, тако и током извршавања програма.
Једноставан модел управљања меморијом: неће бити показивача, нити показивачке аритметике.

Џејмс Гослинг, вођа тима који је креирао Јаву

Архитектонска неутралност и преносивост: компајлер ће преводити до нивоа бајт-кода (бајт-код није исто што и машински кôд, то је међуформат који је архитектонски неутралан и који је преносив на различите врсте процесора и оперативних система); стриктно ће се дефинисати сам језик: величине простих типова ће бити увек исте без обзира на оперативни систем и архитектуру процесора (на пример, означени цео број ће увек заузимати 4 бајта и број ће бити представљен у потпуном комплементу); извршавање ће бити исто на свакој платформи (за дате улазне податке добијаће се исти излазни подаци - што не важи за програмски језик C).
Перформантност: програм ће се компајлирати до бајт-кода, а потом интерпретирати; интерпретер ће радити пуном брзином, јер су значајне сигурносне провере обављене раније; постојаће аутоматски скупљач отпадака, па програмер неће морати да експлицитно ослобађа меморију; секције са интензивним рачунањем ће моћи да се извезу директно у машински кôд ако је потребно.
Интерпретираност, вишенитност и динамичност: интерпретатор ће моћи да извршава бајт-код на било ком рачунару на који је пренесен систем за извршавање; биће подржано програмирање са више токова извршавања (на пример, код веб прегледача, морају „истовремено“ да: се освежавају графичке компоненте, да се учитава страница и преузима датотека).
Динамичко учитавање класе у току извршавања: класе ће се повезивати и учитавати у систем за извршавање само када се буде појавила потреба за тим.

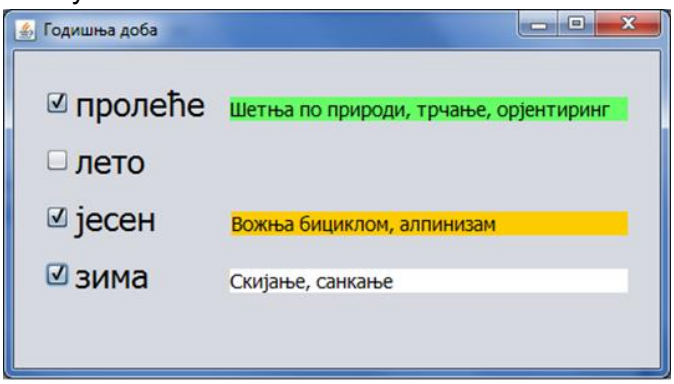
4.3. Типови Јава апликација

Јава је један од најпопуларнијих програмских језика, са широким пољем примене, па не чуди да се коришћењем Јава језика могу креирати различити типови апликација.

4.3.1. Десктоп апликације са графичким корисничким интерфејсом

Јава може бити коришћена за развој преносивих апликација са графичким корисничким интерфејсом (енг. Graphical User Interface) – ГУИ, на свим подржаним платформама. Постоји већи број библиотека које олакшавају ГУИ програмирање у Јави. Најпопуларније су AWT, Swing, SWT и Java FX. Поједини произвођачи софтвера су

креирали сопствене Јава ГУИ библиотеке – тако је, на пример, IBM Јава програмерима понудио SWT библиотеку.



Илустрација ГУИ-а за Јава десктоп апликације развијене уз помоћ Swing-а

4.3.2. Апликације које се покрећу из команде линије

Апликације из командне линије не користе графичке компоненте. То, међутим, не смањује изражајност апликације. Унос и испис се врше путем командне линије уместо путем текстуалних поља, лабела итд.

Илустрација извршавања Јава апликације из командне линије

4.3.3. Апликације за мобилне уређаје

Последњих година је у Јава свету постојала орјентација да се користи Java FX за програмирање мобилних уређаја. Овај тренд је форсирала компанија Oracle, која поседује Јаву, а циљ је био да се униформише развој апликација за све уређаје (телефон, таблет, „класичан“ рачунар)

Међутим, овај приступ није прихваћен од стране заједнице Јава програмера, па је донесена одлука да се у новим верзијама Java FX (донедавно испоручивана заједно са Јавом у оквиру исте инсталације) означи застарелом (енг. deprecated).

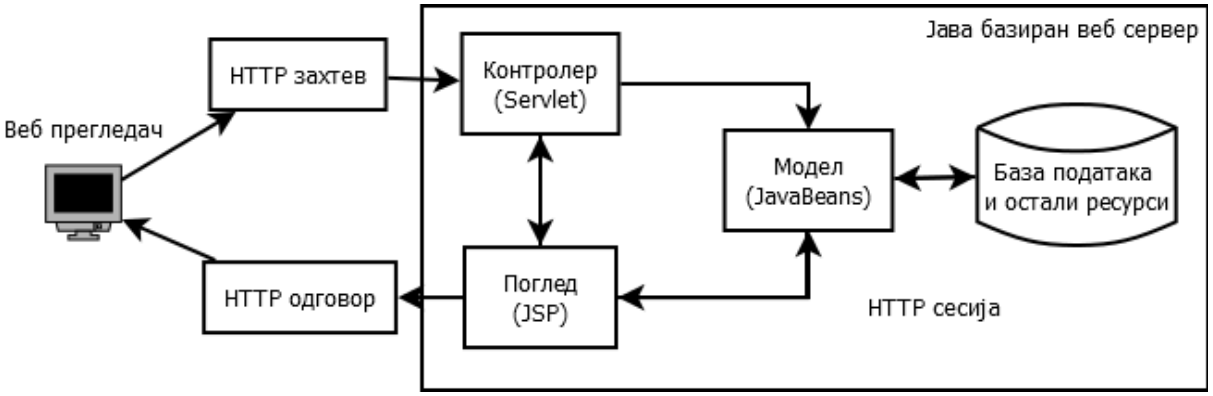
Иако се Java FX само ограничено користи, Јава ипак представља најпопуларнију опцију за развој мобилних апликација на Android паметним телефонима. Може се слободно рећи да је у овом тренутку Јава званичан језик за развој Android софтвера, јер Јава има највећу подршку од стране произвођача Android система (компаније Google) и да је највећи број апликација, које се налазе на Google Play Store, креиран коришћењем Јаве. Наравно, за писање апликација за Android уређаје је, поред језика Јаве, потребно познавати и друге елементе — развојно окружење (на пример, Android Studio), библиотеке, тј. алат за развој Android SDK, алат Gradle, структуру датотека Android Manifest и језик за означавање XML.

4.3.4. Аплети (веб апликације на клијентској страни)

Јава аплети представљају пример тзв. веб програмирања на клијентској страни, што је у овом моменту застарела технологија и ретко се користи. Аплет се, дакле, преузме са сервера, а потом се извршава на клијенту (прегледачу).

4.3.5. Сервлети и Јава серверске стране

Што се Јава извршавања на страни сервера тиче, сценарио је следећи: захтев стиже од клијента, на веб серверу се извршавају наредбе, а потом веб сервер генерише одговор и пошаље га клијенту.



Илустрација извршавања Јава веб апликације која користи образац „Модел-Поглед-Контролер“

Таква врста програмирања зове се програмирање на серверској страни. Већина веб сајтова припада овој групи апликација.

4.3.6. Веб сервиси

Јава омогућује креирање и модификацију веб сервиса. Веб сервиси су клијентске и серверске апликације које комуницирају преко HTTP протокола. Веб сервиси омогућавају комуникацију између апликација које се извршавају на разноврсним платформама. Описи веб сервиса су задати најчешће као XML датотека што омогућава њихову проширивост и динамичност. Пожељно је градити хијерархију веб сервиса у којој они софистицираније користе услуге једноставнијих. Основна предност веб сервиса је интероперабилност, тј. повезивање разноврсних (хетерогених) софтверских система на елегантан начин.

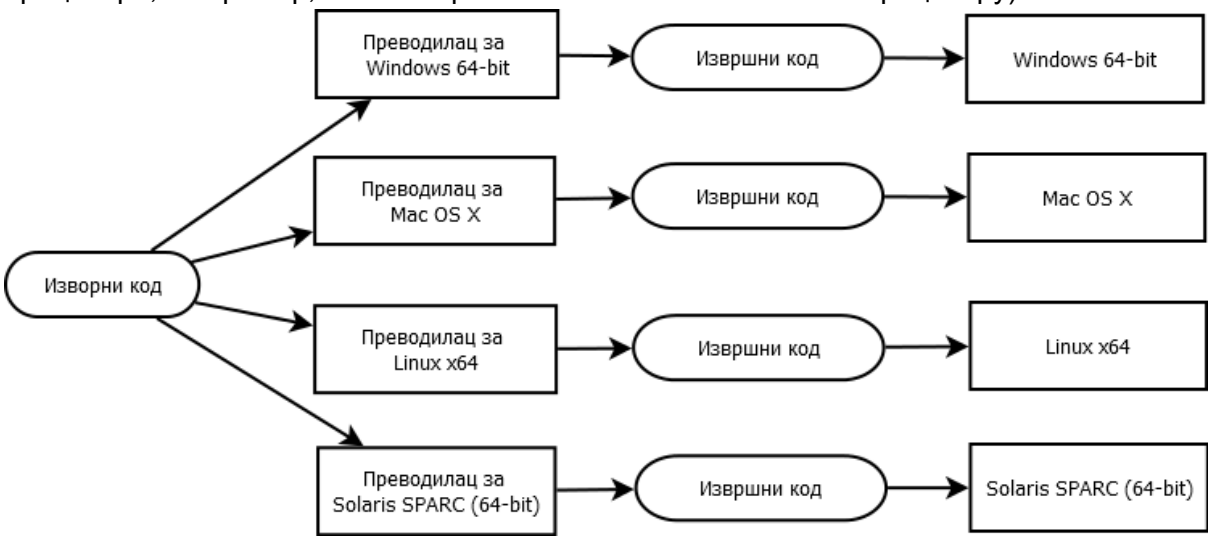
4.3.7. Библиотеке класа

Уместо да програмер само користи постојеће функционалности, тј. библиотеке испоручене приликом инсталације Јаве, он може да оформи и своје. Има смисла правити библиотеку од компоненти које ће се више пута користити у различитим програмима, а које одређују неку функционалност. Програмер те компоненте може спаковати у своју библиотеку и касније их користити у новим пројектима. Библиотека обично садржи већи број сродних функционалности, на пример, библиотека за рад са текстом, библиотека за везу према базама података и слично.

4.4. Процес превођења и извршавања Јава програма

Пре него што се креира апликација, аплет или библиотека у Јави, важно је да се разуме како Јава ради. Јава програм се компајлира, а потом интерпретира. Његово извршавање се спроводи путем тзв. Јава виртуелне машине (JVM).

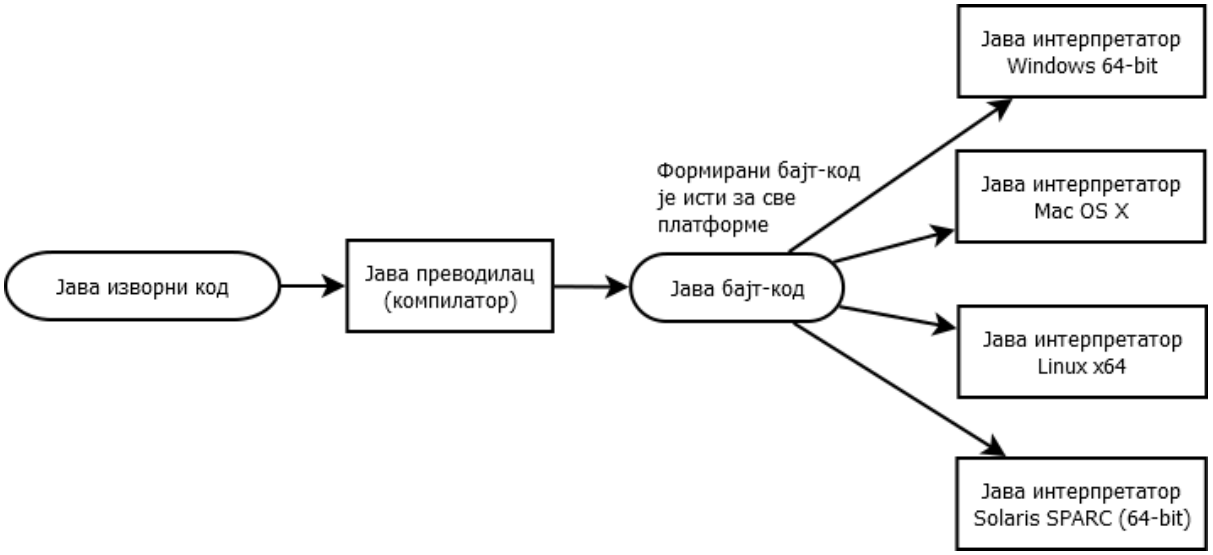
Прво ће бити размотрено како се креира извршни кôд превођењем изворног програма на традиционални начин, коришћењем тзв. „правог преводиоца“ (енг. “true compiler”). У том случају, изворни кôд је јединствен, али извршни кôд зависи од платформе на којој се извршава превођење — исти изворни кôд генерише извршни кôд завистан од платформе (под платформом се мисли на комбинацију оперативног система и типа процесора, на пример, Linux оперативни систем на 64-битном процесору).



Креирање извршног кода од изворног програма приликом коришћења правог преводиоца

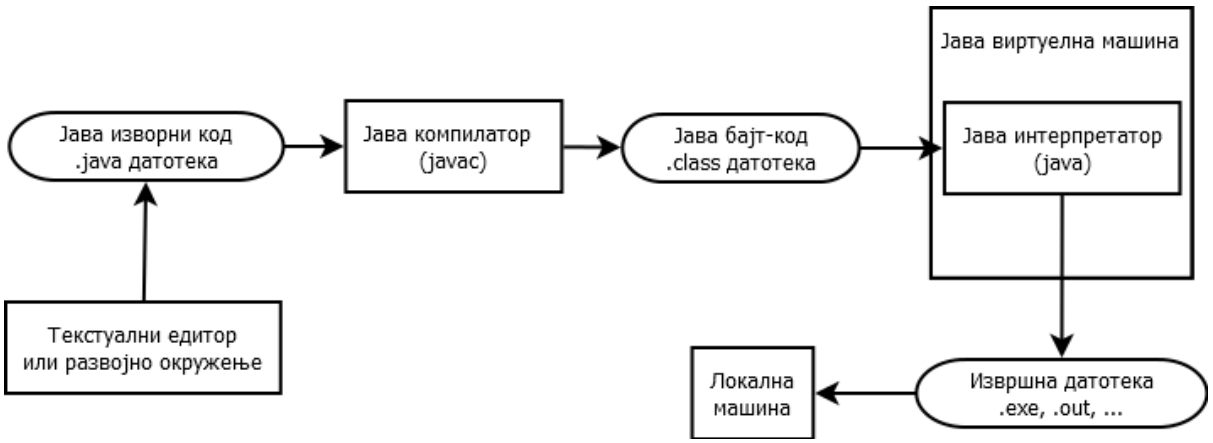
Јава изворни кôд је текстуална датотека која саржи Јава наредбе, тј. текст Јава програма. Она може бити креирана било којим текст-едитором или Јава развојним окружењем. Креирани бајт-код је бинаран и архитектонски неутралан (платформски неутралан). Дакле, потпуно исти бајт-код се извршава на свим платформама које подржавају Јаву.

Коришћењем Јаве се постиже да јединствени изворни Јава кôд на потпуно исти начин ради на различитим платформама — „Напиши једном, извршавај свугде“ (енг. “Write once, run everywhere”). За разлику од језика који имају прави преводилац (попут C, C++, Fortran и других), код Јава превођења не постоји засебан преводилац за сваку циљну платформу, већ је преводилац јединствен. Међутим, он не доводи превођење до завршног (машинског) кода, већ до бајт-кода, који се касније посредством Јава интерпретатора мора довести до машинског у фази извршавања. То значи да свака платформа, уместо преводиоца, сада мора имати платформски-завистан интерпретатор, који је саставни део Јава виртуелне машине.



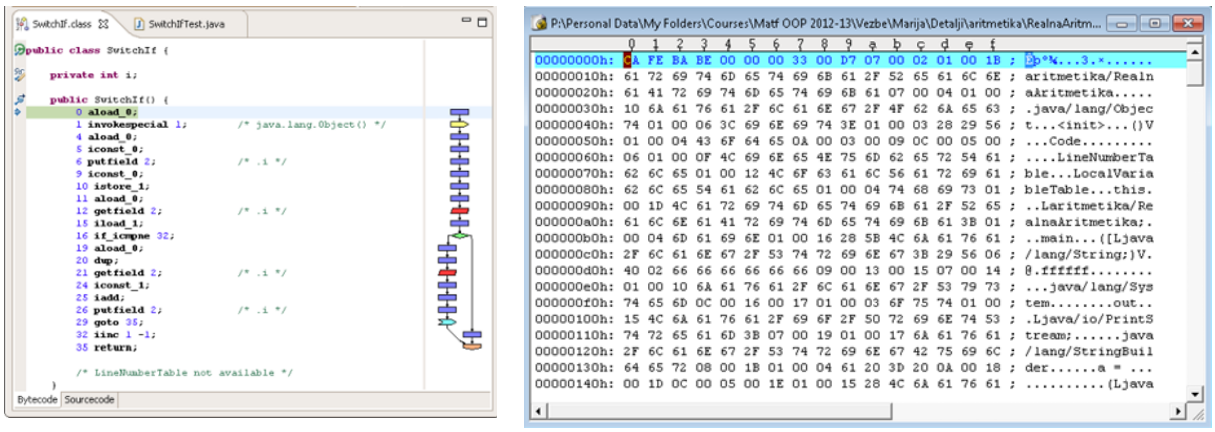
Креирање Јава извршног кода од изворног програма

Прецизније речено, процес извршавања Јава програма одвија се тако што се написани изворни Јава програм прво преведе (коришћењем Јава компилатора **`javac`**) у бајт-код, а потом се преведени бајт-код извршава (уз помоћ Јава интерпретатора **`java`**).



Процес од креирања до извршења Јава програма

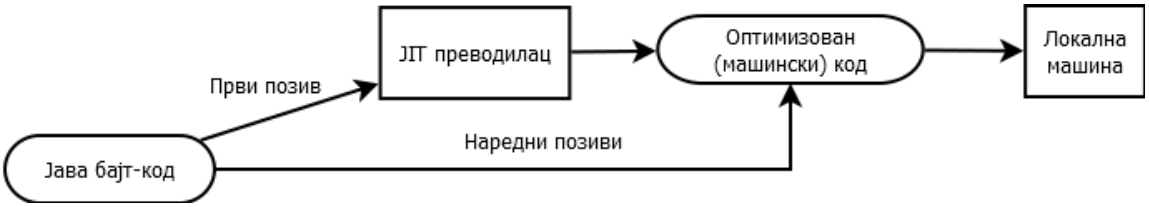
Бајт-код је приказан на следећој слици. Може се приметити да је у питању језик који је по нивоу апстракције ближи асемблерским језицима него вишим програмским језицима — наредбе доста подсећају на конвенционалне асемблерске инструкције. Из тог разлога се може рећи да је Јава „више“ компајлиран него интерпретиран језик – већи део превођења изворног кода до циљног машинског језика спроведе се у фази компилације, тј. пре испоручивања на циљну платформу. Да је обратно, Јава вероватно не би могла да досегне задовољавајући ниво ефикасности. Ово је последица тога што чисто компајлирани језици (са правим преводиоцима) обично имају боље перформансе од чисто интерпретираних језика. Очигледан разлог је то што се код чисто компајлираног језика машински кôд припремио унапред и приликом покретања је потребно само да га процесор изврши, док је код интерпретираног процес превођења и извршавања испреплетен. Јава, као хибрид ове две чисте стратегије превођења “више нагиње” ка компајлираним језицима.



Приказ бајт-кода у бинарном и у дисасемблираном облику

JIT Јава преводилац

И поред тога што је ближа компајлираним језицима, Јавина преносивост ипак изазива одређени губитак перформанси. Разлог томе је што се у процесу интерпретирања бајт-кода превођење у машинске инструкције за конкретну платформу врши више пута. Минимална јединица превођења је метод, односно његов придружени бајт-код. Ако се неки метод позива више пута, што је нарочито присутно у случају рекурзије, сваки позив захтева поновну интерпретацију до машинског кода. Губитак перформанси је смањен коришћењем Just-in-time (срп. «у право време») преводиоца или само JIT преводиоца. JIT је попут других компилатора који производе машински језик на излазу, на пример, GCC за језик C. Међутим, за разлику од њих, превођење се одвија тек по потреби, а не унапред. Друга разлика је у томе што је овде на улазу бајт-код, а не полазни изворни кôд. Приликом превођења бајт-кода до машинског језика за конкретну платформу, JIT преводилац може да врши разне компилаторске оптимизације над методима. Треба имати у виду да претерана оптимизација није увек корисна, на пример спроводити превише оптимизације над неким методом, при чему се он релативно кратко извршава на процесору, може чак да доведе до погоршања перформанси. Дакле, ниво оптимизације треба да буде усаглашен са „значајем“ метода, што се динамички прилагођава кроз рад JIT преводиоца. Када се неки метод преведе помоћу JIT, његов машински кôд се се запамти, тј. упише у својеврсни кеш. Наредни пут, када се поново захтева превођење истог метода, проверава се да ли је његов машински кôд већ у кешу, и ако јесте, само се из њега ишчитава — није потребно интерпретирање. У контексту употребе JIT преводиоца јавља се и феномен познат под називом „спори почетак“ (енг. cold start). Наиме, приликом покретања програма, не постоје методи који су прошли кроз JIT превођење па је потребно превести их или интерпретирати, у зависности од њиховог значаја. JIT превођење додатно поскупљује овај процес па је могуће да ће сам почетак извршавања бити чак скупљи него код класично интерпретираног Јава програма. Након неког времена кеш се напуни преводима метода па и перформансе почињу да се побољшавају. Показује се да употребом JIT преводиоца перформансе конвергирају ка перформансама чистог компајлираног језика (у просеку). Та конвергенција ће бити зависна од степена погодака кеша, а степен погодака кеша од саме семантике програма па допринос JIT перформансама није увек гарантован.



Процес JIT Јава превођења

4.4.1. Јава виртуелна машина

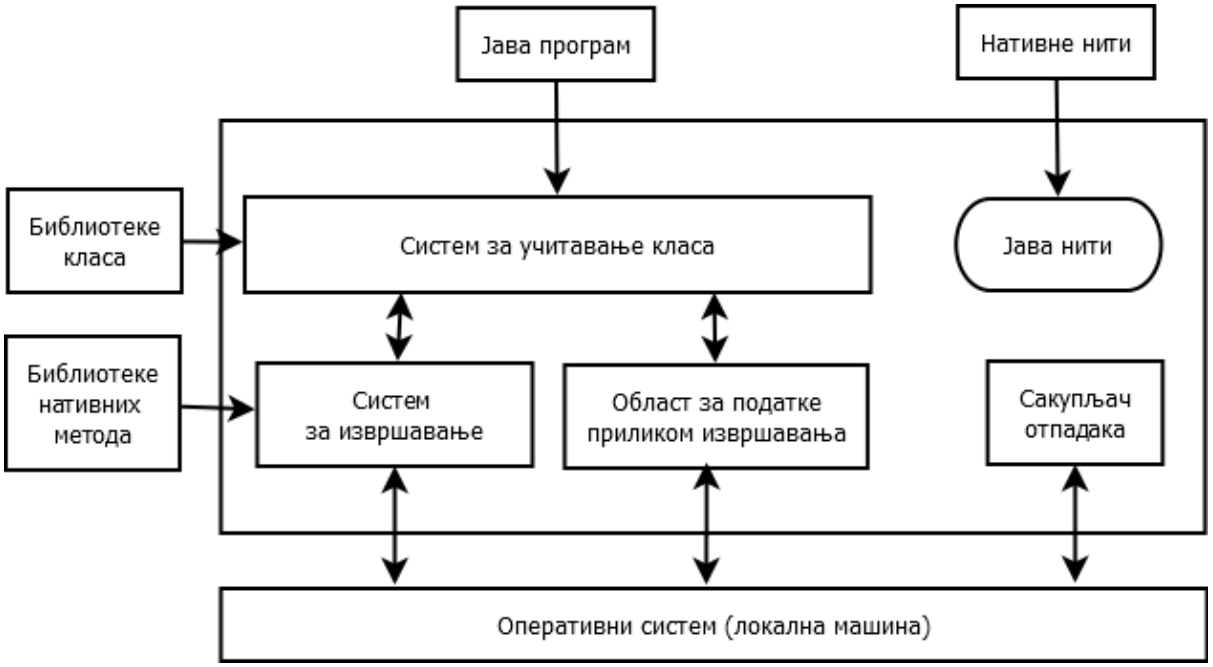
Језгро Јаве је Јава виртуелна машина (eng. Java Virtual Machine — JVM) — виртуелни рачунар који постоји само у меморији. JVM допушта да Јава програми буду извршавани на разноврсним платформама (портабилност). Да би Јава програми могли да раде на одређеној платформи, JVM мора да буде имплементирана на тој платформи.
JVM је врло мала када се имплементира у RAM-у:

- Таква мала величина JVM омогућава да се Јава користи у разноврсним електронским уређајима.
- Цео језик Јава је оригинално развијан тако да се на уму има и кућна електроника.
- Увођењем платформе система модула, почев од Јава 9, величина окружења може додатно да се смањи, тако да буду обухваћени само они модули који ће стварно бити коришћени.

JVM чита ток бајт-кодова из .class датотеке као секвенцу машинских инструкција.
Извршавање инструкција бајт-кода унутар JVM опонаша извршавање машинских инструкција. Код процесора су на улазу машинске инструкције, а на излазу је микрокод (контролни сигнали) који даље контролише унутрашње активности аритметичко логичке јединице, унутрашњих магистрала процесора, као и позиве према системској магистрали. Код JVM система за извршавање, улаз чини бајт-код, док је излаз машински кȏд за конкретну платформу. Кључна разлика је у томе што је JVM чисто софтверски имплементирана, док се машинске инструкције извршавају на хардверу (процесору).

Архитектура JVM

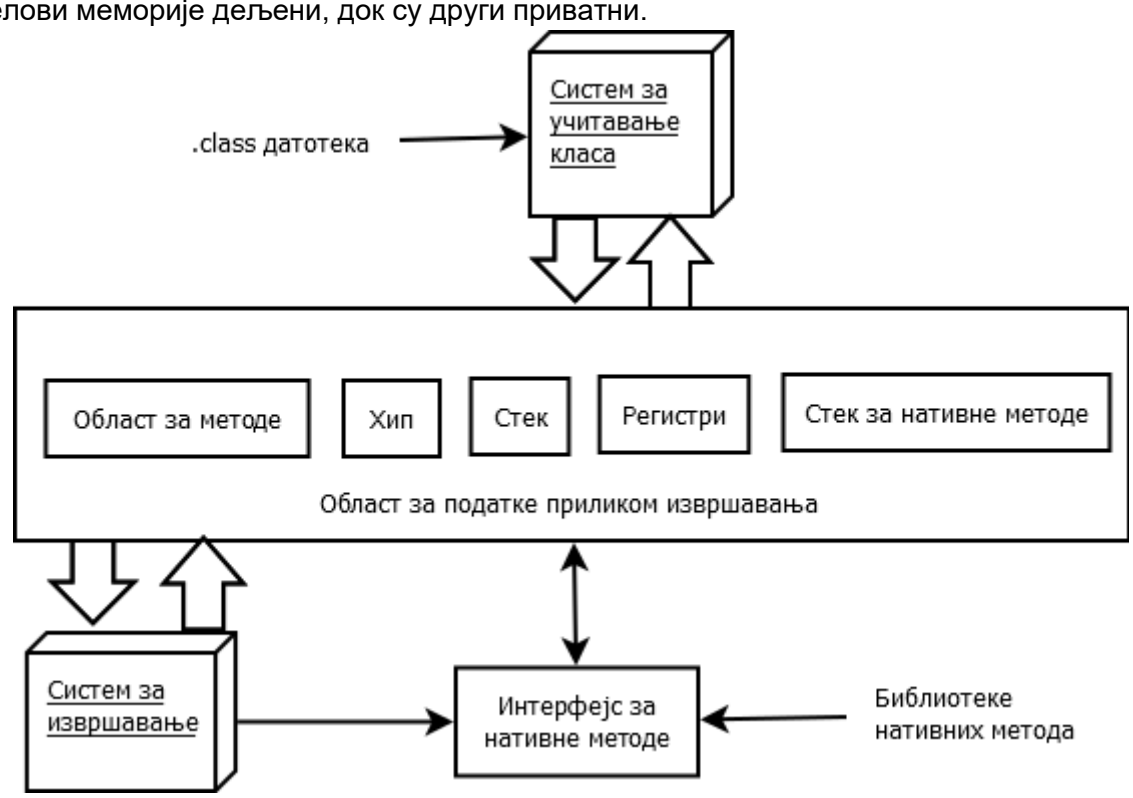
Архитектура JVM осликава архитектуру конкретног рачунарског система. Свака од инструкција JVM је слична асемблерској инструкцији. Она се састоји од једнобајтног операционог кода (опкода, који представља специфичну и препознатљиву наредбу) и од нула, једног или више операнада (података потребних за комплетирање инструкције).
JVM садржи: систем за учитавање класа, систем за извршавање, област за податке приликом извршавања, скупљач отпадака и Јава нити.



Структура Јава виртуелне машине

Меморија JVM

Меморија JVM. тј. област за податке, приликом извршавања, се састоји од: области за Јава методе, хип меморије (eng. heap), стек меморије, регистара и стека за нативне (енг. native) методе. Приликом рада JVM, велики број нити може бити активан па су неки делови меморије дељени, док су други приватни.



Структура меморије Јава виртуелне машине

Област за методе је дељена међу свим нитима JVM. Она, између осталог, садржи бајт-кодове или компајлиране кодове метода. Креира се приликом активирања JVM.

Хип (енг. heap) је такође дељени део меморије из кога се врши инстанцирање и алокација објекта – примерка дате класе. Кад год се алоцира меморија са оператором `new` та меморија долази са хипа.
Стек је додељен на нивоу нити, тј. није дељен међу више њих као што је то случај са облашћу за методе и хипом. Начин рада је исти као и у другим програмским језицима, стек садржи локалне променљиве метода, парцијалне резултате, информације о повратку у претходни позив итд.
Регистри су меморија у којој је записано тренутно стање извршавања различитих нити. Најбитнија информација је програмски број — PC (енг. program counter) који представља редни број инструкције посматране нити (слично као PC унутар процесора). Због дељења времена међу нитима ова информација је потребна како би се знало од које инструкције је потребно наставити извршавање.
Стек за нативне методе је попут конвенционалног C стека и користи се за извршавање нативних метода, односно метода који нису написани у Јави, а Јава програм их користи (позива).

Скупљач отпадака

Објекат који је алоциран на хипу се аутоматски ослобађа уколико на њега више нико не реферише. Ова операција ослобађања зове се скупљање отпадака (енг. garbage collection). Скупљач отпадака (енг. garbage collector) ради као позадинска нит и врши рашчишћавање простора током периода слабије активности процесора.
Будући да је скупљање отпадака аутоматски процес, програмер у Јави не мора експлицитно да брише податке, као што је то случају у програмском језику C (наредба `free()`). Треба имати у виду да експлицитно брисање меморије, попут оног у C-у, омогућава мању потрошњу меморије, под условом да се подаци бришу одмах након што постану непотребни. Јавин процес скупљања отпадака у просеку не брише податке одмах након што постану непотребни, јер би то захтевало да скупљач отпадака буде константно активан, што би нарушавало перформансе JVM. Стога је аутоматско скупљање отпадака компромис између меморијске ефикасности и угодности програмирања, при чему Јава жртвује део меморијске ефикасности зарад повећања угодности програмирања.
Постоји већи број реализација скупљача отпадака, а најчешће коришћени је Oracle HotSpot скупљач, који је заправо „омотач“ око неколико скупљача оптимизованих за различите намене. Заједничка основна идеја свих скупљача отпадака подразумева следеће кораке.

1. Сви нереферисани објекти су идентификовани и означени као спремни за скупљање. Под нереферисаним објектима се мисли на оне према којима не постоји више ниједна активна референца, самим тим њихово постојање на хипу више нема смисла.
2. Означени објекти се бришу.
3. Опциони корак је дефрагментација хип меморије како би преостали објекти чинили континуиран меморијски простор и тиме омогућили бољу искоришћеност при креирању нових објеката.

Сви HotSpot скупљачи отпадака имплементирају тзв. генерацијску стратегију скупљања, која категорише објекте према старости. Мотивација за овај приступ је у чињеници да је већина објекта кратког животног века, тј. објекти у просеку постају спремни за уклањање недуго након свог настанка. Стога је хип подељен на три дела (генерације).

1. Генерацију младих чине новокреирани објекти, а овај део је даље подељен на под-делове у зависности од тога колико су неки објекти преживели циклуса скупљања отпадака. Мало скупљање отпадака подразумева елиминацију оних нереферисаних објеката који припадају генерацији младих, као и пребацивање објеката између под-делова или у наредну генерацију.
2. Генерацију старих чине дуговечни објекти, тј. објекти који су неко време провели у генерацији младих након чега су пребачени у генерацију старих. При већем (захтевнијем) скупљању отпадака проверава се постојање нереферисаних објеката у оквиру генерације старих.
3. Трајну генерацију (PermGen или Metaspace од Јаве 8) чине подаци попут дефиниција класа и метода, тј. подаци који се врло ретко бришу. Ови подаци се налазе у посебном делу хип меморије.

Поред мањег и већег скупљања отпадака (енг. minor и major garbage collection) постоји и потпуно скупљање отпадака (енг. full garbage collection), приликом којег долази до скупљања отпадака у оквиру свих генерација.

Иако се скупљање отпадака имплицитно позива у оквиру JVM, постоји могућност да и програмер позове скупљање експлицитно у оквиру Јава програма. Ово се постиже помоћу Јава метода `System.gc()`, који изазива активирање већег скупљања отпадака.

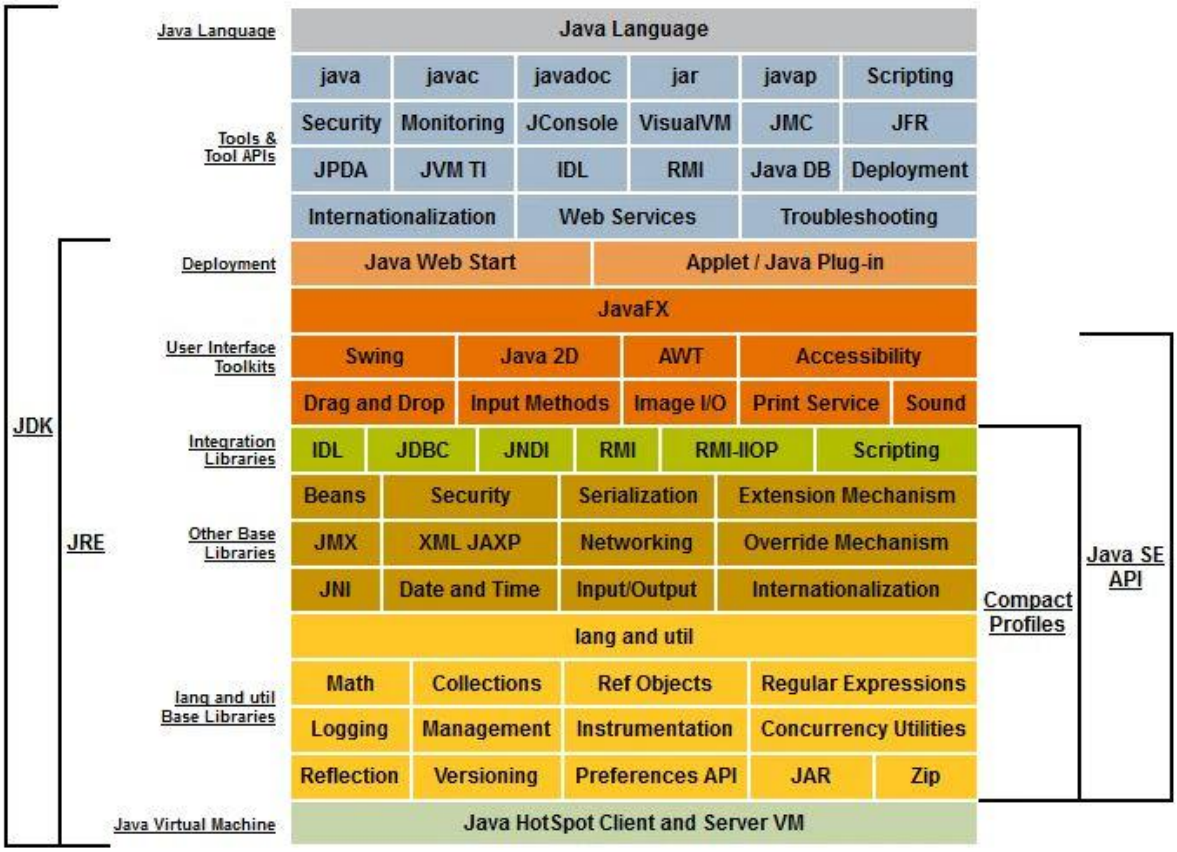
Модерни скупљачи отпадака су веома софистицирани — они имају на располагању информације о употреби меморије и разним статистичким параметрима (о тренутном процесу) што им омогућава да доносе праве одлуке када и који конкретан тип скупљања отпадака треба спровести.

4.5. Јава алати за развој – JDK

Разлог велике популарности језика Јава лежи и у постојању богатог скупа алата и библиотека. У оквиру процеса инсталације, може се изабрати да се уз Јаву инсталирају и алати за Јава развој (енг. Java Development Kit) — JDK.

У случају када је на циљној платформи потребно само извршавати Јава кôд, тј. када се платформан не користи за развој Јава програма, довољно је инсталирати само Јава окружење за извршавање (енг. Java Runtime Environment) — JRE.

Јасно је да је JDK надскуп JRE.



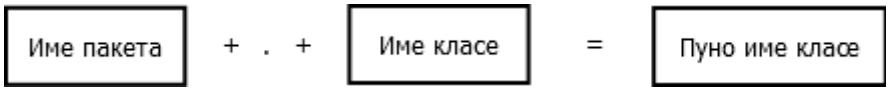
Јава технологије у стандардној едицији (SE)
[Слика преузета са https://www.oracle.com/java/technologies/platform-glance.html]

4.5.1. Библиотеке класа, пакети и модули

Јава платформа омогућава да бајт-код, који ће бити вишестуко коришћен, буде спакован у тзв. библиотеку класа.

Јава API

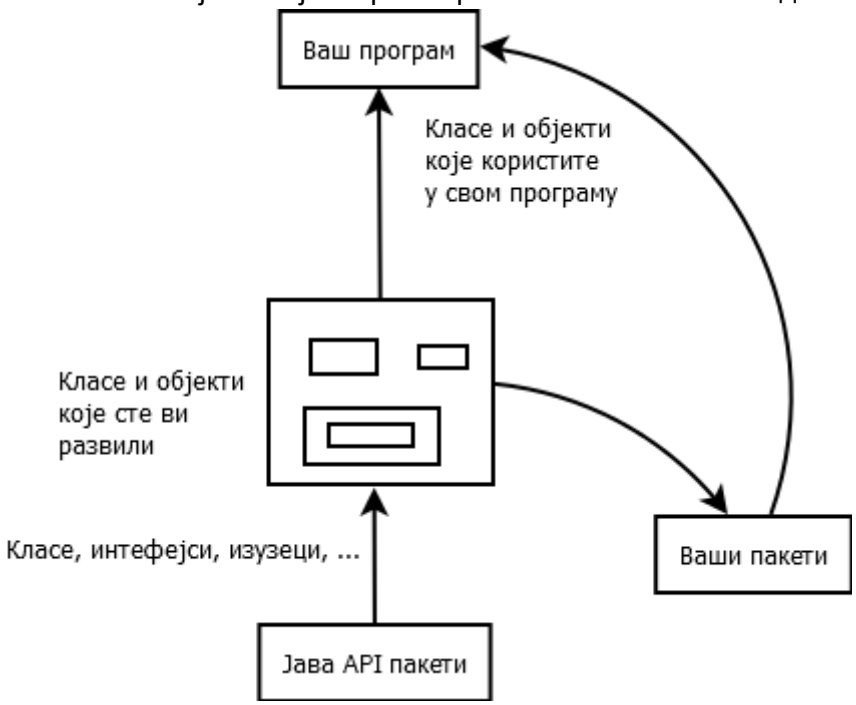
Јава интерфејс за програмирање апликација (енг. Java Application Programming Interface) тј. **Јава API**, је скуп класа које је развио Sun (а надоградио Oracle), за коришћење у језику Јава.
Класе унутар Јава API су груписане у пакете, при чему сваки пакет може садржати више класа и интерфејса. Надаље, свака од класа може имати више поља (енг. fields) и/или метода. Пакети представљају начин за организовање класа. Пакет може садржати класе и друге пакете, на сличан начин као што директоријум садржи датотеке и друге директоријуме. Пун назив Јава класе обухвата називе пакета у којима се та класа садржи. Формира се тако што се надовежу називи пакета, који садрже дату класу (раздвојени тачком), па потом следи тачка и потом име Јаве класе.



Пун назив класе која се налази у пакету

Иако је могуће програмирати у Јави без великог знања Јава API, треба нагласити да свака новоразвијена класа зависи од бар једне класе из Јава API. Надаље, при развоју ма ког нетривијалног програма, нарочито ако се ради са нискама, мрежним конекцијама

и графичким корисчничким интерфејсима, постаје веома значајно познавање класа и објеката из Јава API.
Поред библиотека класа испоручених у оквиру Јава API, програмер може користити и библиотеке класа који су развили други произвођачи софтвера, библиотеке које су развиле његове колеге, као и библиотеке које је он сам развио. Формат у којем се чувају датотеке са библиотекама је такође широко примењив и независан од платформе.



Пример програмирања коришћењем класа из разнородних извора

Модули

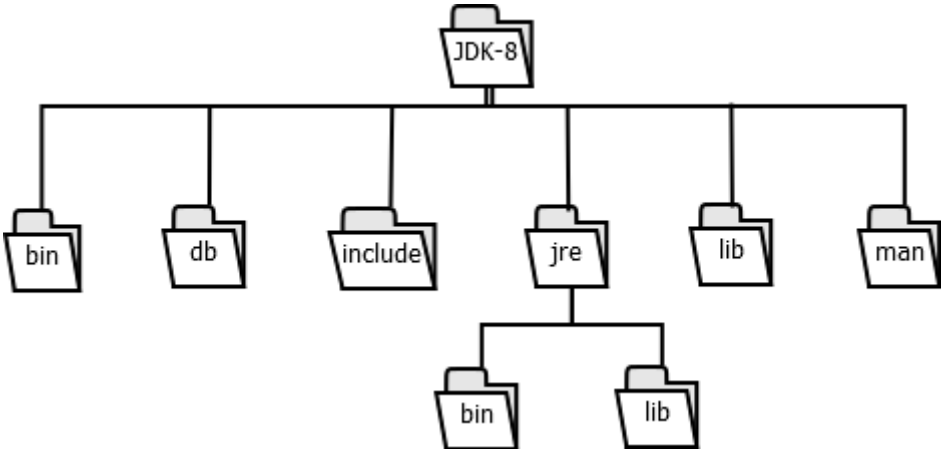
Појам модул обично означава део програма који се преводи независно од осталих делова програма (модула). У програмском језику Јава појам модула означава механизам који омогућава да се Јава апликација (тј. пакети од којих се састоји Јава апликација) организује на елегантнији начин.
У оквиру Јава модула се може дефинисати који од Јава пакета, од којих се састоји аликација, могу бити доступни спољашњости, тј. видљиви за друге Јава модуле. Такође се може подесити који су од спољашњих Јава модула неопходни модулима Јава апликације за њено извршавање.
Иако систем Јава модула уводи додатну сложеност у процес програмирања, предности модула су вишеструке:

1. Величина – када се Јава апликација испоручи преко модуларне платформе, њена величина је мања него у случају када се изврши монолитна испорука. Наиме, у оквиру пројекта Jigsaw, сви API-ји Јава платформе су раздељени у одвојене модуле, па ако развијеној Јава апликацији нису потребни сви API-ји (скоро увек је тако). Могуће је укључити само модуле који садрже потребну функционалност и на тај начин значајно смањити величину испоручене апликације.
   Будући да су током деценија развоја API-ји Јава платформе постали веома велики, онда је при испоруци, пре појаве Јава модула, апликација морала да садржи сав тај програмски кôд, иако огроман део тог кода није коришћен.

2. Учауривање интерних пакета — Јава модул мора експлицитно да дефинише који пакети унутар модула ће бити на располагању осталим модулима који користе дати модул. Јава модул може садржавати и пакете који нису извезени у програмски кôд. Кôд у тим пакетима не може бити коришћен од стране других Јава модула, већ само у Јава модулу у којем су ти пакети дефинисани — то су сакривени пакети или учаурени пакети.
3. Детекција недостајућих модула при покретању — почев од верзије 9, све Јава апликације морају бити организоване као Јава модули, па апликације морају да прецизирају које још модуле користе. Стога Јава виртуелна машина може да провери све зависности апликативног модула при свом покретању. Ако неки од захтеваних модула не постоји, онда ће се пријавити који модул не постоји и одмах ће се прекинути рад.
   (Пре верзије 9, недостајући програмски кôд би био детектован тек када би апликација покушала да га користи, дакле, приликом извршавања.)

4.5.2. Структура JDK директоријума након инсталације

С обзиром да је у Јави 9 уведена платформа система модула, то се структура JDK директоријума за Јава 8 и Јава 9 у великој мери разликују, па је потребно обе описати.

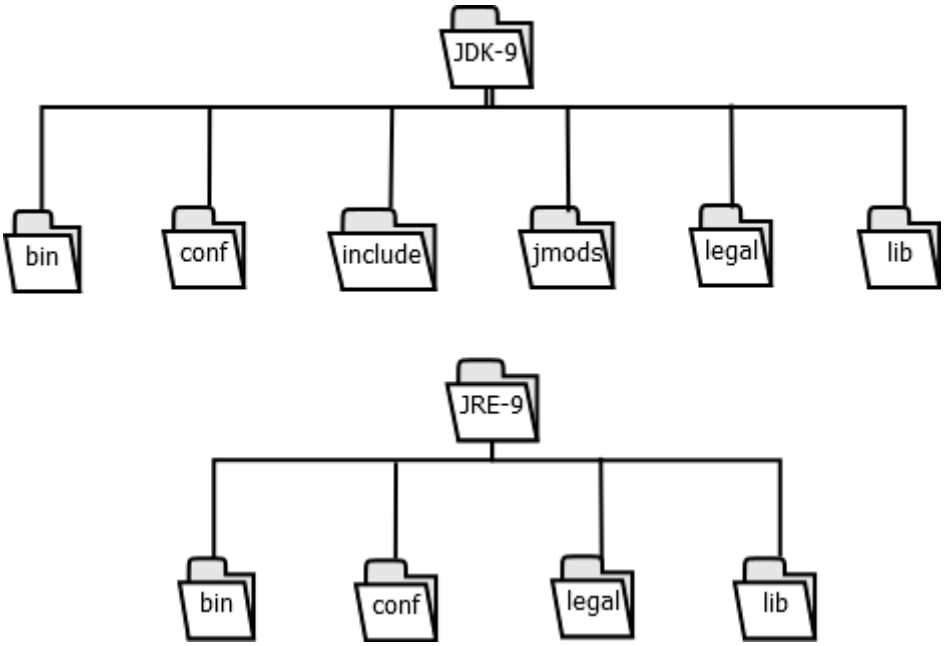


Структура директоријума за Јава 8 JDK

Код Јава 8 JDK инсталације, директоријум `JDK` је корени директоријум. Он садржи датотеке са информацијама о правима копирања и дозволама, као и `README` датотеку, и датотеку `src.zip` у коју је спакован изворни кôд ове Јава платформе.

- Директоријум `JDK/bin` садржи алате који се извршавају из командне линије: `javac`, `javap`, `jconsole`, `jar`, `javadoc` итд. Ти алати се користе за превођење, извршавање, дебагирање итд.
- Директоријум `JDK/db` садржи Java DB базу података.
- Директоријум `JDK/include` садржи заглавља програмског језика C која служе као подршка програмирању помоћу нативног кода у оквиру JNI (енг. Java Native Interface) и JVM дебагер интерфејса.
- Директоријум `JDK/jre` је корени директоријум Јава извршног окружења – JRE.
- Директоријум `JDK/jre/bin` садржи извршне датотеке алата и библиотека које користи JRE.
- Директоријум `JDK/jre/lib` садржи библиотеке, подешавања и ресурсе које користи JRE.

- Директоријум `JDK/lib` садржи датотеке које се користе у развојним алатима, на пример tools.jar који садржи неке класе за подршку у раду алата и услужних функционалности у оквиру JDK.
- Директоријум `JDK/man` садржи упутства за JDK алате.



Структура директоријума за Јава 9 JDK

Код верзије 9 и наредних верзија `JDK` директоријум садржи исте датотеке које и претходно описана верзија 8. Када су у питању директоријуми, ту је нешто другачија ситуација.

- Директоријум `JDK/bin` има исту улогу као и у верзији 8.
- Директоријум `JDK/conf` садржи подешавања (екстензија .properties), полисе (екстензија .policy), и друге конфигурационе датотеке које може да мења програмер или крајњи корисник.
- Директоријум `JDK/lib` садржи детаље имплементације система за извршавање. Датотеке из овог директоријума нису за спољну употребу и не смеју бити мењане.
- Директоријум `JDK/jmods` садржи компајлиране дефиције Јава модула.
- Директоријум `JDK/legal` садржи информације о заштити ауторских права и лиценцне датотеке за сваки модул.
- Директоријум `JDK/include` има исту улогу као и у верзији 8.

Као што се види са претходне слике, `JRE` није унутар `JDK`, већ има свој независни корени директоријум. Он садржи `README` датотеку и наведене директоријуме, који имају исти смисао као и у `JDK` инсталацији, па неће бити поново појашњавани.

Целокупни JDK је подељен у скуп модула. То омогућава програмеру да одабере подскуп који одговара потребама компилације или извршавања за различите типове покретања. У табели испод приказано је само неколико модула и њихових употреба — целокупан списак се може добити помоћу конзолне команде `java --list-modules`.

Модул	**Кратак опис**
`jdk.charsets`	Подршка за карактерске скупове који нису у java.base (најчешће карактери дужине два бајта и IBM карактерски скупови).

`jdk.dynalink`	Дефинише API за динамичко везивање операција високог нивоа над објектима.
`jdk.javadoc`	Дефинише имплементацију алата за генерисање документације и његовог конзолног еквивалента, наредба *javadoc*.
`jdk.jcmd`	Дефинише алате за дијагностику и решавање проблема у JVM, као што су *jcmd*, *jps*, и *jstat* алати.
`jdk.jdi`	Дефинише интерфејс за дебаговање.
`jdk.net`	Дефинише API за мрежно програмирање.
`jdk.security.auth`	Имплементације сигурносних интерфејса и разноврсних модула за аутентификацију.

4.5.3. Издања Јава окружења (стандардно и пословно)

Централни Јава API

Централни (енг. core) API садржи пакете са објектима за које се гарантује да су доступни без обзира на Јава имплементацију. Неки од најпопуларнијих пакета у Централном Јава API-ју су:

- `java.lang` — састоји се од класа које су централне за језик Јава. Он обезбеђује, не само класе-омотаче за просте типове података, као што су `Character` и `Integer`, већ и обраду грешака уз коришћење класа `Throwable` и `Error`. Надаље, класе `System` и `SecurityManager` омогућују програмеру контролу (до извесног нивоа) над Јава системом за извршавање.
- `java.io` — стандардна улазно/излазна Јава библиотека. Овај пакет обезбеђује програмеру могућност креирања и рада са токовима података. Подржан је рад са једноставним типовима, као и са сложеним типовима као што је `StreamTokenizer`.
- `java.util` — садржи већи број корисних класа које нису могле бити уклопљене у друге пакете: класе које омогућавају рад са датумима, класе које омогућавају структурирање података, као што су `Stack` и `Vector`, као и класе које омогућавају парсирање улазног тока података.
- `java.net` — овај пакет чини језик Јава мрежно заснованим језиком. Он обезбеђује способност комуникације са удаљеним чворовима, било преко сокета, било коришћењем URL-ова. На пример, коришћењем овог пакета програмер може креирати своје сопствене Telnet, Chat или FTP клијенте и/или сервере.
- `java.awt` — назив означава скраћеницу за Abstract Window Toolkit. AWT садржи не само конкретне објекте за интерактивне елементе графичког корисничког интерфејса, као што су `Button` и `TextField`, већ и класу за поравнање тих елемената као што је `GridBagLayout`. Такође, садржи класу `Graphic` која обезбеђује богатство графичких могућности, укључујући и могућност цртања разних облика и могућност приказа слика.
- `javax.swing` — Swing је уведен како би се превазишли проблеми на које су наилазили програмери који су користили AWT за креирање графичких апликација – на пример, AWT апликација је могла да изгледа битно другачије на различитим

платформама. Swing се ослања на AWT и садржи класе као што су `JButton` и `JTextField`.

- `java.applet` — овај пакет је најмањи пакет у Јава API, и данас се ретко користи. У њему је дефинисана класа `Applet`, која омогућује рад са Јава аплетима. Класа `Applet` садржи много корисних метода. Информације о аплетовом окружењу се могу добити кроз интерфејс `AppletContext`.

Додатни Јава API (Enterprise, Server, Beans, итд.)

Овде ће бити побројано неколико API-ја који се налазе ван централног, а који се такође често користе: Enterprise API (укључује JDBC, Java IDL и Java RMI), Server API, Security API итд.

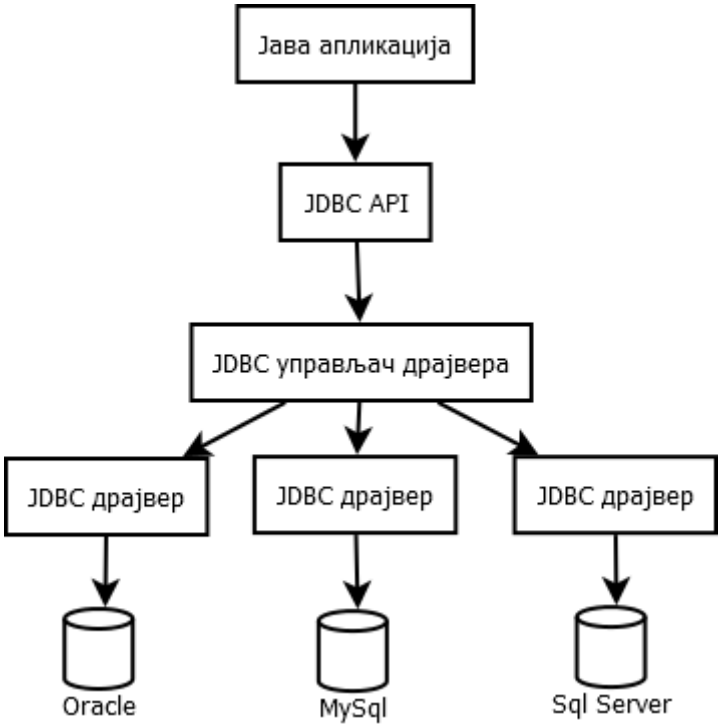
Претходно побројани API-ји су се, закључно са верзијом Јава 8, налазили у оквиру пословног издања Јаве, тј. Java Enterprise Edition (ознака JEE или J2EE), која је била испоручивана и подржавана у оквиру Јава дистрибуције. Почев од верзије Јава 9, J2EE се више не испоручује у оквиру Јаве, већ је под именом Jakarta EE, као софтвер отвореног кода, доступна од стране Eclipse Foundation организације. Позиционирањем ових API-ја као софтвера отвореног кода, омогућено је флексибилно лиценцирање, агилнија измена, еволуција и напредак. Наиме, модел софтвера отвореног кода се показао много успешнијим од модела програмерске заједнице чији рад усмерава софтверска компанија, па је то искуство примењено и на J2EE.

Java Enterprise API

Одржава везу према великим базама података и према наслеђеним апликацијама (eng. legacy applications). Коришћењем овог API-ја развијају се сложене дистрибуиране, клијент/сервер и друге апликације у Јави. Најважнији делови Java Enterprise API су:

- Java Database Connectivity, или JDBC – стандардни интерфејс за приступ SQL базама података, који обезбеђује унифоман приступ за широк опсег релационих база података.[2]
- Јава RMI омогућује позивање удаљених метода између чворова или клијента и сервера у случајевима када се на оба краја позива налазе апликације писане у језику Јава. Библиотека JINI нуди напреднију верзију RMI — она ради слично као и RMI, али уз побољшану сигурност, могућност проналажења удаљених објеката и осталих механизама карактеристичних за рад у дистрибуираним апликацијама.

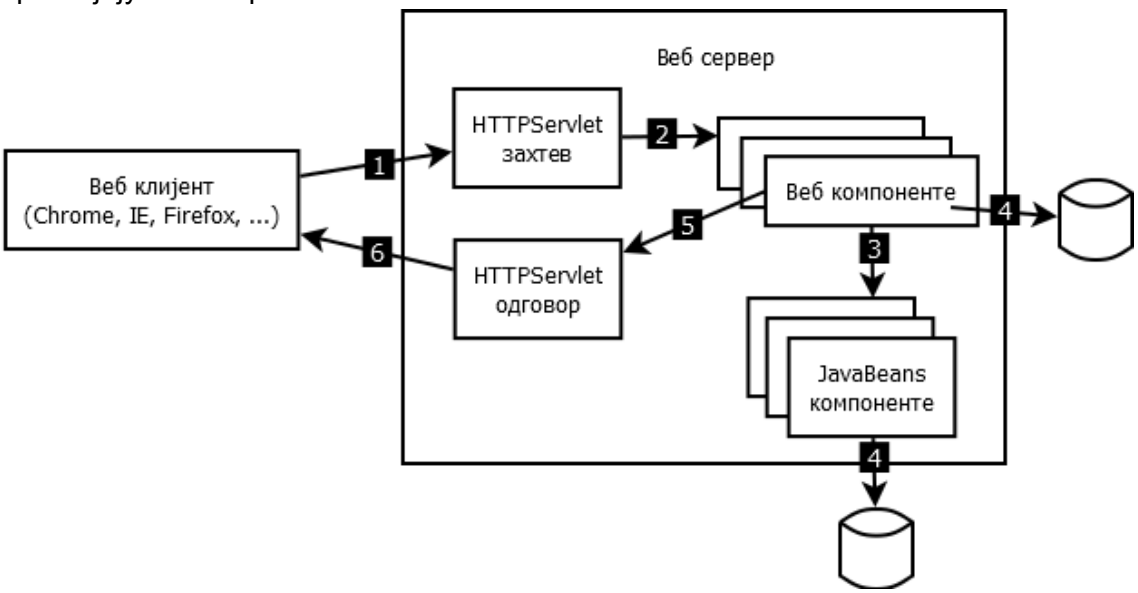
[2] Сада се JDBC налази у Централном Јава API-ју, у пакетима `java.sql` и `javax.sql`.



Приступ бази података преко JDBC

Java Server API

Java Server API је проширив оквир (енг. framework) који омогућује и олакшава развој Јава-заснованих интернет и интранет апликација. Овај API је неопходан програмерима који развијају Јава сервлете[3].



Схематски приказ рада веб сервера

[3] У последње време се чешће препоручује коришћење Spring оквира за развој веб апликација у програмском језику Јава.

Java Security API

Java Security API је оквир за програмере којима је потребно да да лако и поуздано укључе сигурносну функционалност у аплете и апликације. Овај API укључује криптографију са дигиталним потписима, енкрипцију и аутентификацију.

4.7. Резиме

У овом поглављу описана је историја и филозофија развоја језика Јава.
Вишедеценијски развој и унапређивање допринело је великој раширености употребе Јаве у различитим применама: од развоја мобилних апликација до веб сервиса.
Јединствена архитектура Јава извршног окружења, која комбинује употребу компајлера и интерпретатора, омогућила је преносивост Јава програма и њено уједначено извршавање на различитим платформама.
Јава виртуелна машина, због своје компактне имплементације и додатних механизама убрзавања рада програма (на пример, JIT компајлер), омогућила је Јави да буде ефикасна технологија, конкурентна према том аспекту програмским језицима C и C++.

4.8. Питања и задаци

1. У којој верзији Јаве (по њеним творцима) су направљене најзначајније промене? Које су то промене?
2. Шта се подразумева под платформом система модула која је уведена у Java SE 9?
3. Које су захтеве за програмски језик Јава поставили креатори овог језика на почетку развоја?
4. Који типови апликација се могу креирати програмским језиком Јава?
5. Упоредити процес превођења изворног кода који је написан у програмском језику Јава у извршни кȏд и процес превођења изворног кода написаног у програмском језику C у извршни кȏд.
6. Предности и недостаци употребе JIT Јава преводиоца.
7. Објаснити организацију и архитектуру Јава виртуелне машине.
8. Шта је Јава API и за шта се користи?
9. Шта су Јава модули и која је предност употребе система модула?
10. Упоредити структуру JDK директоријума за Јава 8 и Јава 9.
11. Истражити и упоредити различита издања Јава окружења.

5. Језици и опис конструкција језика Јава

Језик је средство за представљање и преношење информација, као и за комуникацију између два или више корисника. Програмски језици чине подскуп скупа свих језика. Корисници програмских језика су, пре свега, човек и рачунар. Међутим, због своје егзактности, програмски језици се користе и за комуникацију међу људима (на пример, за саопштавање научних информација). Програмски језици су вештачки језици намењени запису и преносу специфичних информација.

Природни (говорни) језици су се показали недовољно строгим јер допуштају неједнозначност и непрецизност. Тако нешто није дозвољено у програмским језицима. Дакле, једнозначност сваке конструкције програмског језика је његова најбитнија карактеристика. Како је за човека најудобнији за коришћење његов говорни језик, стално се тежило изградњи програмског језика што ближег говорном језику. Настојећи да направе што удобнији језик за запис програма, људи су направили велики број програмских језика, од којих су многи у употреби и данас. Неки од њих су: FORTRAN, COBOL, LISP, PL/1, Pascal, PROLOG, Ada, Modula 2, C, C++, C#, Јава итд.

5.1. Граматика, синтакса и семантика

Током развоја говорних језика развијале су се и науке о језику. То су: граматика, синтакса, семантика и прагматика. Све ове науке односе се и на програмске језике, а посебно су значајне прве три.[4]

- *Граматика* је наука о језику и његовим законима. Она дефинише скуп правила којима се описују све валидне (исправне) конструкције, прихватљиве у језику.
- *Синтакса* је наука о језику у којој се изучава образовање граматички коректних конструкција језика. Синтаксу језика чини скуп правила за образовање правилних конструкција језика. Уколико је програм написан без поштовања правила о запису конструкција програмског језика, јављају се синтаксне грешке. По правилу, ова врста грешака не изазива велике тешкоће јер може да их открије и програм-преводилац.
- *Семантика* је наука о језику у којој се изучава значење конструкција језика. Семантику језика чини скуп правила за утврђивање значења конструкција језика. За разлику од синтаксних грешака, већину семантичких грешака не може да открије преводилац, већ то мора да уради човек. Овакве грешке се, у принципу, теже откривају.

Сада ће бити дефинисани неки појмови значајни за разумевање даљег текста.

- *Објект-језик* је језик који се изучава.
- *Метајезик* је језик помоћу којег се описује, односно изучава објект-језик.

За формални (строги) опис синтаксе програмских језика најчешће се користе:

- Бекусова нотација и
- синтаксни дијаграми.

Ако се синтакса језика Јава описује помоћу Бекусове нотације, онда она представља метајезик, а Јава је објекат-језик.

Семантика програмских језика се, обично, формално описује помоћу:

- Бечког дефиниционог језика или
- Ван Вајнгаренових граматика.

---

[4] Потпуна спецификација програмског језика Јава 17 се налази на адреси https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf

Формално описивање семантике програмских језика је изузетно сложен посао. Зато се у пракси, по правилу, интуитивно описује семантика програмских језика, коришћењем природног (говорног) језика.

5.1.1. Бекусова нотација

Синтакса програмског језика се описује помоћу коначног скупа металингвистичких формула (МЛФ).
МЛФ се састоји из леве и десне стране раздвојене универзалним мета-симболом ::=
Дакле, МЛФ је облика:

```
α ::= γ
```

где је:

- α металингвистичка променљива;
- γ металингвистички израз.

Металингвистичка променљива је фраза природног језика ограђена стреличастим заградама (<, >).
Металингвистичка константа је знак или кључна реч објект-језика.
Универзални метасимбол ::= чита се „по дефиницији је“.
Универзални метасимбол **|** чита се „или“.
Металингвистички израз може бити:

- металингвистичка променљива,
- металингвистичка константа
- или коначан низ металингвистичких променљивих или металингвистичких константи раздвојених универзалним метасимболом **|** или надовезаних једни на друге.

Да би се јасно разликовали новоуведени метасимболи од знакова објект језика, у наставку ће конструкције мета језика бити подебљане.

Цео декадни број се у програмским језицима записује на исти начин као и у математици.
Примери коректно записаних целих декадних бројева су:

```
3663   -1234   +55252   19  -234   0   833   1_000_234   -1__00__00
```

Подвлака (_) записана у оквиру броја не утиче на његову вредност, на пример, број -1__00__00 је исто што и -10000.
Записи који следе нису коректно записани цели декадни бројеви:

```
00   002   -0   +0   -0_00_00   +1a1   _42
```

Цео декадни број се помоћу Бекусове нотације може дефинисати на следећи начин:

```
<не нула декадна цифра> ::= 1|2|3|4|5|6|7|8|9
<декадна цифра> ::= 0|<не нула декадна цифра>
<знак броја> ::= +|-
<подвлака> ::= _
<декадни цео број без знака> ::= <не нула декадна цифра>
                                |<декадни цео број без знака><декадна цифра>
                                |<декадни цео број без знака><подвлака>
<декадни цео број> ::= 0|<декадни цео број без знака>
                       |<знак броја><декадни цео број без знака>
```

Модификација изворне Бекусове нотације постиже се увођењем нових метасимбола:

1. лева и десна велика заграда (**{**, **}**) — означавају понављање обухваћене конструкције нула, једном или више пута;
2. лева и десна средња заграда (**[**, **]**) — означавају опционо понављање обухваћене конструкције и
3. лева и десна мала заграда (**(**, **)**) — означавају груписање конструкција.

Цео декадни број се помоћу модификоване Бекусове нотације може дефинисати на следећи начин:

```
<не нула декадна цифра> ::= 1|2|3|4|5|6|7|8|9
<декадна цифра> ::= 0|<не нула декадна цифра>
<декадна цифра или подвлака> ::= <декадна цифра>|_
<декадни цео број без знака> ::= <не нула декадна цифра>{<декадна цифра или подвлака>}
<знак броја> ::= +|-
<декадни цео број> ::= [<знак броја>]0|[<знак броја>]<декадни цео број без знака>
```

Цео декадни број се помоћу модификоване Бекусове нотације може дефинисати и на следећи начин:

```
<не нула декадна цифра> ::= 1|2|3|4|5|6|7|8|9
<декадна цифра> ::= 0|<не нула декадна цифра>
<декадни цео број> ::= [(+|-)](0|<не нула декадна цифра>{(<декадна цифра>|_)})
```

5.2. Елементарне конструкције језика Јава

Изворни програм језика Јава је низ Unicode знакова и он се прослеђује преводиоцу. Преводилац анализом програма издваја наредбе, а потом елементарне конструкције (енг. tokens) од којих се креирају сложене конструкције језика Јава. Дакле, у елементарне конструкције се убрајају елементи језика које преводилац издваја као недељиве целине приликом превођења програма.

У елементарне конструкције спадају: идентификатори, кључне речи, литерали, сепаратори, оператори, коментари и белине. Користећи Бекусову нотацију, то се записује на следећи начин:

```
<елементарна конструкција> ::= <идентификатор>|<кључна реч>|<литерал>
                               |<сепаратор>|<оператор>|<коментар>|<белина>
```

5.2.1. Идентификатори

Идентификатор, као што и име казује, служи за идентификовање неке конструкције у Јави. Све конструкције у Јави, као што су: променљиве, класе, методи итд. на јединствен начин се именују преко идентификатора па се идентификатори могу поистоветити са именима. Синтакса идентификатора је следећа:

```
<идентификатор> ::= (<Unicode слово>|$|_){(<Unicode слово>|$|_|<Unicode цифра>)}
```

Из наведеног описа се види да име мора почети словом, знаком за долар или цртом за подвлачење (овде је јасно да метапроменљива `<Unicode слово>` представља било које слово у Unicode-у, а `<Unicode цифра>` било коју цифру). У преосталом делу идентификатора, поред ових знакова, могу да се појаве и цифре.

Следи списак неколико примера идентификатора у Јави:

```
ImeKlase                          _read
X                                 xy123
\u03C0                      $b1
I_Ovo_Je_Identifikator            x1y21T23
jo\u0161                          MojeSve
```

Следеће речи не представљају идентификаторе у Јави:

```
2brata        ►почиње цифром
Novi Sad            ►садржи бланко
lose-definisan      ►садржи црту (знак минус)
x,y.c               ►садржи запету и тачку
a&b                 ►садржи недозвољени знак &
```

Јава је осетљива на величину слова (енг. case-sensitive), тј. у Јави постоји разлика између малих и великих слова, тако да су `Pera1` и `pera1` два различита идентификатора.
Приликом дефинисања идентификатора препоручује се избор прегледних имена:

```
strana, Krug, a1, x11, obimKvadrata, Masa1, godPrihod,...
```

Следећи избор идентификатора доводи до тога да имена могу лако бити помешана међусобно и проузроковати грешке у програму:

```
x1yz, xy1z, x1zy,...
```

5.2.2. Кључне речи

Кључне речи су конструкције који имају специјалну намену у језику Јава и не могу се користити за именовање других ентитета (променљивих, класа и метода).
Кључне речи се могу описати следећом формулом:

```
<кључна реч> ::= abstract|assert|boolean|break|byte|case|catch|char|class|const
                |continue|default|do|double|else|enum|extends|final|finally|float|
                |for|goto|if|implements|import|instanceof|int|interface|long|native|
                |new|package|private|protected|public|return|short|static|strictfp
                |super|switch|synchronized|this|throw|throws|transient|try|void
                |volatile|while
```

Напомене:

- Речи `goto` и `const` су резервисане, али се за сада не користе.
- Кључна реч `strictfp` је уведена са верзијом Јава 1.2, кључна реч `assert` постоји од верзије Јава 1.4, а кључна реч `enum` од верзије Јава 5.

Осим наведених, у Јави постоје литерали: `true`, `false` и `null` који представљају резервисане речи и не могу се користити за именовање других ентитета.

5.2.3. Литерали

Литерали у језику су речи које представљају саме себе, тј. неку вредност. Колоквијално, литерал би се могао назвати запис константе.

Постоје следећи типови литерала: целобројни, реални, логички, знаковни и литерали-ниске. Помоћу Бекусове нотације то записујемо на следећи начин:

```
<литерал> ::= <целобројни литерал>|<реални литерал>|<логички литерал>
              |<знаковни литерал>|<литерал-ниска>
```

Целобројни литерали

Цели бројеви у Јави могу бити записани као декадни, октални или хексадекадни (а од верзије 7 Јаве и као бинарни). Почев од верзије 7, Јава допушта да целобројни литерал садржи и подвлаку.

```
<целобројни литерал> ::= <целобројни декадни>|<целобројни октални>
                         |<целобројни хексадекадни>|<целобројни бинарни>
<целобројни декадни> ::= [<знак броја>]0
                         |[<знак броја>]<не нула декадна цифра>{(<декадна цифра>|_)}
<целобројни октални> ::= [<знак броја>]0{(<октална цифра>|_)}
<октална цифра> ::= 0|1|2|3|4|5|6|7
<целобројни хексадекадни> ::= [<знак броја>](0x|0X){(<хексадекадна цифра>|_)}
<хексадекадна цифра> ::= <декадна цифра>|a|b|c|d|e|f|A|B|C|D|E|F
<целобројни бинарни> ::= [<знак броја>](0b|0B){(0|1|_)}
```

У дефиницији горњих појмова користе са дефинције појмова `<знак броја>`, `<не нула декадна цифра>` и `<декадна цифра>` дате у секцији 5.1.1.

Коректно записани целобројни литерали су:

```
0    ►(0)       125    ►(125)       3567        ►(3567)    0B_11_00  ►(12)
0564 ►(372)     0XABC  ►(2748)      0x23_a4     ►(9124)    01011     ►(521)
56L  ►(56)      043431 ►(2275)      0XE6_53:aL  ►(943418)  0b1011    ►(11)
```

Некоректно су записани следећи литерали (иза стрелице је написан разлог зашто је такав запис некоректан):

```
05693        ► октални број садржи декадну цифру већу од 7
123A4        ► декадни број садржи недекадну цифру
0XaBH2       ► појављује се нехексадекадна цифра у запису броја
```

Реални литерали

Реални литерали су константе које се записују у облику покретне тачке (покретног зареза). У Јави се разликују два типа реалних литерала: `float` и `double`. Разлика између ових типова појављује се само у прецизности записа литерала. Реални литерали могу да се изразе у позиционом запису или експоненцијалном запису.

```
<реални литерал> ::= <мантиса>[<експонент><индикатор>]
<мантиса> ::= <целобројни декадни>
       |<целобројни декадни>.[{(<декадна цифра>|_)}]
       |[<знак броја>].{(<декадна цифра>|_)}
<експонент> ::= (e|E)<целобројни декадни>
<индикатор> ::= f|F|d|D
```

У дефиницији реалних литерала користе са дефиниције за `<целоброни декадни>` из секције 5.2.3, као и за `<знак броја>` и `<декадна цифра>` из секције 5.1.1.

Следеће речи су синтаксно исправни реални литерали:

```
23.57          8.879f             .345d
.569           0.569F             4455.D
```

```
3.14            2e-5f                0.003e+4d
123E-5          1.456575e+12F        3.5E7
```

Следећи записи не представљају реалне литерале (иза стрелице је написан разлог због којег запис на левој страни не представља реалан број):

```
0x0.233        ► не постоје хексадекадни реални литерал
5F             ► није реални литерал
53.4-12        ► недостаје слово E
999E           ► недостаје цео број иза слова E
```

Уколико је литерал типа `float` слово `f` или `F`, мора се навести на крају литерала. Тип `double` је подразумевани тип за реални литерал те се слово `d` или `D` не мора навести.

Логички литерали

Постоје два логичка литерала представљена речима `false` (означава нетачно) и `true` (означава тачно).

```
<логички литерал> ::= true|false
```

Приликом поређења неких величина увек се као вредност добија `true` или `false`. Тако на пример, израз `(2<3)` приликом евалуације даје вредност `true`.

Знаковни литерали

Знаковни литерал (карактер) је било који знак осим апострофа и обрнуте косе црте, записан између апострофа. Празан простор се такође увршатава у знаковне литерале.

```
<знаковни литерал> ::= <графички симбол>|' '|<ескејп-секвенца>
<графички симбол>  ::= '<знак>'
```

Графички симбол је један занак између апосторфа (тј. једноструких наводника ’) Примери записа графичких симбола су:

```
'a'           'b'           'x'           '2'
```

Ескејп-секвенце (енг. escape sequence) су знаковни литерали који почињу обрнутом косом цртом, после чега следи још један или више знакова. У изворном Јава програму могу се користити графички симболи, као и ескејп-секвенце преузете из програмског језика C.

```
'\''     ► апостроф
'\"'     ► наводник
'\\'     ► обрнута коса црта
'\r'     ► знак за повратак на почетак реда (енг. carriage return)
'\n'     ► знак за прелазак у нови ред (енг. new line)
'\f'     ► знак за прелазак на нову страну (енг. form feed)
'\t'     ► знак табулатора
'\b'     ► знак за повратак за једно место уназад (енг. backspace)
```

Поред тога, у Јави се могу користити и ескејп-секвенце којима се описују Unicode знаци, оформљене тако што се између апострофа упише `\u` иза кога следи Unicode кôд тог знака записан у хексадекадном облику:

```
'\u0041'     ► Unicode  знак, тј. слово A
'\u0161'     ► Unicode  знак, тј. слово Š
```

Литерали-ниске

Литерали-ниске се разликују од свих осталих јер нису литерали примитивног типа података. Литерал-ниска се записује као ниска знакова између наводника.

```
<литерал-ниска> ::= "{<знаковни литерал>}"
```

Између наводника може да се појави било који знак осим наводника и обрнуте косе црте — они могу да се појаве само у оквиру ескејп-секвенце.
Примери ниски:

```
""                                      ► празна ниска (не садржи ниједан знак)
"Програмирање i matematika"             ► непразна ниска са различитим писмима
"Ovo je navodnik \", a ovo ne \u3232"   ► ниска са ескејп-секвенцама
```

5.2.4. Сепаратори

У Јави постоји неколико знакова који служе за раздвајање једне врсте елементарних конструкција од других. На пример, у сепараторе спада симбол ; који служи за раздвање наредби у Јави.
Сепаратори су дефинисани на следећи начин:

```
<сепаратор> ::= (|)|{|}|[|];|:|,|.
```

Сепаратори служе само за раздвајање и не одређују операције над подацима.

5.2.5. Оператори и изрази

Оператори омогућавају операције над подацима. Подаци на које се примењују оператори називају се операнди. Према позицији операнада разликују се префиксни, инфиксни и постфиксни оператори. Према броју операнада разикују се унарни, бинарни и тернарни оператори. Најчешће се користи подела на следеће типове оператора: аритметички оператори, релациони оператори, битовни оператори, логички оператори, условни оператор, инстанцни оператор и оператори доделе.

```
<оператор> ::= <аритметички оператор>|<релациони оператор>
               |<битовни оператор>|<логички оператор>|<условни оператор>
               |<инстанцни оператор>|<оператор креирања објекта>|<оператор доделе>
```

Паралелно са изучавањем оператора обично се изучавају и одговарајући изрази. Изрази у Јави се користе да донесу, израчунају и сместе неку вредност. У један израз могу бити укључени: операнди, оператори и сепаратори. Операнд у изразу може да буде: константа, текућа вредност променљиве, резултат позива метода и друго. Приоритет оператора одређује редослед израчунавања.
Примери неких добро формираних израза су дати испод.

```
args.length
funk(x,y)
pera.mika.zika(a,b)
stampaj(a,b,c+funk(b,a))
Math.random()
System.out.print(niz)
```

Ослањајући се на раније дефинисане операторе, можемо дефинисати следеће типове израза:

```
<израз> ::= <аритметички израз>|<релациони израз>
            |<битовни израз>|<логички израз>
            |<условни израз>|<инстанцни израз>
            |<израз доделе>|<израз кастовања>
```

У секцијама које следе ће бити описана структура свих горе побројаних врста израза, осим израза кастовања, који ће бити описани по увођењу типова у секцији 5.3.3.

Аритметички оператори и изрази

Аритметички оператори, заједно са операндима и сепараторима, служе за формирање аритметичких израза. Аритметички изрази служе за израчунавање вредности. Аритметички оператори су:

```
<аритметички оператор> ::= +|-|*|/|%|++|--
```

Оператори `+` и `-` могу бити бинарни и унарни, префиксни и инфиксни. Поред познатих оператора `+` `-` `*` и `/`, оператор `%` се користи за рачунање остатка при дељењу. Унарни оператори `++` и `--` служе за увећање, односно умањење вредности за `1` израза на који се они примењују.

Аритметички израз је дефинисан на следећи начин:

```
<аритметички израз> ::= <терм>|<аритметички израз>(+|-)<терм>
<терм> ::= <фактор>|<терм>(*|/|%)<фактор>
<фактор> ::= <основни аритметички израз>|(-|+)<фактор>
<основни аритметички израз> ::= <вредност локације>[--|++]
                              |<целобројни литерал>|<реални литерал>
                              |<позив инстанцног метода>|<позив статичког метода>
                              |(<аритметички израз>)
<вредност локације> ::=<идентификатор>|<индексна променљива>|<инстанцна променљива>
                      |<референца на поље инстанце>|<референца на статичко поље>
```

Уочава се рекурентна природа горње дефиниције – аритметички израз је дефинисан преко аритметичког израза. Дефиниција за `<идентификатор>` је дата у секцији 5.2.1, а за `<целобројни литерал>` и `<реални литерал>` у секцији 5.2.3. Металингвистичка промељива `<вредност локације>` представља вредност локације или тзв. л-вредност (енг. l-value) и описује све синтаксне конструкције у Јави за елементе који имају сопствену меморијску локацију. У горњим дефиницијама се јављају и појмови који засад нису описани. Позиви метода примерка (инстанцног метода) ће бити описани у секцији 8.4.2, а статичког метода у секцији 8.4.4. Индексне променљиве ће бити описане у секцији 7.2, инстанцне променљиве у секцији 8.1.1, реферисање на поље примерка у секцији 8.3.2, а реферисање на статичка поља у секцији 8.3.3.

У следећем примеру указује се на редослед извршења операција при евалуацији аритметичког израза `7*3-7/2+4`:

```
7*3 - 7/2 + 4     ► 21 - 7/2 + 4    - извршава се множење
21 - 7/2 + 4      ► 21 - 3 + 4      - извршава се целобројно дељење
21 - 3 + 4        ► 18 + 4          - извршава се одузимање
18 + 4            ► 22              - извршава се сабирање
```

Редослед извршења операција приликом евалуације израза прецизно је дефинисан табелом оператора на крају секције 5.2.5.

Битовни оператори и изрази

Оператор по битовима може бити логички или оператор померања. То су следећи оператори:

```
<битовни оператор> ::= &|||~|^|<<|>>|>>>
```

Прва четири међу овим операторима (`&`, `|`, `~` и `^`) су логички битовни, и њима се над одговарајућим битовима операнада реализују логичке операције конјункције, дисјункције, ексклузивне дисјункције и негације.
Последња три побројана оператора (тј. `<<`, `>>` и `>>>`) су оператори померања и њима се реализују операције померања за једно место над бинарним садржајем операнда: померање улево, логичко померање удесно и аритметичко померање удесно.
Битовни изрази увек, као резултат извршавања, производи број. С обзиром на ову чињеницу јасно је да постоји велика сличност између неких битовних и аритметичких израза – разлика је једино у симболима оператора који се користе и у њиховом приоритету и асоцијативности.

```
<битовни израз> ::= <битовни терм>|<битовни израз>(&|||^)<битовни терм>
                  |~<битовни израз>
                  |<битовни израз>(<<|>>|>>>)<аритметички израз>
<битовни терм> ::= <основни битовни израз>|(&|||^)<битовни терм>
<основни битовни израз> ::= <вредност локације>|<целобројни литерал>
                          |<позив инстанцног метода>|<позив класног метода>
                          |(<битовни израз>)
```

Пример примене неких битовних оператора је дат испод.

```
12 | 25      ► 29 - 00001100 | 00011001 = 00011101
12 & 25      ► 8  - 00001100 & 00011001 = 00001000
12 ^ 25      ► 21-  00001100 ^ 00011001 = 00010101
8 >> 2       ► 2 - 00001000 >> 2       = 00000010
```

Релациони оператори и изрази

Релациони оператори се могу још назвати и операторима поређења, јер служе за поређење вредности операнада. То су следећи оператори:

```
<релациони оператор> ::= ==|!=|<|>|>=|<=
```

У Јави се, исто као и у програмском језику C, за испитивање да ли су два операнда једнака користи симбол `==` (двострука једнакост). За испитивање да ли су два операнда различита користи се оператор `!=`. Резултат примене релационих оператора је увек логичког типа (`false` или `true`).
Синтакса релационог израза може бити дефинисана на следећи начин:

```
<релациони израз> ::= <релациони терм>(<|>|<=|>=)<релациони терм>
                    |<једнакосни терм>(!=|==)<једнакосни терм>
<релациони терм> ::= <аритметички израз>|<битовни израз>
<једнакосни терм> ::= <релациони терм>|<инстанцна промељива>|<логички израз>
```

У горњим формулама фигурише и елемент `<инстанцна променљива>`, описана у секцији 8.1.1, која се односи на поређење објеката описано у секцији 8.1.4.
Може се приметити да се у дефиницији користи и `<логички израз>`, који је дефинисан тек у следећој секцији. Разлог је узајмна рекурзија у дефиницији ова два концепта. Наиме, иако је у горњем делу релациони израз дефинисан преко логичког, и логички израз ће бити дефинисан преко релационог израза.

У примеру, који следи, описује се редослед израчунавања при евалуацији израза који садржи аритметичке и релационе операторе.

```
(2*3 - 10/7) != (6 - 7%2)  ► (6 - 10/7) != (6 - 7%2) - множење
(6 - 10/7) != (6 - 7%2)    ► (6 -1)!=(6-7%2)         - дељење
(6 - 1) != (6 - 7%2)       ► 5 != (6-7%2)            - одузимање
5 != (6 - 7%2)             ► 5 != (6-1)              - остатак при дељењу
5 != (6 - 1)               ► 5 != 5                  - одузимање
5 != 5                     ► false                   - провера неједнакости
```

Логички оператори и изрази

Постоје три основна логичка оператора: конјункција, дисјункција и негација:

```
<логички оператор> ::= &&||||!
```

Оператор `!` је унарни и префиксни, док су оператори `&&` и `||` бинарни и инфиксни. Као операнди код логичких оператора могу се појављивати само подаци логичког типа. Будући да је резултат извршавања релационог израза логичка вредност, то значи да и релациони израз може учествовати у дефиницији логичког израза:

```
<логички израз> ::= <основни логички израз>
                  |<логички израз>(&&|||)<основни логички израз>
                  |!<логички израз>
<основни логички израз> ::= <идентификатор>
                          |<логички литерал>
                          |(<логички израз>)
                          |<релациони израз>
```

Израчунавање логичког израза `(2<3)&&(3!=4)  || false` се може реализовати на следећи начин:

```
(2 < 3) && (3 != 4) || false  ► true && (3!=4) || false        - прво поређење
true && (3!=4) || false       ► true && true||false            - друго поређење
true && true || false         ► true || false                  - конјукција
true || false                 ► true                           - дисјункција
```

Условни оператор и израз

Условни оператор се описује помоћу знака питања и двотачке, тј. `(?  :  )` и он се најчешће користи у форми:

```
<условни израз> ::= <логички израз>?<израз>:<израз>
```

Извршавање условног оператора се реализује тако што се прво одреди вредност за израз лево од упитника, тј. за `<логички израз>`. Ако је вредност тог израза `true`, тада се израчунава вредност израза између упитника и двотачке и тако добијена вредност представља коначан резулат извршавања условног оператора. У супротном се извршава вредност десно од двотачке.

Инстанцни оператор и израз

Помоћу инстанцног оператора `instanceof` проверава се да ли је конкретан објекат инстанца дате класе или датог интерфејса. Инстанцни израз има следећу форму:

```
<инстанцни израз> ::= <вредност локације> instanceof <назив класе интерфејса>
<назив класе интрфејса> ::= <идентификатор>
```

Металингвистичка променљива `<назив класе интерфејса>` означава назив класе, односно интерфејса. С обзиром да још нису обрађени пакети, сматраће се да је назив класе/интерфејса идентификатор. Међутим, назив класе/интерфејса може, али не мора, обухватати имена пакета који садрже дату класу/интерфејс. Организација класа по пакетима је описана у секцији 8.2.

Оператор `instanceof` враћа вредност `true` ако је објекат, чији је назив дат са леве стране оператора `instanceof`, примерак наведене класе (или интерфејса), чије је име дато са десне стране оператора `instanceof`. У супротном враћа вредност `false`.

О овом оператору и његовом коришћењу биће посвећена већа пажња у секцији 8.5.2.

Оператор и израз за креирање објекта

Помоћу оператора `new` креира се објекат примерак дате класе или се алоцира простор за низ. Израз за креирање објекта има следећу форму:

```
<израз креирања> ::= <креирање објекта>|<алокација низа>
<креирање објекта> ::= new <назив класе>([<листа аргумената>])
<назив класе> ::= <идентификатор>
<листа аргумената> ::= <аргумент>{,<аргумент>}
```

Променљива `<назив класе>` означава назив класе чија се инстанца креира. Засад се сматра да је то идентификатор, мада он може, али не мора, обухватати имена пакета који садрже ту класу, што је описано у секцији 8.2. Креирање објеката је детаљније обрађено у секцији 8.1.2. Металингвистичака променљива `<алокација низа>` је већ дефинисана у секцији 7.1.

Оператори доделе и изрази доделе

Оператор доделе, као што име казује, служи да додели вредност некој променљивој. Оператор доделе се најчешће употребљава у форми:

```
<оператор доделе> ::= =|<саставни оператор доделе>
<израз доделе> ::= <вредност локације><оператор доделе><израз>
```

Израз доделе се извршава тако што се евалуира израз десно од оператора доделе, а потом се израчуната вредност постави у локацију одређену вредношћу локације лево од знака једнакости. Израз доделе, као резултат, враћа вредност евалуираног израза десно од знака једнакости.

С обзиром на дефиницију израза, који може бити израз доделе (а израз доделе може бити израз), може се применити рекурзивна тј. уланчана примена оператора доделе. Тако, на пример, наредбом:

```
m = n = k = 5;
```

се постиже да променљива `k` добија вредност `5`, а како је `n=k`, то ће променљива `n` добити вредност `5`, и по истом принципу ће и `m` добити вредност `5`.

Саставни оператори доделе настају комбиновањем неких претходних оператора и простог оператора доделе:

```
<саставни оператор доделе> ::= +=|-=|*=|/=|%=|&=||=|^=|<<=|>>=|>>>=
```

Саставни оператор доделе служи за компактнији запис наредби увећања, умањења и слично — тако се наредба `<идентификатор>=<идентификатор><оператор><израз>` може краће записати у облику `<идентификатор><оператор>=<израз>`.

Следи неколико примера коришћења саставног оператора доделе:

```
P *= a;                ► је краћи запис за:          P = P*a;
d /= x+y*z;            ► је краћи запис за:          d = d/(x+y*z);
```

Саставни оператори доделе могу бити уланчани, као што показује следећи кôд:

```
int a=2, b=3, s=5, s1=1;
s+=s1+=a*b;
```

У овом примеру променљива `s1` добија вредност `7`, а променљива `s` вредност `12`.

Оператори: арност, асоцијативност, приоритет

У једном изразу може да се појави већи број оператора па се намеће питање који је коректан редослед њихове примене? Сваком оператору придружен је приоритет. Шта ако више оператора имају исти приоритет? Тада редослед одређује асоцијативност.
Оператор може бити:

- лево-асоцијативан (аритметички оператори),
- десно-асоцијативан (оператор доделе),
- или неасоцијативан (релациони оператори, на пример, нема смисла израз `a < b < c`).

У табели испод мањи број указује на виши приоритет, на пример, оператор `++` је приоритетнији од оператора `||`.

Приоритет	Оператор	Асоцијативност
1	(), []	неасоцијативан
2	new	неасоцијативан
3	.	лево-асоцијативан
4	++, --	неасоцијативан
5	-(унарни), +(унарни), !, ~, ++, --, (тип)	десно-асоцијативан
6	*, /, %	лево-асоцијативан
7	+, -	лево-асоцијативан
8	<<, >>, >>>	лево-асоцијативан
9	<, >, <=, >=, instanceof	неасоцијативан
10	==, !=	лево-асоцијативан
11	&	лево-асоцијативан
12	^	лево-асоцијативан
13	|	лево-асоцијативан
14	&&	лево-асоцијативан
15	||	лево-асоцијативан
16	?:	десно-асоцијативан
17	=, *=, /=, %=, -=, <<=, >>=, >>>=, &=, ^=, |=	десно-асоцијативан

5.2.6. Белине

Белина је знак који нема графички приказ на излазном уређају. Белине служе за међусобно раздвајање елементарних конструкција и за обликовање програма. Белине

могу бити: размак, хоризонтални табулатор, знак за крај реда, знак за нову страну и знак за крај датотеке.

```
<белина> ::= <размак>|<хоризонталини табулатор>|<знак за крај реда>
             |<знак за нову страну>|<знак за крај датотеке>
```

Белина нема никакав утицај на даљи рад програма. Другиим речима, око сваке металингвистичке променљиве у формули може бити произвољан број белина, које неће променити семанитку (значење) формуле. У металингвистичким формулама које следе то неће бити посебно наглашено, већ ће се постављати један размак између конструкција које морају бити раздвојене белинама.
(Напомена: овде се не мисли на белине које се појављују у оквиру ниски – оне, наравно, утичу на семантику програма, тј. нису неутралне конструкције програмског језика Јава.)

5.2.7. Коментари

Коментари служе да се објасне поједина места у програму. Коментари су, пре свега, намењени човеку, али се у Јави могу искористити и за аутоматско генерисање документације. Конструкције Јаве су често довољно јасне па коментари понекад могу бити и сувишни. Пожељно је на почетку програма објаснити чему програм служи, ко га је писао, када је написан итд.
У Јави постоје 3 врсте коментара: вишелинијски (коментар у стилу језика C), једнолинијски (коментар у стилу језика C++) и документациони.
Једнолинијски коментари се могу писати од почетка реда или у реду где се завршава нека наредба. На пример, има смисла писати:

```
// Секција иницијализације променљивих
s=0; // почетна вредност суме
```

Почетак вишелинијског коментара означен је са секвенцом `/*`, а крај са секвенцом `*/`.
Пример вишелинијског коментара:

```
/* Ово је коментар који
се простире
кроз три реда */
```

Документациони коментар се може користити за аутоматско генерисање документације.
Пример документационог коментара је:

```
/** У овој методи се врши корекција приспелих података.
 Корекција се врши на основу података прочитаних из
 информационог система банке и података прочитаних са Интернета */
```

Коментари се могу описати Бекусовом нотацијом на следећи начин:

```
<коментар> ::= <једнолин. коментар>|<вишелин. коментар>|<документациони коментар>
<једнолин. коментар> ::= //{<знак не крајреда>}<знак за крај реда>
<знак не крај реда> ::= <Unicode знак различит од знака за крај реда>
<вишелин. коментар> ::= /*{{<знак не звезда>}*{<знак не слеш>}}*/
<знак не звезда> ::= <Unicode знак различит од знака ‘*’>
<знак не слеш> ::= <Unicode знак различит од знака ‘/’>
<документациони коментар> ::= /**{{<знак не звезда>}*{<знак не слеш>}}*/
```

5.3. Типови података у Јави

Тип података представља један од основних појмова у строго типизираном програмском језику. Тип у Јави има следеће карактеристике.

1. Тип података одређује скуп вредности које могу бити додељене променљивима или изразима.
2. Над њима се могу извршавати одређене операције, односно функције.
3. Тип променљиве или израза може се одредити на основу изгледа или описа, а да није неопходно извршити неко израчунавање (на пример, не мора се стварно извршити операција дељења за конкретне бројеве да би се знало да је резултат реалан број у рачунарском смислу).

Јава је строго типизиран језик и свака операција или функција реализује се над аргументима фиксираног типа. Тип резултата се одређује према посебним фиксираним правилима.

Увођењем типова података омогућава се да преводилац лакше открије неисправне конструкције у језику, што даље представља један вид семантичке анализе. Типови података доприносе прегледности програма, лакшој контроли операција од стране преводиоца и већој ефикасности преведеног програма.

У језику Јава се прави строга разлика између појединих типова и није дозвољено мешање типова. На пример, целобројни тип не може да се третира као логички, што је дозвољено у програмском језику C.

У језику Јава се нови типови података дефинишу преко већ постојећих. Дакле, унапред морају постојати некакви **примитивни** (прости, предефинисани) типови података, који немају компоненте. Тип који није примитиван се при свом креирању ослања на објекте, па се овакав тип назива **објектни** тип. Стога се објектни типови могу посматрати као “омотачи” око примитивних типова — када се сви ти „омотачи“ уклоне, остају само вредности примитивних типова. Ако би се објектни тип представио као дрво, у корену дрвета би био сам објектни тип који се дефинише, у средишњим чворовима би били други објектни типови које циљни користи као подобјекте (директно или индиректно), док би се у листовима дрвета нашли примитивни типови.

Како се објектима приступа преко посебних променљивих, које се називају и референце, објектни тип се још назива и **референцни** (или **референтни**) тип. Референца је аналогна показивачкој променљивој у C-у, с том разликом што се њена нумеричка вредност (адреса меморијске локације) не може прочитати нити изменити.

Помоћу Бекусове нотације ово се записује на следећи начин:

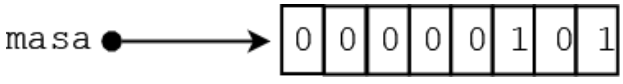
```
<тип> ::= <примитивни тип>|<објектни тип>
```

5.3.1. Примитивни типови података

Ако је нека променљива примитивног типа, она представља локацију у коју ће бити смештена примитивна вредност. Такве променљиве се још називају променљивима **контејнерског** типа. На пример, нека је записано:

```
byte masa = 5;
```

У меморији рачунара ће постојати локација којој је додељено име `masa` и која ће садржати вредност `5` у бинарном облику, као на следећој слици:



У зависности од конкретног примитивног типа, величина меморијске локације може бити различита, али она ће увек садржати вредност примитивног типа.
Сваки примитивни тип карактерише нека резервисана реч. Примитивни тип може бити аритметички или логички. Логички тип се описује резервисаном речју `boolean`:

```
<примитивни тип>::=<аритметички тип>|boolean
```

Аритметички тип може бити целобројни или реални.

```
<аритметички тип> ::= <целобројни тип>|<реални тип>
```

Постоји пет целобројних типова (у целобројни тип убраја се и знаковни):

```
<целобројни тип> ::= byte|short|int|long|char
```

Реални тип може бити једноструке и двоструке тачности, тј:

```
<реални тип> ::= float|double
```

Целобројни тип података

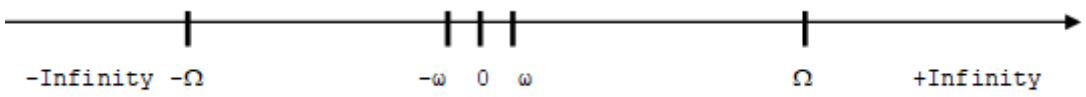
У оквиру целобројних типова података се разликују следећи типови: `byte`, `short`, `int`, `long` и `char`. За сваки од тих типова постоји одређени интервал (одређен величином меморијске речи) из којег се могу узимати вредности. На пример, податак типа `byte` се уписује у меморијску реч дужине један бајт (у потпуном комплементу) па су вредности из интервала $[-2^7, 2^7-1]$. За разлику од програмског језика C, величина примитивних типова је увек фиксирана и не зависи од платформе на којој се извршава бајт-код. Тако су `char` и `short` увек двобајтни, `int` четворобајтни, `long` осмобајтни.
Сви целобројни типови, осим знаковног, могу имати негативне вредности. Цели бројеви су у Јави представљени у формату потпуног комплемента.

Реални тип података

Реални тип у Јави има за циљ представљање скупа реалних бројева у математици — што, наравно, није у потпуности могуће. Вредности реалног типа су елементи одређени реализацијом подскупа реалних бројева на рачунару. Прецизно говорећи, реални тип чини скуп вредности који је подскуп скупа рационалних бројева у математици. Зашто онда употребљавамо термин реални тип? Приликом оперисања са реалним бројевима рационални бројеви и разломци са бесконачно много децимала се редовно замењују (приближно представљају) децималним бројевима са коначно децимала и у пракси су то реални бројеви. У програмским језицима једноставно се представљају децимални бројеви са коначно много децимала и они се третирају као приближни реални бројеви. У Јави постоје два реална типа: `float` и `double`, односно реални бројеви једноструке тачности и реални бројеви двоструке тачности. За представљање бројева једноструке тачности користи се бинарна реч дужине 32 бита, а за бројеве двоструке тачности реч дужине 64 бита. Бројеви се записују према стандарду IEEE 754. Интервали из којих се могу представљати реални бројеви за оба типа приказани су у следећој таблици:

Тип	Интервал
float	1.40239846e-45 до 3.40282347e+38
double	4.94065645841246544e-324 до 1.79769313486231570e+308

Реални тип чине негативни реални бројеви, нула и позитивни реални бројеви. За сваки од претходна два подтипа постоје: најмањи и највећи негативан реални број, нула, најмањи и највећи позитиван реални број. Стога реални тип можемо представити помоћу бројне осе на следећи начин:



Овде су са ω и Ω означени, редом, минимални и максимални реалан број по апсолутној вредности у оквиру одговарајућег реалног типа.
Реални бројеви из области (-∞, - Ω) не могу се регистровати и ако је резултат неке операције из тог интервала, наступило је прекорачење (енг. overflow) — тај резултат третира се као -∞ (-Infinity). Слично, бројеви из области (Ω, +∞) не могу се регистровати и третирају се као +∞ (+Infinity).
Реални бројеви из области (-ω, 0)∪(0, ω) такође не могу бити регистровани. Ако је резултат неке операције из ове области, појављује се поткорачење (енг. underflow), али тај резултат се третира као нула (зато што је реч о веома малим бројевима — блиским нули). Међутим, при оперисању са оваквим бројевима треба бити опрезан јер се могу добити некоректни резултати.
Реални бројеви из области [-Ω, -ω] ∪ {0} ∪ [ω, Ω] могу се регистровати у Јави. У ствари, тачно се могу регистровати само тзв. централни бројеви, а сви остали само приближно. Ако је x централни број, тада се сви реални бројеви (у математичком смислу), из довољно мале околине за x, замењују бројем x.
На реални тип података могу да се примењују:

- релациони и
- аритметички опертори.

Приликом оперисања са реалним бројевима може се као резултат појавити нешто што није број (на пример, ако се нула дели нулом) и стога постоји посебна вредност означена са NaN (енг. Not a Number). Реални бројеви могу бити преведени у целе бројеве, а важи и обрнуто. Треба напоменути да овде постоје посебна правила о превођењу једног типа бројева у други, али се на њима нећемо задржавати.

Знаковни тип података

Знаковни тип је одређен скупом знаковних литерала из Unicode 2.0[5] и операцијама над њима. Сваки знак у Unicode 2.0 је једнозначно одређен целим бројем, тј. својим редним

[5] Кодна страница Unicode 2.0 се може наћи на следећој адреси:
https://www.unicode.org/versions/Unicode2.0.0/

бројем, па се са знаковним типом оперише као са целобројним. Вредности овог типа су ненегативне и чувају се у меморијским локацијама величине два бајта. То значи да вредности овог типа припадају интервалу [0, 65535].

Логички тип података

Логички (`boolean`) тип је окарактерисан:

- скупом логичких константи `true` и `false` које представљају кључне речи;
- скупом логичких оператора и операторима једнакости и неједнакости.

Логички тип је добио назив по имену енглеског математичара Була (George Boole, 1815 – 1864) који се сматра оснивачем математичке логике. Следеће наредбе у Јави: `if`, `while`, `for`, `do-while` и условни оператор `?:` захтевају логичке вредности за навођење услова. Није могућа конверзија (кастовање) из логичког типа у неки други тип, тј. из другог типа у логички.

5.3.2. Објектни тип

Појам објекта је кључан у сваком објектно оријентисаном језику. Објекат у себи обједињује скуп података и поступака за рад са тим подацима. Објектни тип у Јави може бити:

- кориснички — дефинише га сам корисник преко имена класе или имена интерфејса;
- низовни — може се дефинисати тако да компоненте низа буду корисничког објектног типа, или примитивног типа;
- набројиви — дефинише се преко кључне речи `enum`.

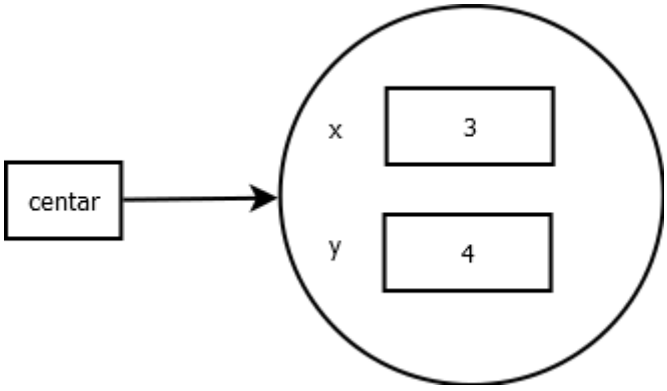
Помоћу Бекусове нотације ово записујемо на следећи начин:

```
<објектни тип> ::= <кориснички објектни тип>|<низовни тип>|<набројиви тип>
<кориснички објектни тип> ::= <тип класе>|<тип интерфејса>
```

Сваки од објектних типова ће бити детаљно изучаван у одговарајућим поглављима овог уџбеника: тип класе у поглављу 8, тип интерфејса у секцији 9.2, низовни тип у поглављу 7 и набројиви тип у поглављу 12. Овде ће у наставку бити дате основне информације о Јава објектима и класама, јер они на неки начин представљају основу за све остале објектне типове.

Објекти и класе

Јава објекти постоје само током извршавања програма. На нивоу меморије променљива која представља објекат (референца) у Јави, уствари, садржи информацију којом се реферише на део меморијског простора који заузима дати објекат.
На пример, инстанцна променљива `centar` представља тачку у дводимензионалном координатном простору са координатама `(3,4)`, што се визуелно може представити на следећи начин.



Представљање променљиве објектног типа

Дакле, вредност која одговара променљивој `centar` је адреса простора у ком су смештене координате тачке.
Дефиницијом класе практично се дефинише нови тип у Јави и сваки инстанца те класе ће имати струкутуру (атрибуте и методе) који су одређени дефиницијом класе.
Дефиниција класе би, коришћењем Бекусове нотације, могла бити дата на следећи начин:

```
<тип класе> ::= class <назив класе><тело класе>
<назив класе> ::= <идентификатор>
<тело класе> ::= <блок>
```

Ова дефиниција класе у није прецизна. На пример, не може сваки блок бити тело класе, већ тело класе има јасно дефинисану структуру. Надаље, она није ни комплетна: засад се разматрају само класе које не користе модификаторе, наслеђивање нити имплементацију интерфејса итд. Како се у наредним поглављима (почев од поглавља 8) буду уводили ови концепти, тако ће дефиниција класе бити додатно прецизирана и комплетирана.

5.3.3. Експлицитна конверзија типа

Конверзија типова у програмском језику Јава је, што се тиче примитивних типова, иста као што је то у програмском језику C – ако се приликом евалуације израза у Јави појави потреба да се ужи тип интерпретира као шири (нпр `int` у `long`, `int` у `double` и слично), то ће бити аутоматски урађено. Дакле, у том случају програмер нема обавезу да означава да ће наступити конверзија.
Међутим, ако програмер уочи потребу да податак ширег типа буде тумачен као податак ужег типа (и на тај начин се можда изгуби прецизност или опсег), тада мора применити експлицитну конверзију типа (кастовање) и креирати тзв. израз кастовања.
Имплицитна и експлицитна конверзија типа постоје не само за примитивни тип, већ и за типове класа и интерфејса, при чему се информација о томе који је тип шири (општији) а који ужи, ослања на односе наслеђивања класа (секција 8.5) и проширења и имплементације интерфејса (секције 9.2.3 и 9.2.2).
Израз кастовања има следећу синтаксу:

```
<израз кастовања> ::= (<тип>)<израз>
```

5.4. Променљиве

Променљива представља локацију у меморији. Свака променљива се карактерише: именом, типом и вредношћу:

1. **име** променљиве је идентификатор,
2. **тип** променљиве је један од претходно уведених типова,
3. променљива садржи вредност ако је примитивног типа или референцу на објекат у супротном.

У оквиру Јава програмског језика, могу се разликовати две групе променљивих:

1. атрибути (поља) и
2. локалне променљиве и аргументи метода.

Атрибути (поља) се према животном веку могу поделити на:

A. инстанцне атрибуте (зову се и атрибути примерка или атрибути објекта) и
B. класне (статичке) атрибуте.

Инстанцни атрибути су везани за животни век конкретног објекта неке класе.
Класни атрибути, са друге стране, немају везе са конкретним објектима, већ са самом класом — објекат уопште не мора бити креиран да би постојао класни атрибут.
Локалне променљиве и аргументи метода се карактеришу сличним понашањем, и њихов животни век одговара животном веку конкретног позива метода у којем се они налазе. Локалне променљиве очигледно могу бити и примитивног и референтног типа.
Још једна карактеристика локалних променљивих је да оне морају имати фиксирану величину, унапред познату још у фази компилације програма. Разлог овоме је чињеница да се позиви метода ослањају на стек. Наиме, да би стек могао да функционише, тј. да би стек оквири за узастопне позиве метода могли да се слажу један на други, величина сваког оквира мора бити унапред позната (не динамичка). Овај услов је очигледно испуњен будући да примитивни типови имају фиксну величину, а када су у питању објекти креирани унутар неког метода, мора се нагласити да се на стеку налазе само референце ка њима, док је њихов садржају на хипу.
У поглављу 8 биће детаљније објашњен начин рада са атрибутима, опсег њиховог важења, њихови меморијски аспекти итд.

5.4.1. Декларација и иницијализација вредности променљиве

Свака променљива мора бити декларисана, при чему се одређује и тип променљиве. Опционо, може се доделити и почетна вредност (иницијализација).
Променљиве примитивних типова се декларишу и по потреби иницијализују слично као што је рађено у програмском језику C, коришћењем резервисаних речи `byte`, `short`, `int`, `long`, `char`, `boolean`, `float`, `double` за тип променљиве.
На сличан начин се декларишу и по потреби иницијализују објектне променљиве, тј. променљиве типа класе. Наиме, сваки објекат мора бити инстанца (примерак) неке дефинисане класе. Име класе које се користи за декларисање променљиве доводи до тога да декларисана променљива добија тип те класе.
На пример, ако се дефинише класа `Osoba` на следећи начин:

```
class Osoba {
       ...
}
```

тада има смисла декларисати променљиве:

```
Osoba pera, mika;
```

Дакле, последњом Јава наредбом декларисане су две променљиве `pera` и `mika` помоћу којих се може приступати конкретним објектима, инстанцама класе `Osoba`.

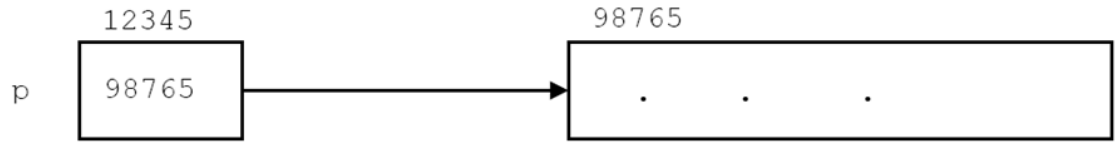
Начин записа података објектног типа (тј. објеката) у меморији се разликује од начина записа података примитивног типа. У оба случаја, подацима у меморији се приступа преко променљивих. Међутим, док код примитивних типова променљиве садрже податке са којима се оперише, код објектног типа променљиве представљају референце (показиваче) на објекте.

За претходно дефинисану класу `Osoba`, можемо једном наредбом декларисати променљиву `p` типа `Osoba`, креирати инстанцу класе `Osoba` и подесити да променљива `p` реферише на ту инстанцу:

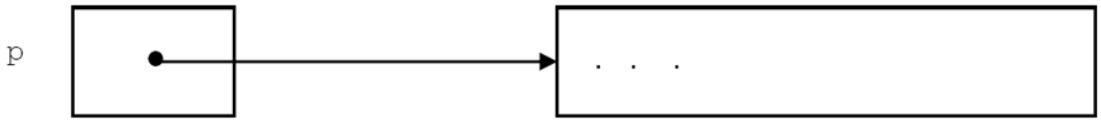
```
Osoba p = new Osoba();
```

Овим је постигнуто да `p` садржи референцу на адресу у меморији од које почиње запис креираног објекта.

Дакле, ако се за променљиву `p` користи локација са адресом `12345`, а запис објекта почиње од адресе `98765`, онда се то графички представља на следећи начин:



Конкретне вредности адресе су, у овом контексту, небитне па је однос променљиве `p` и објекта погодније приказати следећим дијаграмом:



Очигледно, начин записа података објектног типа (тј. објеката) у меморији се битно разликује од начина записа података примитивног типа.

Као и у програмској језику C, могуће је декларисати више променљивих истог типа у оквиру једне вишеструке наредбе за декларацију.

Следеће металингвистичке формуле описују декларацију и иницијализацију променљивих:

```
<декларација и иницијализација променљивих> ::=
               <декларација и иницијализација променљивих примитивног/класног типа>
              |<декларација и иницијализација низовних променљивих>
               |<декларација и иницијализација променљивих набројивог типа>
               |<декларација променљивих типа интерфејса>
<декларација/иницијализација променљивих примитивног/класног типа> ::=
                 <назив пр./кл. типа> <листа декл./иниц. пр./кл. типа>;
<назив пр./кл. типа>::= byte|short|int|long|char|boolean|float|double|<назив класе>
<листа декл./иниц. пр./кл. типа> ::= <декл./иниц. пр./кл. типа>
                {,<декл./иниц. пр./кл. типа>}
<декл./иниц. пр./кл. типа> ::= <назив променљиве>[=<израз>]
<назив променљиве>::= <идентификатор>
```

Дакле, у декларацији (и евентуалној иницијализацији) најпре се наводи тип променљиве. Потом следи један или више идентификатора раздвојених зарезима, који

представљају називе променљивих. Специјално, након назива променљиве може уследити и иницијализација, која је заснована на претходно описаним изразима.

Декларација и иницијализација за низовне променљиве, за променљиве набројивог типа, те декларација променљиве типа интерфејса су описане у секцији 7.1, поглављу 12 и секцији 9.2, респективно.

Примери декларација и иницијализација локалних променљивих су дати испод.

```
int brojGodina;
String mojaNiska;
Knjiga x;
float x, y, tezina;
Sting prva, druga, tvojaNiska;
int i, k, n = 32;
boolean ind = false;
String ime = "Душан";
float a = 3.4f, b = 5.8f, y = 0.2f;
```

Локалне променљиве морају имати постављену вредност пре него што се та вредност искористи (на пример, за читање или у оквиру неког израза) – компајлер проверава да ли је ово задовољено тако што анализира изворни кôд. У програмском језику C ово није захтев па се може десити манипулација променљивом која нема експлицитно постављену вредност, што даље може изазвати велике проблеме у раду програма.

Атрибути не морају иницијализовати вредност пре коришћења пошто за њих постоје подразумеване вредности:

- `null` уколико је реч о објектном типу,
- `0` за нумеричке типове,
- `'\0'` за знаковни тип,
- `false` за логички тип.

5.5. Наредбе

Наредбе у програму служе за опис радњи које треба да се изврше над подацима. Преко извршавања наредби извршава се програм корак по корак и реализује алгоритам описан програмом. Искористићемо Бекусову нотацију да дефинишемо наредбу у језику Јава.

```
<наредба> ::= <декларација и иницијализација променљиве>|<наредба израза>
            |<блок>|<наредба гранања>|<наредба понављања>|<наредба break>
            |<наредба continue>|<обележена наредба>|<празна наредба>
            |<наредба return>|<наредба throw>|<наредба try>|<наредба synchronized>
```

Надаље ће бити описане све наведене наредбе, изузев прве и последње четири. Наредба за декларацију и иницијализацију променљивих описана је у секцији 5.4.1. Наредба `return` се односи на рад са методама и она је описана у секцији 8.4.1. За наредбе `throw` и `try` је потребна посебна пажња и увођење нових концепата, те ће оне бити обрађене у поглављу 11 које се бави изузецима. Наредба `synchronized` је везана за рад у вишенитном окружењу, које није покривено овом књигом.

5.5.1. Наредба израза

Наредба израза се добија тако што се на крај израза, који је дефинисан у секцији 5.2.5, дода знак `;` (тачка-зарез).

Приметити да је овим покривена и наредба доделе, јер је додела један вид израза.

```
<наредба израза> ::= <израз>;
```

5.5.2. Блок

Блок је секвенца од нула, једне или више наредби или декларација локалних променљивих ограђених витичастим заградама:

```
<блок> ::= { <наредбе блока> }
<наредбе блока> ::= {<наредба блока>}
<наредба блока> ::= <наредба>
```

Уочава се да је претходна дефиниција блока рекурзивна:

1. блок је дефинисан преко наредбе,
2. а већ смо видели у секцији 5.5 да наредба може бити блок.

Тела класа, метода итд. су блокови.
Блок може садржати друге блокове и било које друге наредбе које се извршавају једна за другом док се не наиђе на наредбу за промену тока управљања.
У блоку могу бити декларисане локалне променљиве. Оне су видљиве (могу се користити) само од места декларисања до краја блока. Локална променљива не може бити коришћена уколико јој није додељена почетна вредност.
Пример 1. Наредни пример демонстрира шта су блокови у Јави и каква је веза видљивости локалних променљивих и блокова. ■

```
public class TestBlok {
	public static void main(String[] args) {
		System.out.println("Здраво свете");
		{
			int x=5;
			if (x < 3) {
				int y = x++;
				System.out.println(y);
			}
			System.out.println(x);
			// System.out.println(y);
		}
		for(int x=0; x<10; x++) {
			System.out.println(x);
			// int x = 10;
		}
	}
}
```

Тело класе је блок, као и тело сваког метода. У оквиру метода `main()` је креиран блок који, чини се, нема никакву намену. Међутим, он има утицај на видљивост локалне променљиве `x`, која неће бити видљива (дефинисана) ван њега. То омогућава да се у оквиру наредног блока (бројачки блок за наредбу `for`) поново користи променљива са истим називом. Ова два блока су независна (ни један се не садржи у другом) што омогућава вишеструко дефинисање променљиве са истим називом – `x`.
Покушај поновне декларације променљиве `x` у оквиру бројачког блока није могућ па је кôд те декларације коментарисан – у супротном се програм не би компајлирао.
Променљива `y`, која је дефинисана у оквиру блока наредбе `if`, није видљива ван овог блока. Из тог разлога је коментарисан кôд који ван тела наредбе `if` оперише са променљивом `y` – у супротном се програм не би компајлирао.

5.5.3. Наредбе гранања

Наредбе гранања омогућавају доношење одлуке у зависности од испуњења неких услова. Постоје две наредбе гранања у језику Јава: наредба `if` и наредба `switch`.

```
<наредба гранања> ::= <наредба if>|<наредба switch>
```

Наредба if

Скоро сваки програмски језик омогућава неку врсту гранања у програму. За условно извршење наредбе или за избор између извршења две наредбе обично се користи наредба `if`. Синтакса наредбе `if` у Јави има два облика (непотпуни и потпуни):

```
<наредба if> ::= <if-непотпуна>|<if-потпуна>
<if-непотпуна> ::= if(<израз>)<наредба>
<if-потпуна> ::= if(<израз>)<наредба>else<наредба>
```

Непотпуна наредба if

Наредба `if` у непотпуном облику извршава се на следећи начин:

1. Израчунава се вредност израза.
2. Ако је вредност израза истинита, извршава се наредба која следи после израза.
3. Ако је вредност израза неистинита, извршава се прва наредба која следи после наредбе `if`.

У следећем делу програма:

```
...
a = 2;
if (x < 0)
    a = 3;
x = 1;
...
```

ако је `x<0`, вредност променљиве `a` постаје `3`, иначе остаје `2`. У оба случаја `x` добија вредност `1`.

Потпуна наредба if

Наредба `if` у потпуном облику се извршава на следећи начин:

1. Израчунава се вредност израза.
2. Ако је вредност израза истинита, извршава се наредба која следи после израза.
3. У супротном се извршава наредба иза резервисане речи `else`.

Ако је у следећој наредби `a` различито од `0`, израчунава се вредност променљиве `c`, у супротном, штампа се порука о некоректној вредности имениоца.

```
...
if (a != 0)
  c = b/a;
else
  System.out.println("Именилац је некоректан!");
...
```

Уз напомену да испред наредбе `else` мора да стоји `;` (тачка-запета).
У овом примеру се, после израза у `if` наредби и у `else` грани, појављују блокови.

```
...
if (figura.equals("krug")){
  obim = 2*pi*r;
  povrsina = pi*r*r;
}else{
  obim = 4*a;
  povrsina = a*a;
}
...
```

Блок испред речи `else`, у овом случају, не сме да се завршава знаком `;` (тачком-зарезом).
Наредба која веома подсећа на потпуну наредбу `if` је наредба условног израза. Условни оператор и условни израз су уведени раније у оквиру секције 5.2.5. Наредба условног израза омогућава сажет запис потпуне наредбе `if` у програму. Тако уместо:

```
if (x < y)
   min = x;
else
   min = y;
```

може се писати:

```
min = (x < y) ? x : y;
```

У случају када се појављује наредба `if` унутар наредбе `if`, са једном придруженом наредбом `else`, поставља се питање да ли је наредба `else` придружена првој наредби `if` или другој наредби `if`. Другачије записано, ако је дата конструкција:

```
if (U1) if (U2) Nar1 else Nar2
```

да ли ће се `Nar2` извршити ако је:

1. логички израз `U1` тачан, а логички израз `U2` нетачан, или
2. ако је логички израз `U1` нетачан?

Из претходних описа није јасно како ће се извршити. У језику Јава је, према дефиницији, наредба `else` придружена другој наредби `if`. То значи да ће се претходна конструкција реализовати према опису под 1.
Ако је потребна релизација према опису под 2., онда другу наредбу `if` треба сместити у блок, тј. треба написати:

```
if (U1) {
    if (U2) Nar1
}else
     Nar2
```

Наредбе `if` могу бити једна унутар друге (каже се: „угњеждена једна унутар друге“), као у следећем случају:

```
if (U1) if (U2) if (U3) Nar
```

Ако нема гране `else` (као у наведеном случају), уместо наведене гнездасте структуре, погодније је користити једну `if` наредбу:

```
if (U1 && U2 && U3) Nar
```

Пример 2. Написати програм за израчунавање вредности функције:

$$\operatorname{sgn} x = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

при чему је x случајно генерисан цео број из интервала (-10, 10).□

Решење 1. У овом решењу користиће се само непотпуна наредба `if`.

```
public class Sgn1 {
    public static void main(String[] args) {
        int x, sgn = 1;
        x = (int) (-10 + 20 * Math.random());
        if (x < 0)
               sgn = -1;
        if (x == 0)
               sgn = 0;
        System.out.println("x=" + x + "  sgnx=" + sgn);
    }
}
```

У овом решењу се иницијално претпостави да је број `x` позитиван па се `sgn` постави на 1. Потом се непотпуном `if` наредбом проверава да ли је број `x` негативан, ако јесте, `sgn` се постави на -1. Независно од претходне провере врши се и провера да ли је број 0, и у том случају се `sgn` поставља на 0. Овде се уочава сувишност провере `x == 0` у ситуацији када је већ испуњено `x < 0`.

Решење 2. У овом решењу користиће се само потпуна наредба `if`.

```
public class Sgn2 {
    public static void main(String[] args) {
        int x, sgn;
        x = (int) (-10 + 20 * Math.random());
        if (x < 0)
               sgn = -1;
        else if (x == 0)
               sgn = 0;
        else
               sgn = 1;
        System.out.println("x=" + x + "  sgnx=" + sgn);
    }
}
```

У овом случају се програм понаша ефикасније, јер се врше само неопходне, а не и сувишне провере знака броја. Такође се у овом случају не врши иницијализација променљиве `sgn`.
Ако би се у програму изнад изоставила последња наредба `else`, програм се не би компајлирао. Разлог је то што компајлер не дозвољава употребу локалних променљивих којима није подешена вредност, што би се десило уколико би број био позитиван.
Извршавањем било којег од наведених програма добија се резултат попут следећег:

```
x=-9   sgnx=-1
```

или

```
x=4   sgnx=1
```

Пример 3. Написати програм за израчунавање минимума два цела броја. Бројеви се учитавају преко стандардног улаза.☐
Задатак се једноставно решава коришћењем једне потпуне наредбе `if`.

```
public class Min2 {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        int a, b, min;
        System.out.println("Унеси 2 цела броја");
        a = sc.nextInt();
        b = sc.nextInt();
        System.out.println("Унети бројеви су: " + a + " и " + b);
        if (a < b)
               min = a;
        else
               min = b;
        System.out.println("Мин=" + min);
    }
}
```

У претходном решењу није испитивано да ли су бројеви једнаки. Ако су једнаки, минимум је било који од њих. (У овом случају то ће бити вредност променљиве `b`).
За потребе учитавања корисничког уноса искоришћен је објекат класе `Scanner` (пун назив класе је `java.util.Scanner`) и њен метод `nextInt()`, који омогућава учитавање целог броја са стандардног улаза. Детаљнији примери употребе класе `Scanner` ће бити дати у секцији 6.4.
Пример извршавања је дат испод.

```
Унеси 2 цела броја
5
87
Унети бројеви су: 5 и 87
Мин=5
```

Пример 4. Написати програм за уређење три реална броја у неопадајући поредак.☐
Претпоставимо да се реални бројеви из интервала [0,1] генеришу на случајан начин и памте преко променљивих: `a`, `b` и `c`. Извршиће се измена садржаја ових променљивих тако да се у `a` налази број мањи или једнак осталима. То се постиже упоређивањем садржаја променљиве `a` са садржајем променљиве `b`. Ако је у променљивој `b` мања вредност, врши се размена ових двеју променљивих. Затим се понавља поступак за `a` и `c`. Након тога у `a` се налази најмања вредност. Након упоређивања `b` и `c`, у `b` се смешта мања вредност и тиме се обезбеђује да низ вредности `a`, `b` и `c` буде неопадајући.

```
public class Uredjenje {
    public static void main(String[] args) {
        double a, b, c, pom;
        a = Math.random();
        b = Math.random();
        c = Math.random();
        System.out.println("Генерисани су бројеви: ");
        System.out.println("" + a + "   " + b + "   " + c);
        if (b < a) {
               pom = a;
               a = b;
               b = pom;
```

```
        }
        if (c < a) {
               pom = a;
               a = c;
               c = pom;
        }
        if (c < b) {
               pom = b;
               b = c;
               c = pom;
        }
        System.out.println("Након уређења добијамо:");
        System.out.println("a=" + a + "  b=" + b + "  c=" + c);
    }
}
```

Следи пример извршавања.

```
Генерисани су бројеви:
0.045162879062829393  0.9222393727385482  0.5405265729687608
a=0.045162879062829393  b= 0.5405265729687608  c=0.9222393727385482
```

Наредба switch и наредба break

Наредба `switch` служи за избор једне наредбе из скупа од неколико могућих, а на основу вредности неког израза. Већ је показано да се овај избор може извршити и помоћу наредбе `if`, међутим, запис помоћу наредбе `switch` је елегантнији и прегледнији. Формално, синтакса `switch`-наредбе се дефинише на следећи начин:

```
<наредба switch> ::= switch (<идентификатор>){
                          {case <литерал>: {<наредба>}}
                           [default:{<наредба>}]
                     }
```

Идентификатор који следи иза резервисане речи `switch` назива се селектор и мора бити типа:

1. `byte`, `char`, `short` или `int`;
2. енумерисани тип (од верзије 5), описан у поглављу 12;
3. `String`, као и типа неке од класа-омотача простих типова тј.: `Character`, `Byte`, `Short` или `Integer` (од верзије 7), описан у секцији 6.3.

Литерали (вредности) који се појављују иза резервисане речи `case` морају бити истог типа као и селектор. Додатно, све вредности морају бити међусобно различите. Семантика наредбе `switch` детаљније је објашњена следећим примером.

```
switch(S) {
       case c1:
              Nar1;
       case c2:
              Nar2;
              break;
       default:
              Nar3;
}
```

Ток извршавања је следећи:

1. Испитује се вредност логичког услова `c1 == S`. Ако је задовољен, извршава се наредба `Nar1`, након чега се се прелази на корак 2. Ако није задовољен, само се прелази на корак 2, без извршавања наредбе `Nar1`.

2. Испитује се вредност логичког услова `c2 == S`. Ако је задовољен, извршава се наредба `Nar2`, након чега се извршава наредба `break`, која завршава извршавање наредбе `switch` (крај). Ако није задовољен, прелази се на корак 3;
3. Извршава се наредба `Nar3`.

Дакле, може се приметити да се испитивања логичких услова спроводе секвенцијално и да се улази у придружене наредбе ако су услови испуњени (попут `if`, тј. `else if` гране). Логички услови, за разлику од потпуне `if` наредбе, не могу имати произвољну форму, већ то увек мора бити провера на једнакост. Стога је `switch`-наредба мање изражајна од потпуне `if` наредбе, односно све што се може изразити помоћу `switch` може и преко `if`, али не важи обратно. Последњи одељак `switch`-наредбе, означен речју `default`, брине се за све преостале могућности (нешто попут `else` гране). Ефекат наредбе `break` је моментални излазак из целе `switch` наредбе. Наредба `break` се скоро увек користи, јер је захтев да вредности у оквиру `case` наредби морају бити међусобно различите (па су и логички услови који се испитују међусобно искључиви). Поставља се онда питање зашто `break` није подразумеван? Разлог је у томе што се на овај начин може комбиновати више логичких услова (дисјункцијом) са појединачним скупом наредби. То демонстрира наредни пример.

Пример 5. Написати програм који за унету годину и редни број месеца у години исписује број дана у том месецу.□

```
public class BrojDanaUMesecu {
    public static void main(String[] args) {
        int g, m;
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.println("Унеси годину");
        g = sc.nextInt();
        if (g < 1)
               System.out.println("Неисправан унос " + g);
        System.out.println("Унеси редни број месеца [1-12]");
        m = sc.nextInt();
        switch (m) {
        case 2:
               if (g % 400 == 0 || (g % 4 == 0 && g % 100 != 0))
                      System.out.println("Број дана је 28.");
               else
                      System.out.println("Број дана је 29.");
               break;
        case 4:
        case 6:
        case 9:
        case 11:
               System.out.println("Број дана је 30.");
               break;
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
```

```
        case 10:
        case 12:
               System.out.println("Број дана је 31.");
               break;
        default:
               System.out.println("Неисправан унос " + m);
        }
    }
}
```

Будући да на пример, април, јун, септембар и новембар имају исто број дана сви су груписани помоћу узастопних `case` наредби. На крају је извршена наредба која исписује да месец има 30 дана. Дакле, на овај начин се омогућава прављење уније логичких услова. Следе примери два извршавања.

Први пример се односи на месец фебруар код којег постоји додатно разматрање у зависности од тога да ли је година преступна или проста. Година дељива са 4 је преступна, осим ако је дељива са 100, када је проста. Изузетак су године које су дељиве са 400 које су такође преступне.

```
Унеси годину
1996
Унеси редни број месеца [1-12]
2
Број дана је 28.

Унеси годину
2001
Унеси редни број месеца [1-12]
5
Број дана је 31.■
```

5.5.4. Наредбе понављања

Наредбе понављања омогућавају вишеструко извршавање једне или више наредби у току једног извршавања програма. Наредбе чије извршавање се понавља у току једног извршавања програма образују циклус (петљу). Према томе, наредбе понављања служе за опис циклуса.
У Јави постоје четири врсте наредби за циклусе.

```
<наредба понављања> ::= <наредба while>|<наредба do-while>|<наредба for>
                       |<колекцијска наредба for>
```

Прве три врсте наредби понављања су по својој форми исте као у C-у и оне ће бити обрађене у наставку. Четврта врста, којом се описује колекцијски `for` циклус, има смисла само код низова и колекција и о њој ће бити говора у секцијама 7.3 (за низове) и 14.2 (за колекције).

Наредба while

Наредба `while` се још назива наредба циклуса са предусловом. У њој се најпре проверава да ли је испуњен услов и ако јесте извршавају се наредбе циклуса. Синтакса while-наредбе је следећа:

```
<наредба while> ::= while(<логички израз>)<наредба>
```

Ефекат наредбе `while` је следећи:

1. Израчунава се вредност логичког израза.
2. Ако је вредност израза истинита, извршава се наредба и враћа се на корак 1.
3. Ако је вредност израза неистинита, завршава се извршавање наредбе `while` и прелази се на прву следећу наредбу иза наредбе `while`.

Да циклус не би био бесконачан мора постојати могућност промене вредности израза у наредби.

Пример 6. Исписати првих 10 природних бројева применом наредбе `while`.□

```
public class IspisBrojeva {
    public static void main(String[] args) {
        int broj = 1;
        while (broj <= 10) {
               System.out.print(broj + " ");
               broj++;
        }
        System.out.println();
    }
}
```

Следи испис добијен извршењем програма.

```
1 2 3 4 5 6 7 8 9 10■
```

Пример 7. Написати програм за израчунавање:

$$S = \sum_{i=1}^{n} \frac{1}{i^3}$$.□

```
public class Suma {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        double s = 0.0;
        int n, i = 1;
        System.out.println("Унесите природан број");
        n = sc.nextInt();
        if (n < 1)
               System.out.println("Нисте унели природан број!");
        else {
               while (i <= n) {
                      s += 1.0 / (i * i * i);
                      i++;
               }
               System.out.println("Тражени збир је: " + s);
        }
    }
}
```

Следи испис добијен извршењем програма.

```
Унесите природан број
20
Тражени збир је: 1.2008678419584364■
```

Наредба do-while

За разлику од наредбе `while`, услов за излазак из циклуса код наредбе `do-while` налази се на крају, па се ова наредба назива наредба циклуса са постусловом. Синтакса наредбе `do-while` је следећа:

```
<наредба do-while> ::= do <наредба> while(<логички израз>);
```

Извршавање наредбе `do-while` реализује се преко следећих корака:

1. Извршава се наредба иза резервисане речи `do`.
2. Израчунава се вредност логичког израза и ако је вредност израза истинита, враћа се на корак 1.
3. У супротном, прелази се на прву наредбу иза наредбе `do-while`.

Пример 8. Написати програм који генерише и исписује случајне бројеве све док се не добије број већи или једнак 0.9. □

```
public class GenerisiDoUslova {
    public static void main(String[] args) {
        double x;
        do {
               x = Math.random();
               System.out.println(x);
        } while (x < 0.9);
    }
}
```

Последњи број који ће бити исписан је онај који је већи или једнак 0.9 будући да се провера услова изласка из петље спроводи тек након што је већ завршена итерација, односно написан број.
Следи испис добијен извршењем програма.

```
0.530181979996428
0.05975043553978754
0.8043782196285869
0.5156369306746813
0.8900713452569925
0.8852024999788971
0.4307994787189653
0.16907103107183175
0.025891622023825667
0.962839105701089■
```

Све што се описује помоћу наредбе `do-while` може се, такође, описати помоћу наредбе `while`. Наиме, наредбом:

```
do <наредба> while(<логички израз>)
```

постиже се исти ефекат као и наредбама:

```
<naredba>
while (<логички израз>) <наредба>
```

Да ли важи и обрнуто?
Све што се описује помоћу наредбе `while`, може се описати помоћу наредбе `do-while` и наредбе `if`.
Наредбе у телу циклуса код наредбе `do-while` морају да се изврше бар једанпут, за разлику од наредбе `while`, где не морају.

Да ли је неопходно постојање наредбе `do-while`? С обзиром на то да се наредба `do-while` може потпуно заменити наредбом `while`, њено постојање није неопходно. Међутим, у извесним ситуацијама природнија је употреба наредбе `do-while`. На тај начин се добијају лакше-читљиви и прегледнији програми.

Пример 9. Написати програм за израчунавање дужине (броја цифара) унетог целог броја типа `long`.□

Алгоритам је једноставан. Понавља се операција дељења учитаног броја са 10 све док се као резултат не добије 0. При сваком дељењу дужина броја се увећава за 1. Почетна вредност дужине броја је 0.

```
public class DuzinaBroja {
	public static void main(String[] args) {
		java.util.Scanner sc = new java.util.Scanner(System.in);
		long n;
		int duzina = 0;
		System.out.println("Унесите цео број");
		n = sc.nextInt();
		do {
			n /= 10;
			duzina++;
		} while (n != 0);
		System.out.println("Дужина учитаног броја је: " + duzina);
	}
}
```

Следи испис добијен извршењем програма.

```
Унесите цео број
3523221
Дужина учитаног броја је: 7■
```

Пример 10. Написати програм за уношење низа речи (једна реч у једном реду) са стандардног улаза све док се не препозна реч „КРАЈ“. Затим одштампати укупан број речи као и број појављивања речи: „програмирање“, „математика“ „физика“.□

Уводе се четири променљиве: `n` — укупан број речи, `bp` — број појављивања речи „програмирање“, `bm` — број појављивања речи „математика“ и `bf` — број појављивања речи „физика“. Унутар `do-while` петље проверава се појављивање сваке од претходно поменутих речи и ако се нека појављује, увећава се одговарајући бројач.

```
public class PrebrojavanjeReci {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        String s;
        int n = 0, bp = 0, bm = 0, bf = 0;

        System.out.println("Унесите називе школских предмета");
        do {
            s = sc.next();
            n++;
            if (s.equals("програмирање"))
                bp++;
            if (s.equals("математика"))
                bm++;
            if (s.equals("физика"))
```

```
                    bf++;
        } while (!s.equals("КРАЈ"));
        System.out.println("Број учитаних речи " + n);
        System.out.println("Број појава речи 'програмирање'  је: " + bp);
        System.out.println("Број појава речи 'математика'  је: " + bm);
        System.out.println("Број појава речи 'физика'  је: " + bf);
    }
}
```

Следи пример извршавања програма.

```
Унесите називе школских предмета
математика
физика
хемија
биологија
физика
програмирање
programiranje
програмирање
КРАЈ
Број учитаних речи 9
Број појава речи 'програмирање'  је: 2
Број појава речи 'математика'  је: 1
Број појава речи 'физика'  је: 2■
```

Наредба for (бројачки циклус)

Наредба `for` је моћна наредба за опис циклуса. Најчешће се користи за опис циклуса код којих је број понављања наредби унапред познат. У том смислу она представља типичну „бројачку" наредбу. Међутим, наредба `for` се може третирати као боље структурирана наредба у поређењу са наредбом `while`. Значи, може се користити свуда где се користи и наредба `while`. Синтакса наредбе `for` је следећа:

```
<наредба for> ::= for([<иницијализација>];[<логички израз>];[<итерација>])<наредба>
<иницијализација> ::= <листа израза>|<декларација локалне променљиве>
<итерација> ::= <листа израза>
<листа израза> ::= <израз>{,<израз>}
```

Наредба `for` се извршава на следећи начин:

1. Најпре се извршава иницијализациони део петље `for`. Ако нема садржаја у овом делу, не врши се иницијализација. Начелно, инцијализација је израз у којем се подешава вредност управљачке променљиве петље `for`. Иницијализација се извршава само једанпут.
2. Након иницијализације, израчунава се вредност логичког израза који представља услов останка у петљи (ако постоји). Ако израз није дат, његова подразумевана вредност је `true`.
3. Ако израз има вредност `true`, извршава се наредба петље (тело петље), а затим се извршава итерација петље и поново се иде на корак 2.
4. Уколико израз има вредност `false`, окончава се извршавање петље.

У наредби `for` често се појављује променљива помоћу које се врши некакво пребројавање и која се назива бројач. Пребројавање може да се врши уз увећање

вредности бројача (бројање унапред), почев од неке почетне вредности, све док је логички израз тачан или уз умањење вредности бројача (бројање уназад).

Пример 11. Написати програм који у десет редова на екрану исписује текст „Ана воли програмирање“ применом `for` наредбе.☐

```
public class BrojackiIspis {
    public static void main(String[] args) {
         for (int i=1; i<=10; i++)
                 System.out.println("Ана воли програмирање");
    }
}
```

Програм производи тражени испис.■

Пример 12. Написати програм који израчунава факторијел унетог броја применом циклуса `for`.☐

```
public class FaktorijelBroja {
    public static void main(String[] args) {
        long f = 1;
        int n;
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.println("Унесите ненегативан цео број");
        n = sc.nextInt();
        if (n < 0)
               System.out.println("Унети број је негативан.");
        else {
               for (int k = 1; k <= n; k++)
                      f *= k;
               System.out.println("Факторијел је " + f);
        }
    }
}
```

Следи резултат једног могућег извршавања.

```
Унесите ненегативан цео број
12
Факторијел је 479001600■
```

Пример 13. Написати програм за израчунавање суме првих n природних бројева користећи `for` наредбу за бројање уназад. ☐

```
public class SumirajAritmetickiNizUnazad {
    public static void main(String[] args) {
        int s = 0;
        int n;
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.println("Унесите ненегативан цео број");
        n = sc.nextInt();
        if (n < 0)
               System.out.println("Унети број је негативан.");
        else {
               for (int k = n; k > 0; k--)
                      s += k;
               System.out.println("Сума аритметичког низа је " + s);
        }
    }
```

```
}
```

Овај задатак једноставно можемо решити примењујући наредбу `for` за бројање унапред. То важи генерално: свуда где се примењује наредба `for` за бројање унапред, може се применити и наредба `for` за бројање уназад. Важи и обрнуто. Међутим, у неким ситуацијама програм је прегледнији ако се користи бројање унапред, а другим ако се користи бројање уназад. Напоменимо да постављени задатак можемо једноставно решити (без употребе циклуса) користећи формулу s=n(n+1)/2 за израчунвање суме аритметичког низа. Међутим, овде је циљ да се прикажу могућности наредбе `for`, а не да се нађе што једноставније/ефикасније решење.
Следи резултат једног могућег извршавања.

```
Унесите ненегативан цео број
10
Сума аритметичког низа је 55■
```

Наредба `for` је веома општа и не користи се само као бројачка, већ се може користити и за опис циклуса опште намене. У следећим примерима демонстрирају се још неке могућности наредбе `for`.

Пример 14. Реализовати бесконачни `for` циклус. ☐

```
public class BeskonacniFor {
    public static void main(String[] args) {
        long n=0;
        for(;;)
                n++;
        //System.out.println(n);
    }
}
```

Приликом извршавања ништа се неће исписивати, јер се никад неће стићи до наредбе за испис (компајлер указује на овај проблем па је наредба исписа коментарисана). Програм ће трајати све док га експлицитно не прекинемо помоћу одговарајуће команде оперативног система. Што се тиче садржаја променљиве `n`, она ће, с обзиром на ограничени број битова, улазити у вишеструка прекорачења током трајања извршавања. ■

Пример 15. Написати програм којим се израчунава сваки наредни степен променљиве (чија почетна вредност је већа од 0, а мања од 1) све док не постане мањи или једнак 0.1. Овде наредба `for` практично замењује наредбу `while` будући да се изрази за иницијализацију и за итерацију изостављају. ☐

```
public class StepenovanjeBroja {
    public static void main(String[] args) {
        double y = 0.9;
        double s = 1;
        int k = 0;
        for (; s > 0.1;) {
                s *= y;
                k++;
        }
        System.out.println(y+"^" + k + " = " + s);
    }
}
```

Може се приметити да би у овој ситуацији елегантнија била употреба наредбе `while`. Следи пример извршавања.

```
0.9^22 = 0.0984770902183612■
```

Из претходних примера може се уочити да све компоненте наредбе `for` могу бити изостављене, осим сепаратора '`;`'. Бројач у наредби `for` не мора бити целобројна променљива. У следећем делу програма, као бројач, користи се променљива типа `double`.

Пример 16. Написати програм који за аргумент из интервала [0, 1] са кораком 0.1 рачуна и приказује вредност корена броја. ☐

```
public class KorenNaIntervalu {
    public static void main(String[] args) {
        double x, y;
        for (x = 0; x <= 1.0; x += 0.1) {
               y = Math.sqrt(x);
               System.out.println("sqrt(" + x + ") = " + y);
        }
    }
}
```

У испису, који је дат испод, може се приметити да су неке вредности променљиве `x` записане са неочекивано великим бројем значајних цифара у разломљеном делу. Разлог је чињеница да се бројеви `double` типа (као и `float`) записују у бинарној основи. Због тога се при конверзији из декадног система дешава да број који има коначан број значајних цифара у декадном систему, има бесконачан (или превелики за величину регистра) број цифара у бинарном систему. Овде се, дакле, дешава неповратни губитак информације. Стога се након повратне конверзије у декадни запис добија нетачан број. За прецизан рад са разломљеним бројевима може се користити уграђена класа `BigDecimal`.

```
sqrt(0.0) = 0.0
sqrt(0.1) = 0.31622776601683794
sqrt(0.2) = 0.4472135954999579
sqrt(0.30000000000000004) = 0.5477225575051662
sqrt(0.4) = 0.6324555320336759
sqrt(0.5) = 0.7071067811865476
sqrt(0.6) = 0.7745966692414834
sqrt(0.7) = 0.8366600265340756
sqrt(0.7999999999999999) = 0.8944271909999159
sqrt(0.8999999999999999) = 0.9486832980505138
sqrt(0.9999999999999999) = 0.9999999999999999■
```

У оквиру наредбе `for` се може користити више променљивих, што је демонстрирано наредним примером.

Пример 17. Написати програм који исписује индексе елемената на споредној дијагонали квадратне матрице. Димензија матрице је задата природним бројем. ☐

```
public class IspisIndeksaSporedneDijagonale {
    public static void main(String[] args) {
        int i, j;
        int n = 10;
        for (i = 0, j = n - 1; i < n && j >= 0; i++, j--)
               System.out.println("(" + i + ", " + j + ")");
```

```
    }
}
```

Као што се види у коду, извршена је иницијализација оба бројача, а такође и њихова накнадна измена кроз део за итерацију. Може се приметити да је и израз услова останка у циклусу сада нешто компликованији него раније.

Следи испис произведен извршавањем програма.

```
(0, 9)
(1, 8)
(2, 7)
(3, 6)
(4, 5)
(5, 4)
(6, 3)
(7, 2)
(8, 1)
(9, 0)■
```

Наредба break и циклуси

Нешто раније приказана је употреба наредбе `break` у контексту `switch` наредбе. Употреба у контексту циклуса је слична (може се користити уз било који тип циклуса) и ефекат је прекид рада циклуса. Битно је напоменути да се прекид односи само на најближи циклус, односно онај којем `break` наредба директно припада. У случају угњежденог циклуса `break` би се односио на прекид рада унутрашњег, али не и спољашњег циклуса. Синтакса наредбе `break` је следећа:

```
<наредба break> ::= break[ <обележје>];
<обележје> ::= <идентификатор>
```

Као што се може видети, постоји могућност да нареба `break` буде са обележјем, тј. да се иза кључне речи `break` нађе обележје. Таква ситуација ће бити обрађена у секцији 5.5.5.

Пример 18. Помоћу бесконачног `while` циклуса и `break` наредбе, написати програм који сумира све унете бројеве док се не унесе први негативан број. Након уноса негативног броја извршавање програма се прекида. □

```
public class UnosBrojevaBreak {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        int x, suma = 0;
        while (true) {
               x = sc.nextInt();
               if (x < 0)
                      break;
               suma += x;
        }
        System.out.println("Сума бројева је " + suma);
    }
}
```

Следи испис произведен извршавањем.

```
18
242
11
0
234
-12
Сума бројева је 505■
```

Наредба continue и циклуси

Наредба `continue`, за разлику од `break`, не прекида придружени циклус већ наставља са његовом наредном итерацијом, а прескаче сав кôд који се налази до краја актуелне итерације. Синтакса наредбе `continue` је следећа:

```
<наредба continue> ::= continue[ <обележје>];
<обележје> ::= <идентификатор>
```

Као што се може видети, постоји могућност и да нареба `continue` буде са обележјем, што ће бити обрађено у секцији 5.5.5.

Пример 19. Помоћу бесконачног `while` циклуса, `continue` и `break` наредби написати програм који сумира све позитивне бројеве док се не унесе број 0, након чега се извршавање програма прекида. □

```
public class UnosBrojevaContinue {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        int x, suma = 0;
        while (true) {
               x = sc.nextInt();
               if (x == 0)
                      break;
               if (x < 0)
                      continue;
               suma += x;
        }
        System.out.println("Сума позитивних бројева је " + suma);
    }
}
```

Уколико је број `x` негативан, применом наредбе `continue` се моментално прелази у наредну итерацију.
Следи испис произведен извршавањем овог програма.

```
12
14
-12
130
0
Сума позитивних бројева је 156■
```

5.5.5. Обележена наредба

Наредба у програмском језику може имати једно или више обележја (ознака) и онда се назива обележеном наредбом. Синтакса обележене наредбе је следећа:

```
<обележена наредба> ::= {<обележје>:}<наредба>
<обележје> ::= <идентификатор>
```

Обележена наредба има ограничену употребу у језику Јава и користи се у комбинацији са наредбама `break` и `continue` када се жели безусловни пренос управљања на одређено место у програму. На пример, може се написати:

```
vrh: while (true)
     {
        ...
         break vrh;
        ...
     }
     ...
```

или:

```
...
prva: for(i=0;i<n;i++)
       {
          ...
         continue prva;
          ...
       }
...
```

Обележене наредбе највише смисла имају у оквиру угњеждених циклуса, што ће и демонстрирати наредна два примера.

Пример 20. Написати програм који проверава да ли се неки задати текст налази као под-текст (подниска) другог задатог текста. ☐

```
public class TraziPodnisku {
    public static void main(String[] args) {
        String tekst = "Тражимо неку подниску у овој ниски.";
        String podniska = "под";
        boolean pronadjen = false;
        int maks = tekst.length() - podniska.length();
        test: for (int i = 0; i <= maks; i++) {
            int n = podniska.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (tekst.charAt(j++) != podniska.charAt(k++)) {
                    continue test;
                }
            }
            pronadjen = true;
            break test;
        }
        System.out.println(pronadjen ? "Пронађена" : "Није пронађена");
    }
}
```

Спољни `for` циклус је обележен са `test` па се позиви `break test` и `continue test` односе на спољни циклус. Код `break` наредбе додавање обележја није било неопходно пошто она већ директно припада спољном циклусу, али свакако не шкоди. У унутрашњем циклусу проверава се да ли се текст `podstring`, на одређеној позицији испод текста `tekst`, поклапа са истим. Ако се наиђе на прво непоклапање, нема смисла даље проверавати поклапање, па се одмах завршава унутрашњи циклус и започиње наредна итерација спољашњег, односно померање („клизање“) `podstring` текста за једно место десно у односу на `tekst`.

Следи испис произведен извршавањем овог програма.

```
Пронађена ■
```

5.5.6. Празна наредба

Празна наредба је наредба без дејства. Користи се у оним деловима програма где нема никаквих акција. Њена синтакса се може овако изразити:

```
<празна наредба> ::= ;
```

У следећој наредби `for` тело петље је празна наредба:

```
for (x=a; x<81; x+=5)
    ;
```

Празне наредбе нису бесмислен концепт, као што на први поглед делују. Наиме, код оперативних система (у вишенитном извршавању) се могу користити за реализацију концепта активног чекања, тј. када једна или више нити чекају да се испуни неки услов чекања (или обратно наставка рада). Притом на испуњеност тог услова могу да утичу најмање две нити.

Даље, празне наредбе смо већ видели приликом употребе „окрњене“ верзије циклуса `for`, када су неки или сви елементи `for` наредбе могли да буду изостављени (на пример, иницијализација).

Понекад програмер жели намерно да нагласи у коду да се у некој грани не ради ништа како не би било сумње да је програмер заборавио да обради ту грану (могућност). Наравно, ово се може постићи и постављањем коментара без празних наредби, али у ситуацијама када има пуно преосталих (“необрађених) могућности, практичније је користити комбинацију, тј. празне наредбе са додатним коментарима.

```
if(n%2==0)
   n/=2;
else
   ; // непарне бројеве не обрађујемо
```

5.6. Резиме

У овом поглављу је уведена лексика и синтакса програмског језика Јава, са освртом на типове података у Јави и правила њихове употребе. Кроз велики број примера демонстрирана је употреба стандардних програмских конструкција попут: гранања, циклуса, израза, литерала и слично.

Мала разлика у односу на језик C је нешто интензивнија употреба блокова, који сада имају проширену улогу у односу на ону у оквиру језика C (ограђивање тела функције и контролне структуре). Такође, број Јава кључних речи је нешто већи него у C-у.

С обзиром да је ово поглавље доминантно фокусирано на лексику, синтаксу и делимично типове података, не могу се још уочити велике семантичке разлике у односу на програмски језик C. Ове разлике ће постати очигледне у поглављима која следе.

5.7. Питања и задаци

1. Шта је основна разлика између природних (говорних) језика и програмских језика?

2. Шта су граматика, синтакса и семантика језика?
3. Дефинисати реалне бројеве у модификованој Бекусовој нотацији.
4. Да ли су следеће речи идентификатори? Образложити одговор.
   - `new`
   - `a1`
   - `prvi Broj`
   - `8$4`
   - `prvi_Broj`
   - `finally`
   - `$x#`
   - `true`
5. Који од следећих литерала су исправно записани? Образложити одговор.
   - `"abc " def"`
   - `'\x'`
   - `'0'`
   - `23.14-56`
   - `23.14E5`
   - `12`
   - `0X45G`
6. Да ли има разлике у префиксној и постфиксној примени оператора `++` и `--`? Илустровати примерима.
7. Нека су дати:

   ```
   int a = 4, b = 3, c = 0, d = 2;
   boolean uslovA, uslovB;
   uslovA = uslovB = ((a--)>=(++b)) || ((++c)==(3/d));
   ```

   Колике су вредности променљивих `a`, `b`, `c`, `d`, `uslovA` и `uslovB` након извршења наведног дела кода?
8. Нека је дат 8-битни запис целог неозначеног броја 00010101. Извршити померање за једну позицију улево и објаснити како померање утиче на декадну вредност датог броја. Након тога, извршити померање броја за једну позицију удесно и објаснити како померање утиче на декадну вредност датог броја.
9. За шта се користе коментари? Које врсте коментара постоје у програмском језику Јава?
10. Објаснити шта значи да је Јава строго типизиран језик. Истражити који су још програмски језици строго типизирани, као и шта је разлика између строго и слабо типизираних језика?
11. У чему је разлика између примитивних и референтних (објектних) типова у програмском језику Јава? Упоредити референтни тип из програмског језика Јава и показивачки тип из програмског језика C.
12. Размотрити ситуације када се користе различити целобројни типови `byte`, `short`, `int`, `long` и `char`.
13. Када долази до прекорачења, а када до поткорачења при употреби реалних типова?
14. Које објектне типове разликујемо у програмском језику Јава?
15. Како се у меморији записују подаци објектног типа, а како подаци примитивног типа?
16. За шта се користи тачка нотација у програмском језику Јава? Илустровати примером.

17. Објаснити и илустровати примером употребу наслеђених атрибута и метода у потклаcи.
18. Шта су променљиве и које врсте променљивих се разликују у програмском језику Јава?
19. За све наредбе дате у коду примера 1 одредити којој врсти наредби припадају.
20. Примером илустровати ситуације када је потребно користити потпуну `if` наредбу, као и ситуацију када је довољна употреба непотпуне `if` наредбе.
21. Дат је следећи део кода:

```
int a, b, c;
System.out.println("Унеси 2 цела броја");
a = sc.nextInt();
b = sc.nextInt();
if (a%2==0 && b>100)
      c = a/2+b;
else if (a%2 == 0 && b == 100)
      c = b;
else if (a%2==0 && b%2==0)
      c = (a+b)/2;
else
      c = a;
```

Које услове треба да задовољавају променљиве `a` и `b` да би променљива `c` добила вредност једнаку вредности `a`, наредбом у последњем реду датог кода.

23. Дат је следећи део кода:

```
int a, b;
System.out.println("Унеси 1 цели број");
a = sc.nextInt();
switch(a){
      case 100:
      case 1000:
      case 10000:
            b = a/10;
            break;
      case 100000:
            b = a/100;
            break;
      default:
            b = a;
}
```

Коју вредност добија променљива `b` након извршења наведеног дела кода. Записати дати кȍд употребом наредбе `if`.

25. Задатак из примера 9, за израчунавање дужине (броја цифара) унетог целог броја типа `long`, решити употребом `for` петље.
26. Задатак из примера 15, за израчунавање сваког наредног степена променљиве (чија почетна вредност је већа од 0, а мања од 1) све док не постане мањи или једнак 0.1, решити употребом `while` петље.
27. Примерима илустровати ситуацију у којој постоји разлика између употребе `do-while` наредбе понављања и употребе `while` или `for` наредби понављања.
28. Задатак из примера 18, који који сумира све унете бројеве док се не унесе први негативан број, решити без употребе наредбе `break`.
29. Задатак из примера 19, који сумира све позитивне бројеве док се не унесе број 0, након чега се извршавање програма прекида, написати без употребе наредби `break` и `continue`.

30. Објаснити шта су обележене наредбе и примером илустровати ситуацију када је корисно да се користе.
31. Шта је празна наредба и у којим ситуацијама се користи?

6. Коришћење класа и објеката испоручених уз JDK

У овом поглављу ће бити представљене неке од основних могућности Јава библиотеке класа попут исписа и читања са стандардног излаза/улаза, мерења протеклог времена, рада са текстом, псеудослучјаним бројевима, математичким функцијама итд.

6.1. Приступ систему, класа System

Класа `System` се може назвати услужном класом (енг. utility class) будући да се не очекује креирање њених објеката. Акценат је на позивању њених метода у тзв. статичкој нотацији, која је коришћена и у претходним поглављима. Статичка поља и методи ће бити детаљније разматрани у секцијама 8.3.3 и 8.4.4. Засад је довољно рећи да су у питању ентитети који нису зависни од постојања објекта посматране класе, тј. може им се приступати коришћењем имена класе на следећи начин (што доста подсећа на стил процедуралног програмирања):

```
NazivKlase.statickiMetod()
NazivKlase.statickoPolje
```

6.1.1. Приказ текста

Класа `System` има статичко поље `System.out`, које представља објекат везан за стандардни излаз (попут `stdout` у језику C). `System.out` подразумевано показује на конзолу па се запис текста манифестује променом графичког садржаја конзоле.
Овај објекат поседује методе (нестатичке) којима се приступа употребом тачка нотације. Преглед неких често коришћених метода је дат испод.

```
System.out.print(tip arg)
System.out.println(tip arg)
System.out.printf(String format, Object...args)
```

`System.out.print()` исписује аргумент на стандардни излаз (аргумент је произвољног типа, било примитивног или објектног).
`System.out.println()` ради исто што и `System.out.println()` уз то да додаје ознаку за прелазак у нови ред.
`System.out.printf()` исписује текст форматирано, врло слично начину на који то раде функције за форматирани испис у C-у (`printf()`, `sprintf()`, `fprintf()`).
Пример 1. Написати Јава програм који демонстрира употребу исписа различитих типова података у форматираној и неформатираној варијанти на стандардном излазу.□

```
public class RazlicitiIspisi {
    public static void main(String[] args) {
        System.out.print("Пример текста");
        System.out.print(" са конкатенацијом и бројем "+4);
        System.out.println(); // испис празног реда
        System.out.println(67.4);
        double x = 26.43462;
        int y = 43243;
        float z = 1645.14f;
        System.out.printf("x=%9.6f y=%8d z=%.3f", x, y, z);
```

```
        System.out.println();
        String s = "Неки текст";
        char c = 'c';
        System.out.printf("Стринг се умеће помоћу формата %%s овако %s", s);
        System.out.printf(" док се карактер умеће помоћу формата %%%c%s", c,
                          System.lineSeparator());
        System.out.printf("%s\t%s%n", "KPAJ", "ISPISA");
    }
}
```

Када се користи форматирани испис, први број после знака `%` представља број места који ће се користити за комплетан запис броја. Ако нема овог броја, број места се рачуна аутоматски. Број после тачке означава колико места се користи за разломљени део броја. На пример, `%9.6f` значи да ће бити употребљено девет места, од чега ће 6 бити употребљено за запис разломљеног дела реалног броја.

Будући да знак `%` има специјално значење код форматираног исписа, да би се записао знак `%` потребно је записати га два пута у оквиру ниске за запис формата (`%%`).

`System.lineSeparator()` враћа секвенцу за крај реда прилагођену оперативном систему, али се ово може краће записати као `%n` (у оквиру форматираног исписа).

Приметити да је приликом исписа могуће комбиновати различита писма – ово је последица тога што су ниске у Јави конципиране као секвенце Unicode карактера.

Следи резултат извршавања програма.

```
Пример текста са конкатенацијом и бројем 4
67.4
x=26.434620 y=   43243 z=1645.140
Стринг се умеће помоћу формата %s овако Неки текст док се карактер умеће помоћу формата
%c
KPAJ    ISPISA■
```

Док је `System.out` надлежан за испис информација у току нормалног функционисања програма, `System.err` је препоручени начин за испис грешака које се јављају у току извршавања или приликом покретања програма. На пример, ако би се тражило да корисник унесе број, а корисник уместо тога унесе секвенцу карактера „4543-АФД-325-11“, програм би могао да испише на `System.err` упозорење о лошем запису броја и да прекине извршавање или да захтева поновни унос. Списак доступних метода је идентичан као и у случају `System.out`, што је и очекивано с обзиром да су оба статичка поља истог типа и представљају ток података под називом `PrintStream` (више о токовима у секцији 15.1).

Локација исписа `System.out` и `System.err` се може променити применом метода `System.setOut(PrintStream out)` и `System.setErr(PrintStream err)`. На пример, може се креирати датотека која ће се користити за испис, уместо исписа на конзолу.

6.1.2. Мерење протеклог времена

Мерење времена у оквиру програма може бити корисно у анализи перформанси захтевних делова кода. `System` класа омогућава два начина за мерење времена:

1. употребом метода `System.currentTimeMillis()` и
2. употребом метода `System.nanoTime()`.

Принцип рада оба метода је исти — враћају време протекло од неког референтног тренутка у прошлости (Unix epoch - 1. јануар 1970 по Гриничу), с тим што је други метод

прецизнији, јер рачуна протекло време у наносекундама, за разлику од првог који рачуна у милисекундама. На овај начин је могуће индиректно закључити колико траје неки интервал тако што се одузму „апсолутна“ времена догађаја за крај и почетак интервала.
Пример 2. Написати Јава програм који мери време извршавања метода за сабирање бројева од 1 до n. Тестирати програм за различите вредности n, као и за различите методе за мерење времена. Упоредити прецизност добијених мерења изражених у милисекундама.□

```
public class IzmeriVreme {
    public static long sumiraj(int n) {
        long suma = 0;
        for (int i = 0; i < n; i++)
               suma += i;
        return suma;
    }

    public static void main(String args[]) {
        for (int n = 10000000; n <= 1000000000; n *= 10) {
               long pocetak1 = System.nanoTime();
               long suma = sumiraj(n);
               System.out.printf("Сума природних бројева до %d је %d%n", n, suma);
               long kraj1 = System.nanoTime();
               System.out.println("Време у ns употребом nanoTime(): "
                             + (kraj1 - pocetak1));
               System.out.printf("Време у ms употребом nanoTime(): %.5g%s",
                             (kraj1 - pocetak1) / 1000000.0,System.lineSeparator());

               long pocetak2 = System.currentTimeMillis();
               suma = sumiraj(n);
               System.out.printf("Сума природних бројева до %d је %d%n", n, suma);
               long kraj2 = System.currentTimeMillis();
               System.out.println("Време у ms употребом currentTimeMillis(): "
                             + (kraj2 - pocetak2));
              System.out.println("-----------------------------------------------");
        }
    }
}
```

Може се приметити да `nanoTime()` даје много прецизнија мерења и да број протеклих милисекунди метода `currentTimeMillis()` постаје тачнији (или мање погрешан) са растом n, тј. дужином трајања метода `sumiraj()`.
Следи резултат рада програма.

```
Сума природних бројева до 10000000 је 49999995000000
Време у ns употребом nanoTime(): 10856500
Време у ms употребом nanoTime(): 10.857
Сума природних бројева до 10000000 је 49999995000000
Време у ms употребом currentTimeMillis(): 4
-----------------------------------------------
Сума природних бројева до 100000000 је 4999999950000000
Време у ns употребом nanoTime(): 22943200
Време у ms употребом nanoTime(): 22.943
Сума природних бројева до 100000000 је 4999999950000000
Време у ms употребом currentTimeMillis(): 23
-----------------------------------------------
Сума природних бројева до 1000000000 је 499999999500000000
```

```
Време у ns употребом nanoTime(): 228048000
Време у ms употребом nanoTime(): 228.05
Сума природних бројева до 1000000000 је 499999999500000000
Време у ms употребом currentTimeMillis(): 228
-----------------------------------------------■
```

6.1.3. Захтев за покретањем скупљача отпадака, метод System.gc()

Као што је раније речено, Јава аутоматски уклања са хипа податке који више нису у употреби, тј. на које се више не референцира, кроз тзв. систем скупљача отпадака (енг. garbage collector). Скупљач отпадака се активира по потреби и његов рад се заснива на коришћењу разних хеуристика. Захваљујући хеуристикама, скупљач отпадака се прилагођава конкретном процесу, предвиђа употребу меморије и слично. Квалитет скупљања отпадака може се оценити коришћењем два критеријума:

1. колико је у просеку неки објекат „чекао“ у меморији од момента када је постао непотребан до момента уклањања и
2. колики ефекат на перформансе целе виртуелне машине има рад скупљача отпадака.

Ова два критеријума су често у колизији, тј. минимизација времена чекања на уклањање подразумева чешће активације скупљача отпадака и самим тим смањење перформанси.

(Напомена: добро написан C програм, у којем се меморија експлицитно ослобађа чим више није неопходна (наредбом `free()`), нуди максималне перформансе уз минимизацију ефекта “чекања”.)

Програмер, дакле, у Јави не може уклањати податке из меморије самостално. Највише што може је да сугерише скупљачу отпадака да се активира преко метода `System.gc()`. Ретке су ситуације у којима програмер може да има користи од експлицитног позивања овог метода и генерално се његово позивање не препоручује.

Пример 3. Написати Јава програм који користи једну референцну променљиву за креирање великог броја нових ниска објеката на хипу. Ово за последицу има велики број нереферисаних објеката током рада програма. Упоредити количину доступне и слободне меморије на хипу у варијантама:

1. када скупљач сам одређује кад ће се активирати и
2. када му корисник повремено сугерише скупљање помоћу `System.gc()` метода.

□

```
public class PoziviGC
    static void stanjeMemorije() {
        long slobodno = Runtime.getRuntime().freeMemory() / 1024 / 1024;
        long maksimalno = Runtime.getRuntime().totalMemory() / 1024 / 1024;
        System.out.printf("%dMB од %dMB | ", slobodno, maksimalno);
    }

    public static void main(String[] args) {
        int n = 10000000;
        int nIspis = 200000;
        String s;
        long pocBezGC = System.nanoTime();
        System.out.println(
                    "Величине слободне меморије без позивања gc() пред испис");
```

```
        for (int i = 0; i < n; i++) {
                s = new String("Текст под редним бројем " + i);
                if (i % nIspis == 0)
                        stanjeMemorije();
        }
        System.out.printf("%nВреме без GC\t%.2f%n",
                        (System.nanoTime() - pocBezGC) / 1e6);

        long pocSaGC = System.nanoTime();
        System.out.println(
                        "Величине слободне меморије са позивањем gc() пред испис");
        for (int i = 0; i < n; i++) {
                s = new String("Текст под редним бројем " + i);
                if (i % nIspis == 0) {
                        // експлицитно позивање скупљања отпадака
                        // изазива смањење перформанси
                        System.gc();
                        stanjeMemorije();
                }
        }
        System.out.printf("%nВреме са GC\t%.2f%n",
                        (System.nanoTime() - pocSaGC) / 1e6);
    }
}
```

У варијанти без употребе `System.gc()` метода види се да програм ради са већом (флексибилнијом) количином максималне меморије, што омогућава да скупљач не мора често да се активира. Тек повремено, приликом значајног смањења удела слободне меморије, скупљач се самостално активира и обрише непотребне податке. Са друге стране, у режиму експлицитног позивања скупљача отпадака, количина слободне меморије је врло близу максимуму. Време извршавања је у другом случају нарушено, што је очекивана последица скупог процеса скупљања отпадака.

Следи резултат рада програма.

```
Величине слободне меморије без позивања gc() пред испис
1021MB од 1024MB | 978MB од 1024MB | 987MB од 1024MB | 973MB од 1024MB | 957MB од 1024MB
| 941MB од 1024MB | 927MB од 1024MB | 1015MB од 1024MB | 999MB од 1024MB | 983MB од
1024MB | 967MB од 1024MB | 955MB од 1024MB | 939MB од 1024MB | 923MB од 1024MB | 907MB од
1024MB | 891MB од 1024MB | 879MB од 1024MB | 863MB од 1024MB | 847MB од 1024MB | 831MB од
1024MB | 815MB од 1024MB | 799MB од 1024MB | 787MB од 1024MB | 771MB од 1024MB | 755MB од
1024MB | 739MB од 1024MB | 723MB од 1024MB | 711MB од 1024MB | 695MB од 1024MB | 679MB од
1024MB | 663MB од 1024MB | 647MB од 1024MB | 631MB од 1024MB | 619MB од 1024MB | 603MB од
1024MB | 587MB од 1024MB | 571MB од 1024MB | 555MB од 1024MB | 543MB од 1024MB | 527MB од
1024MB | 511MB од 1024MB | 495MB од 1024MB | 479MB од 1024MB | 463MB од 1024MB | 451MB од
1024MB | 435MB од 1024MB | 1019MB од 1024MB | 1003MB од 1024MB | 987MB од 1024MB | 975MB
од 1024MB |
Време без GC    373.09
Величине слободне меморије са позивањем gc() пред испис
55MB од 56MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB |
31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB |
31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB |
31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB |
31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB |
31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 55MB од 56MB |
31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB |
31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB |
```

```
31MB од 32MB | 31MB од 32MB |
Време са GC    445.92■
```

6.1.4. Излазак из апликације, метод System.exit()

Попут функције `exit()` у програмском језику C и Јава има могућност да прекине извршавање актуелног програма. (У Јави се заправо гаси читава Јава виртуелна машине.) Метод је статички и налази се у класи `System`. Његов потпис је следећи:

```
public static void exit(int status)
```

Статус 0 указује на успешан завршетак извршавања док не-нула вредности указују на неуспешно извршавање, тј. на грешку.

Пример 4. Написати Јава програм који реализује и тестира метод за исписивање карактеристика монитора, при чему су аргументи метода ширина и висина. Потребно је израчунати број тачака (пиксела) и сврстати монитор у категорију стандардни или широки на основу односа ширине и висине: монитор је широк уколико је ширина два или више пута већа од висине. Такође је потребно на одговарајући начин реаговати на некоректне уносе. □

```
public class IzadjiIzAplikacije {
    static void karakteristikeMonitora(int sirina, int visina) {
        if (sirina <= 0 || visina <= 0) {
                System.err.println("Ширина и висина морају бити позитивни бројеви.");
                System.exit(1);
        }
        System.out.println("Ширина:\t" + sirina);
        System.out.println("Висина:\t" + visina);
        System.out.println("Број тачака:\t" + sirina * visina);
        System.out.println("Тип монитора:\t" +
        (sirina >= 2 * visina ? "широки" : "стандардни"));
    }

    public static void main(String[] args) {
        karakteristikeMonitora(1024, 1024);
        karakteristikeMonitora(1920, 768);
        karakteristikeMonitora(0, 230);
        System.exit(0);
    }
}
```

Приликом позива метода `System.exit()` излази се из читавог програма, а не само из тренутног метода, као што је случај са позивом `return`.

Као последњу наредбу метода `main()` могуће је, али није обавезно, ставити наредбу `System.exit(0)` – она указује на регуларан завршетак извршавања програма. Ако се ова наредба изостави, подразумеваће се да је извршавање протекло регуларно.

Резултат рада програма је дат испод.

```
Ширина:    1024
Висина:    1024
Број тачака:    1048576
Тип монитора:    стандардни
Ширина:    1920
Висина:    768
Број тачака:    1474560
```

```
Тип монитора:    широки
```

6.2. Рад са нискама, инстанцама класе String

Ниске (стрингови) представљају један од најчешће коришћених типова података не само у Јави, већ и у другим програмским језицима. У програмском језику C ниска је била представљена као низ карактера. Јава користи исту идеју, али енкапсулира тај низ карактера у класу под називом `String`. За разлику од C-а, не захтева се да се ниска завршава унапред дефинисаним карактером (терминирајућом нулом).

Каракеристике ниски, имутабилност

Ниска, као и сваки други низ, омогућава приступ карактерима у временској сложености O(1).
Ниска у Јави је имутабилна (непроменљива), што значи да измена енкапсулираног низа карактера није дозвољена. Ово се односи на било који тип измене: брисање, промена, додавање карактера итд. Било какав покушај измене објекта класе `String` доводи до креирања новог објекта који садржи полазну ниску измењену на тражени начин.

Креирање ниски и неке стандардне методе над ниском

Ниска се може креирати на неколико начина, а најједноставнији је употребом литерала ниске.

```
String s = "Литерал ниске";
```

У овом случају Јава ће претражити тзв. списак константних ниски (енг. string constant pool) и, ако такав литерал већ постоји, променљива `s` ће поставити референцу ка њему. Овај део меморије има специјални статус (подскуп хипа), јер су подаци који се у њему налазе очигледно познати већ у фази компилације. Дакле, у примеру испод обе променљиве показују на исту меморију.

```
String s1 = "Литерал ниске";
String s2 = "Литерал ниске";
```

Са друге стране, ако се ниска креира уз експлицитну употребу оператора `new`, форсираће се креирање нове инстанце на хипу, без обзира што је садржај ниске познат већ у фази компилације. Поред тога, креираће се литерал ниске и сместити у списак константних ниски.

```
String s1 = new String("Литерал ниске");
String s2 = new String("Литерал ниске"); // s1 и s2 показују на различито
```

Дакле, ефекат кода изнад је да ће бити креирана два објекта `String` на хипу и један ниска литерал. Променљиве `s1` и `s2` ће реферисати на објекте са хипа, а не ка оном из скупа константних ниски.
Могуће је креирање ниске на основу низа карактера. Овај случај је попут претходног, само што не укључује креирање константне ниске.

```
char[] niz = new char[] {'к','а','р','а','к','т','е','р','и'};
String s = new String(niz);
```

Ниска не мора бити позната у фази компилације, тј. може постати позната тек у фази извршавања. Пример који демонстрира такав сценарио ће бити дат нешто касније у

секцији 6.4. У табели испод дат је преглед неких стандардних метода у оквиру класе `String`.

Метод	Опис
`charAt(int i)`	Враћа карактер на позицији `i`.
`compareTo(String s)`	Лексикографски упоређује ниску са прослеђеном ниском `s`.
`concat(String s)`	Надовезује ниску са прослеђеном и враћа нову ниску као повратну вредност. Сличну ствар ради и оператор `+` над нискама.
`contains(String s)`	Проверава да ли ниска садржи подниску `s`.
`endsWith(String s)`	Проверава да ли се ниска завршава са подниском `s`.
`equals(String s)`	Упоређује на једнакост ниску са прослеђеном ниском `s`.
`indexOf(char c)`	Враћа позицију првог појављивања карактера `c` или `-1` ако не постоји.
`lastIndexOf(char c)`	Враћа позицију последњег појављивања карактера `c` или `-1` ако не постоји.
`length()`	Враћа дужину ниске.
`replace(String s, String r)`	Замењује сва појављивања подниске `s` са подниском `r` и враћа тако добијену новоформирану ниску.
`split(String r)`	Раздваја ниску на низ подниски у складу са прослеђеним начином раздвајања (регуларни израз `r`).
`startsWith(String s)`	Проверава да ли ниска започиње подниском `s`.
`substring(int i)`	Враћа подниску која почиње на позицији `i`. Додатно може постојати и индекс на којем се подниска завршава.
`toCharArray()`	Враћа низ карактера од којих је сачињена ниска.
`toLowerCase()`	Враћа ниску у којој су сва слова пребачена у мала.
`toUpperCase()`	Враћа ниску у којој су сва слова пребачена у велика.
`trim()`	Враћа ниску у којој је уклоњен празан простор на крајевима.

Пример 5. Написати Јава програм који задату реченицу обрће на нивоу речи. Дакле, слова унутар појединачних речи треба да остану у правом редоследу, али редослед речи унутар реченице треба да постане обрнут. Претпоставити да су једини типови сепаратора (унутар реченице) карактери за празно место. Такође, прилагодити почетно велико слово унутар новоформиране реченице. □

```
public class ObrniReciURecenici {
    public static void main(String[] args) {
        String recenica = "Ниска у Јави је имутабилна (непроменљива) што значи да "
                        + "измена енкапсулираног низа карактера није дозвољена.";
        String[] reci = recenica.split(" ");
        if(reci.length==0)
        {
                System.out.println("Нема шта да се обрне.");
                return;
        }
```

```
        int kraj = reci.length-1;
        reci[kraj]=reci[kraj].replace(".", "");
        reci[0] = reci[0].substring(0,1).toLowerCase()+reci[0].substring(1);
        reci[kraj]= reci[kraj].substring(0,1).toUpperCase()+reci[kraj].substring(1);
        String novaRecenica = reci[kraj];
        for(int i=kraj-1; i>=0; i--)
               novaRecenica = novaRecenica.concat(" "+reci[i]);
        novaRecenica+=".";
        System.out.println(novaRecenica);
    }
}
```

Најпре се уклања тачка са последње речи. Након тога следи смањивање почетног великог слова прве речи – овим се демонстрира употреба метода `substring()` и `toLowerCase()` (могло је да се реализује и једноставније, само позивом `toLowerCase()` над целом речи). Слично се врши повећавање првог слова последње речи. На крају се речи спајају у обрнутом редоследу. Спајање се врши применом метода `concat()`, који се понаша исто као и оператор `+`. Приликом надовезивања сваке нове речи формира се нова (проширена) ниска на хипу, након чега се референца ка њој смешта у променљиву `novaRecenica`.

Резултат извршавања програма је дат испод.

```
Дозвољена није карактера низа енкапсулираног измена да значи што (непроменљива)
имутабилна је Јави у ниска.■
```

Поређење ниски

Могуће је поредити ниске на два начина:

1. испитивати да ли су једнаке или
2. испитивати да ли је једна ниска испред друге на основу лексикографског уређења.

За поређење да ли су две ниске једнаке користе се следећи методи:

```
public boolean equals(Object o)
public boolean equalsIgnoreCase(String s)
```

Метод `equals()` је наслеђен од класе `Object` и прилагођен (превазиђен) у складу са начином поређења ниски. Да би две ниске биле исте морају имати исту дужину и на свакој упоредној позицији морају имати исте карактере. Метод `equalsIgnoreCase()` ради сличну ствар, али дозвољава да карактери на упоредним позицијама не морају бити идентични у случају да је реч о словима. Карактери се сматрају истим уколико представљају исто слово, без обзира на величину (мало или велико слово).

```
String rec1 = "Ниска";
String rec2 = "Niska";
String rec3 = "нискА";
String rec4 = "Ниска";
String rec5 = new String("Ниска");
System.out.println(rec1==rec1); // true (поређење референце, увек тачно)
System.out.println(rec1==rec4); // true (константна меморија, показују на исто)
System.out.println(rec1==rec5); // false (једна у константној, друга не)
System.out.println(rec1.equals(rec2)); // false (различити карактери, писмо)
System.out.println(rec1.equals(rec3)); // false (осетљиво на величину слова)
System.out.println(rec1.equals(rec4)); // true (идентичан садржај)
System.out.println(rec1.equals(rec5)); // true (идентичан садржај)
```

```
System.out.println(rec1.equalsIgnoreCase(rec2)); // false (различити карактери)
System.out.println(rec1.equalsIgnoreCase(rec3)); // true (иста до на величину)
System.out.println(rec1.equalsIgnoreCase(rec4)); // true (идентичан садржај)
```

Класа StringBuilder

Непроменљивост ниски може представљати проблем за ефикасно коришћење хип меморије у случају када је потребно вршити честе „измене“ над нискама. Као што је објашњено, измене ниски нису директно могуће, већ се у позадини прави нови простор на хипу са новом ниском која представља измењену варијанту полазне ниске.

Пример 6. Написати Јава програм који креира ниску добијену надовезивањем задате ниске одређени број пута. За реализацију користити само класу `String`. □

```
public class PonoviNisku {
    public static void main(String[] args) {
        String s = "Тест";
        int n = 10;
        String sn = "";
        for(int i=0; i<n; i++)
               sn+=s;
        System.out.println(sn);
    }
}
```

Сваки пут када се позове оператор за надовезивање (+), на хипу се креира нова ниска потребне димензије и потом се референца ка њој запамти у оквиру променљиве `sn`. Претходна ниска након тога постаје нереферисана, тј. отпадак који ће у неком моменту бити елиминисан од стране скупљача отпадака. Међутим, имајући у виду да скупљач отпадака периодично елиминише отпатке, а не моментално, доћи ће до акумулације отпадака, тј. претходних верзија ниски. Конкретно, након прве итерације биће нереферисана само ниска „“, након друге биће нереферисана ниска „“ и ниска „Тест“, након треће ниске: „“, „Тест“ и „ТестТест“ итд. Види се да ће количина нереферисане меморије расти квадратно са бројем итерација (сума аритметичког низа).

Следи резултат рада програма.

```
ТестТестТестТестТестТестТестТестТестТест ■
```

Погодан начин за превазилажење проблема са меморијом, приликом интензивног рада са нискама, заснива се на употреби класе `StringBuilder`. За разлику од класе `String`, чије инстанце су непроменљиве, класа `StringBuilder` енкапсулира изменљиви низ карактера којим манипулише на ефикасан начин. На крају, када се формира циљни низ карактера, врши се његова једнократна конверзија у `String` објекат. `StringBuilder` нуди методе који омогућавају индиректну манипулацију унутрашњим низом карактера. На тај начин могуће је његово динамичко повећавање, тј. не захтева се од програмера да води рачуна о реалокацији низа.

Пример 7. Написати Јава програм који креира ниску добијену надовезивањем задате ниске одређени број пута. За реализацију користити и класу `StringBuilder`. □

```
public class PonoviNiskuStringBuilder {
    public static void main(String[] args) {
        String s = "Тест";
        int n = 10;
        StringBuilder sb=new StringBuilder();
```

```
        for(int i=0; i<n; i++)
                sb.append(s);
        // једнократна конверзија у ниску
        String sn = sb.toString();
        System.out.println(sn);
    }
}
```

Последица оваквог приступа је избегавање акумулације нереферисаних података. Резултат рада програма је исти као и раније.

```
ТестТестТестТестТестТестТестТестТестТест ■
```

6.3. Рад са омотачима података примитивног типа

Омотачи примитивних типова, као што и само име каже, представљају објекте који енкапсулирају примитивне типове. Сваки од примитивних типова `byte`, `short`, `char`, `int`, `long`, `float` и `double` има, редом, свој омотач тип `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float` и `Double`. У Јави постоји добра подршка за рад са методима, структурама података (колекцијама) или конструкцијама језика попут генеричких типова (поглавље 13). Сви ови елементи Јаве су прилагођени раду са објектима, али не и са примитивним типовима. Постојање омотача примитивних типова омогућава елегантан начин да се та подршка пренесе индиректно и на примитивне типове.

Познато је да за примитивне типове Јава подржава искључиво прослеђивање по вредности аргумената методима. Дакле, кад год се пренесе променљива у метод, прави се њена копија. Уколико је променљива инстанцна, онда то што је у питању њена копија није проблем, јер копија референце и даље показује на оригинални објекат и омогућена је његова измена. С друге стране, ако се пренесе променљива примитивног типа, онда изменом њене копије оригинална променљива остаје непромењена, што није увек жељени сценарио употребе. На основу овога би се могло закључити да омотач-објекти омогућавају промену вредности коју садрже унутар метода. Међутим, ово ипак није случај будући да су омотач типови имутабилни, попут ниски.

Обмотавање примитивних типова

Процес обмотавања примитивног типа (енг. boxing) је омогућен применом конструктора одговарајућег омотач типа. На пример, класа `Integer` поседује конструктор који прихвата као аргумент променљиву типа `int`. По истом принципу су конципирани и конструктори осталих омотач типова.

Алтернативно, обмотавање је могуће и употребом статичког метода `valueOf()` у оквиру одговарајуће омотач класе.

```
public static Integer valueOf(int x)
```

Почев од Јаве 5 могуће је и аутоматско обмотавање примитивних типова у којем се примитивна променљива једноставно може доделити променљивој омотач типа. У овој ситуацији Јава у позадини генерише проширени програмски кôд који садржи позив ка претходно поменутом методу `valueOf()`.

```
int x=242;
Integer y = new Integer(x);  // обмотавање конструктором
Integer z=Integer.valueOf(x); // обмотавање статичким методом
```

```
Integer w=x; // аутоматско обмотавање
```

Одмотавање омотач типова

Супротан поступак обмотавању је одмотавање омотач типова, те добијање одговарајућих примитивних типова. Ово је омогућено применом одговарајућег метода објекта омотача. На пример објекат класе `Integer` поседује метод `intValue()` чији је потпис:

```
public int intValue()
```

Почев од Јаве 5 омогућено је аутоматско одмотавање објекта једноставном доделом објектне омотач променљиве њеној примитивној варијанти.

```
Integer x=new Integer(2311);
int y=x.intValue(); // експлицитно одмотавање
int z=x; // аутоматско отпакивање
```

Пример 8. Написати и тестирати Јава метод у којем се израчунава квадрат прослеђеног реалног броја. Проверити понашање у случају да је аргумент примитивног типа, као и случају да је реч о омотач типу. □

```
public class MutabilniFloat {
    float val;

    public MutabilniFloat(float val) { this.val=val; }

    static void kvadrirajPrimitvni(float x) { x*=x; }

    static void kvadrirajUgradjeniOmotac(Float x) { x*=x; }

    static void kvadrirajMutabilniOmotac(MutabilniFloat x) { x.val*=x.val; }

    public static void main(String[] args) {
        float x = 11;
        Float xOmotac = x;
        MutabilniFloat xMutabilniOmotac = new MutabilniFloat(x);

        kvadrirajPrimitvni(x);
        System.out.println(x);

        kvadrirajUgradjeniOmotac(xOmotac);
        System.out.println(xOmotac);

        kvadrirajMutabilniOmotac(xMutabilniOmotac);
        System.out.println(xMutabilniOmotac.val);
    }
}
```

Овај пример демонстрира три покушаја измене прослеђене променљиве унутар метода. Први приступ, слање примитивне променљиве, очекивано не ради. Разлог је прослеђивање по вредности, приликом којег се прави копија вредности примитивне променљиве. Изменом копије вредности се не мења оригинална променљива `x`. Други приступ обећава, јер је сада у питању објекат омотач типа. Међутим, због имутабилности омотач типова, ни овај покушај није успешан, јер се унутар метода за рачунање квадрата, приликом измене инстанцне променљиве, дешава креирање новог објекта. Последњи покушај омогућава захтевано понашање и заснован је на

прилагођеној и поједностављеној реализацији омотач типа чиме је омогућена мутабилност. Следи резултат извршавања.

```
11.0
11.0
121.0 ■
```

6.4. Рад са скенерима, инстанцама класе java.util.Scanner

Класа `Scanner` омогућава учитавање текста и његово парсирање на велики број начина. Текст може бити унет у виду ниске, са стандардног улаза, датотеке или неког другог типа улазног тока података (поглавље 15). Скенер раздваја улазни текст у складу са постављеним сепаратором, који је подразумевано празан простор. У њему је дефинисан већи број метода за парсирање неких стандардних типова података попут целих бројева, реалних бројева, речи, целих линија текста, логичких вредности и слично.

```
public String next()
public String nextLine()
public boolean nextBoolean()
public int nextInt()
public float nextFloat()
...
```

У комбинацији са горенаведеним методима за парсирање обично се користе и методи за испитивање да ли се парсирање може извршити.

```
public boolean hasNext()
public boolean hasNextLine()
public boolean hasNextBoolean()
public boolean hasNextInt()
public boolean hasNextFloat()
...
```

Скенирање података из ниске

Ниска се може користити у комбинацији са скенером како би се из ње „извукле“ потребне информације.

Пример 9. Написати Јава програм који из текста извлачи информације о просечној години производње, снази и запремини мотора аутомобила. Текст је сачињен од реченица, где свака има следећу структуру: „[Марка] [Модел] [Година производње] [Снага] KS [Запремина] cm3.“. □

```
public class ParsirajAutoInformacije {
    static void greska(String poruka) {
        System.err.println(poruka);
        System.exit(1);
    }

    public static void main(String[] args) {
        String automobili = "Peugeot 3008 2017 120 KS 1560 cm3. "
                          + "Suzuki Vitara 2018 120 KS 1600 cm3. "
                          + "Toyota RAV4 2021 175 KS 1998 cm3.";
        double prosecnaSnaga = 0, prosecnaZapremina = 0, prosecnaGodina = 0;
        int brojAutomobila = 0;
```

```
        java.util.Scanner skener = new java.util.Scanner(automobili);
        while (skener.hasNext()) {
                int snaga, zapremina, godina;
                if (!skener.hasNext())
                        greska("Марка недостаје.");
                skener.next();
                if (!skener.hasNext())
                        greska("Модел недостаје.");
                skener.next();
                if (!skener.hasNextInt())
                        greska("Година производње недостаје.");
                godina = skener.nextInt();
                if (!skener.hasNextInt())
                        greska("Снага мотора недостаје.");
                snaga = skener.nextInt();
                if (!skener.hasNext("KS"))
                        greska("Ознака KS недостаје.");
                skener.next("KS");
                if (!skener.hasNextInt())
                        greska("Запремина мотора недостаје.");
                zapremina = skener.nextInt();
                if (!skener.hasNext("cm3."))
                        greska("Ознака cm3. недостаје.");
                skener.next("cm3.");

                prosecnaGodina += godina;
                prosecnaSnaga += snaga;
                prosecnaZapremina += zapremina;
                brojAutomobila++;
        }
        prosecnaGodina/=brojAutomobila;
        prosecnaSnaga/=brojAutomobila;
        prosecnaZapremina/=brojAutomobila;
        skener.close();
        if (brojAutomobila > 0) {
                System.out.println(String.format("Просек година:\t%.2f%n"
                                + "Просек снаге:\t%.2f%nПросек запремине:\t%.2f%n",
                                prosecnaGodina, prosecnaSnaga, prosecnaZapremina));
        }
    }
}
```

Програм остаје у `while` петљи док год постоји текст за парсирање (`skener.hasNext()`). Након тога се комбиновањем `hasNext…()` и `next…()` метода проверава постојање одговарајућег податка и његово парсирање. У случају непостојања очекиваног податка (лоше структура улазних података) моментално се прекида извршавање програма. Методом `close()` се врши затварање ресурса придружених скенер објекту.
Резултат рада програма је дат испод.

```
Просек година:    2018.67
Просек снаге:    138.33
Просек запремине:    1719.33■
```

Скенирање података са стандардног улаза

Честа примена скенера је читање и парсирање информација са стандардног улаза (обично је то тастатура). У поређењу са уносом ниске нема скоро никаквих разлика, осим што се, приликом креирања скенера, прослеђује другачији ток података – `System.in`. О стандардном излазу је било речи у секцији 6.1.1. Овде је принцип рада исти, осим што је смер комуникације обрнут.

Пример 10. Написати Јава програм који рачуна просек реалних бројева унетих са стандардног улаза. На почетку уноса корисник унапред најављује колико бројева жели да унесе, након чега следи њихов унос у сваком наредном новом реду. □

```
public class ProsekUnetihBrojeva {
    public static void main(String[] args) {
        java.util.Scanner skener = new java.util.Scanner(System.in);
        System.out.println("Колико бројева уносите: ");
        int n = skener.nextInt();
        double prosek = 0;
        for (int i = 0; i < n; i++) {
               System.out.println("Број " + (i + 1) + ": ");
               double b = skener.nextDouble();
               prosek += b;
        }
        prosek /= n;
        System.out.println(String.format("Просек унетих бројева је: %.2f", prosek));
        skener.close();
    }
}
```

Следи пример једног извршавања програма.

```
Колико бројева уносите:
4
Број 1:
33.3
Број 2:
11
Број 3:
1000
Број 4:
12.2
Просек унетих бројева је: 264.13■
```

6.5. Рад са математичким функцијама, класа Math

Јава подржава рад са разноврсним математичким функцијама. Позивање математичких функција се врши статички путем услужне класе `Math` (попут класе `System` описане у секцији 6.1). Табела испод даје преглед неких чешће коришћених метода.

Метод	**Опис**
abs(double x)	Апсолутна вредност (постоји и за друге бројевне типове).
ceil(double x)	Број заокружен на горе.
cos(double x)	Косинус.
exp(double x)	e^x

floor(double x)	Број заокружен на доле.
log(double x)	Природни логаритам броја.
max(double x, double y)	Максимум два броја (постоји и за друге бројевне типове).
min(double x, double y)	Минимум два броја (постоји и за друге бројевне типове).
pow(x, y)	x^y
random()	Псеудослучајан број из [0, 1].
round(double x)	Заокружује број.
signum(double x)	Знак броја.
sin(double x)	Синус.
sqrt(double x)	Корен броја x.
toDegrees(double rad)	Пребацује радијане у степене.
toRadians(double deg)	Пребацује степене у радијане.

Пример 11. Написати Јава програм који врши нумеричку Монте Карло интеграцију неке од следећих математичких функција: `sqrt()`, `log()` или `exp()` на задатом интервалу [a, b], a>=1, и са задатим бројем узорака N>0, који се прави приликом Монте Карло интеграције. Идеја Монте Карло интеграције је заснована на хоризонталном и вертикалном ограничавању графика подинтегралне функције, на задатом интервалу, на пример, правоугаоником. Након тога се генеришу координате дводимензионалних тачака, на случајан начин, тако да припадају изабраном правоугаонику. На основу односа броја тачака чије ординате су мање од вредности функције за случајно изабране апцисе („упадају“ испод графика функције) и укупног броја генерисаних тачака, апроксимира се вредност интеграла. (Ово важи у случају горенаведених функција — не важи увек). □

```
public class MonteKarloIntegracija {
    static void greska(String poruka) {
        System.err.println(poruka);
        System.exit(1);
    }

    static double primeniFunkciju(String funkcija, double x) {
        if (funkcija.equals("sqrt"))
               return Math.sqrt(x);
        else if (funkcija.equals("log"))
               return Math.log(x);
        else
               return Math.exp(x);
    }

    public static void main(String[] args) {
        String funkcija;
        double a, b;
        int N;

        java.util.Scanner skener = new java.util.Scanner(System.in);
        System.out.println("Одаберите функцију sqrt, log или exp:");
```

```
        funkcija = skener.next();
        if (!funkcija.equals("sqrt") && !funkcija.equals("log") &&
                        !funkcija.equals("exp"))
                greska("Некоректна функција.");
        System.out.println("Унесите a:");
        if (!skener.hasNextDouble())
                greska("a треба да буде број.");
        a = skener.nextDouble();
        if (a < 1)
                greska("a треба да буде >= 1.");
        System.out.println("Унесите b:");
        if (!skener.hasNextDouble())
                greska("b треба да буде број.");
        b = skener.nextDouble();
        if (b < a)
                greska("b треба да буде >= a.");
        System.out.println("Унесите N:");
        if (!skener.hasNextInt())
                greska("N треба да буде цео број.");
        N = skener.nextInt();
        if (N < 1)
                greska("N треба да буде > 0.");
        double donja = 0;
        double gornja = primeniFunkciju(funkcija, b);
        int yUnutra = 0;
        for(int i=0; i<N; i++){
                double x = Math.random()*(b-a)+a;
                double y = Math.random()*(gornja-donja)+donja;
                double vred = primeniFunkciju(funkcija, x);
                if(y<vred)
                        yUnutra++;
        }
        double integral = (b-a)*(gornja-donja)*yUnutra/N;
        System.out.println("Апроксимација интеграла је "+integral);
        skener.close();
    }
}
```

Након уноса података врши се Монте Карло интеграција. У свакој итерацији `for` петље бира се случајан пар бројева (`x`, `y`) где се `x` бира пседослучајно униформно из интервала [`a`, `b`], а `y` из интервала [`donja`, `gornja`]. За доњу границу се узима вредност 0 док се за горњу границу узима вредност функције у тачки `b`, јер су све наведене функције монотоно растуће. Након тога се проверава да ли тако случајно одабрана тачка „упада“ у график функције. На крају се интерал апроксимира умношком: 1) површине правоугаоника чије су странице паралелне координатним осама, а наспрамна темена (`x`, `donja`) и (`y`, `gornja`) и 2) односа броја тачака које су “упале” и укупног броја тачака. Резултат рада програма за неколико различитих извршавања је дат испод.

```
Одаберите функцију sqrt, log или exp:
log
Унесите a:
2
Унесите b:
6.5
Унесите N:
```

```
100
Апроксимација интеграла је 6.317332347042871
-----------------------------------------
Одаберите функцију sqrt, log или exp:
sqrt
Унесите a:
1
Унесите b:
12.2
Унесите N:
200
Апроксимација интеграла је 28.36194069523452
-----------------------------------------
Одаберите функцију sqrt, log или exp:
exp
Унесите a:
3
Унесите b:
6.5
Унесите N:
10000
Апроксимација интеграла је 645.0876128080744
-----------------------------------------
Одаберите функцију sqrt, log или exp:
sqrt
Унесите a:
0
a треба да буде >= 1.■
```

6.6. Рад са датумима и временима

Ранија подршка за рад са датумима и временом је била обезбеђена преко класа `Date` и `Calendar`. Поменуте класе су и даље доступне и чине део стандардних Јава библиотека. Међутим, препорука је да се уместо њих (због одређених проблема у раду) користе новије класе `LocalDate`, `LocalTime` и `LocalDateTime` за представљање, редом, датума без времена, времена у току дана без датума и комбинованог датума и времена. Све три класе су имутабилне. Још једна значајна класа је `DateTimeFormatter`, која омогућава форматирање.

Класа `LocalDate` представља датум у ISO-8601 формату (yyyy-MM-dd), без придруженог времена и временске зоне. Може бити корисна када у програмима желимо да запамтимо, на пример, празнике, рођендане и слично. Креирање инстанце ове класе која одговара тренутном датуму могуће је записати на следећи начин.

```
java.time.LocalDate danas = java.time.LocalDate.now();
```

Алтернативно се могу користити и статички методи за креирање било ког датума задавањем године, месеца и дана у виду бројева или у виду текста.

```
java.time.LocalDate datum1 = java.time.LocalDate.of(2021, 11, 25);
java.time.LocalDate datum2 = java.time.LocalDate.parse("2021-11-25");
```

`LocalDate` поседује велики број корисних метода. Преглед неких интересантнијих је дат испод.

Метода	Опис

`atStartOfDay()`	Враћа инстанцу `LocalDateTime` класе која одговара почетку дана.
`atTime(int s, int m)`	Враћа инстанцу `LocalDateTime` која одговара прослеђеном сату и минуту. Постоје и методи са прецизнијим временом, тј. секундама и наносекундама.
`format(DateTimeFormatter f)`	Форматирање датума у складу са прослеђеним форматом.
`getDayOfMonth()`	Редни број дана у току месеца.
`getDayOfWeek()`	Дан у току недеље.
`getDayOfYear()`	Редни број дана у току године.
`getMonth()`	Месец у току године.
`getYear()`	Година.
`isAfter(ChronoLocalDate d)`	Да ли је датум хронолошки после датума `d`. `ChronoLocalDate` је интерфејс који, између осталих, имплементира и класа `LocalDate`.
`isBefore(ChronoLocalDate d)`	Да ли је датум хронолошки пре датума `d`.
`isEqual(ChronoLocalDate d)`	Да ли су датуми једнаки.
`isLeapYear()`	Да ли је преступна година.
`lengthOfMonth()`	Број дана у месецу.
`minusDays(long d)`	Враћа нову инстанцу датума која је умањена за број дана. Подржано је одузимање и других стандардних и произвољних временских јединица.
`plusMonths(long m)`	Враћа нову инстанцу датума која је повећана за број месеци.
`now()`	Статичка метода која враћа тренутни датум.
`toEpochDay()`	Враћа број дана од дана Епохе, тј. од 1. јануара 1970. године.
`withYear(int g)`	Враћа нову инстанцу датума у којој је година замењена прослеђеном. Подржано и за друге компоненте времена: месеце, дане у месецу.

Пример 12. Написати Јава програм који најпре очекује од корисника да унесе месец и годину, а потом на излазу графички приказује одабрани месец у виду табеле где су колоне дани у недељи, а редови недеље у оквиру месеца. Притом је у ћелији табеле уписан редни број дана у месецу. □

```
public class PrikaziMesec {
    public static void main(String[] args) {
        java.util.Scanner skener = new java.util.Scanner(System.in);
        System.out.println("Унесите годину ");
        int godina = skener.nextInt();
        System.out.println("Унесите редни број месеца");
        int mesec = skener.nextInt();
        if (mesec < 1 || mesec > 12) {
               System.err.println("Редни број месеца је између 1 и 12.");
               System.exit(1);
        }
```

```
        System.out.println("Пон.\tУто.\tСре.\tЧет.\tПет.\tСуб.\tНед.");
        java.time.LocalDate dan = java.time.LocalDate.of(godina, mesec, 1);
        int danUNedeljiRbr = dan.getDayOfWeek().getValue();
        for (int i = 1; i < danUNedeljiRbr; i++)
                System.out.print("\t");
        do {
                System.out.print(dan.getDayOfMonth() + "\t");
                dan = dan.plusDays(1);
                danUNedeljiRbr = dan.getDayOfWeek().getValue();
                if (danUNedeljiRbr == 1)
                        System.out.println();
        } while (dan.getDayOfMonth() != 1);
        skener.close();
    }
}
```

Прва петља прави одговарајући размак за први дан у месецу, на пример, ако је први дан у месецу среда, онда ће се додати два размака у виду табулатор карактера.
Наредна петља приказује остале дане у месецу на одговарајућој позицији – одговарајућем реду и колони (за назив дана у оквиру недеље).
Следи пример једног извршавања програма.

```
Унесите годину
1986
Унесите редни број месеца
11
Пон.    Уто.    Сре.    Чет.    Пет.    Суб.    Нед.
                                         1       2
3       4       5       6       7       8       9
10      11      12      13      14      15      16
17      18      19      20      21      22      23
24      25      26      27      28      29      30
```

Класа `LocalTime` представља време у ISO-8601 формату (yyyy-MM-dd) без придруженог датума и временске зоне. Прецизност је до нивоа наносекунди.
Креирање инстанце ове класе која одговара тренутном датуму се врши на следећи начин:

```
java.time.LocalTime sad = java.time.LocalTime.now();
```

Алтернативно се могу користити и статички методи за креирање произвољног времена у току дана задавањем сата, минута, секунде, наносекунде у виду бројева или у виду текста.

```
java.time.LocalTime vreme1 = java.time.LocalTime.of(9, 42, 10);
java.time.LocalTime vreme2 = java.time.LocalTime.parse("09:42:10");
```

`LocalTime` поседује велики број корисних метода. Преглед неких интересантнијих је дат испод.

Метода	**Опис**
`atDate(LocalDate d)`	Враћа инстанцу `LocalDateTime` која одговара прослеђеном датуму.
`format(DateTimeFormatter f)`	Форматирање времена у складу са прослеђеним форматом.
`getHour()`	Сат у дану 0-23.

`getMinute()`	Минут у сату 0-59.
`getSecond()`	Секунда у минути 0-59.
`getNano()`	Наносекунда у секунди 0-999,999,999.
`isAfter(LocalTime v)`	Да ли је време хронолошки после времена `v`.
`isBefore(LocalTime v)`	Да ли је време хронолошки пре времена `v`.
`minusHours(long s)`	Враћа нову инстанцу времена умањену за број сати. Подржано је одузимање и минута, секунди и наносекунди. У случаја прелаза на претходни или наредни дан примењује се модуларна аритметика.
`plusSeconds(long s)`	Враћа нову инстанцу времена која је повећана за број секунди.
`now()`	Статичка метода која враћа тренутно време.
`toNanoOfDay()`	Враћа број наносекунди протеклих од почетка дана.
`toSecondOfDay()`	Враћа број секунди протеклих од почетка дана.
`withMinute(int m)`	Враћа нову инстанцу датума у којој је минут замењен прослеђеним. Подржано и за друге мерне јединице.

Класа `LocalDateTime` представља датум и време у ISO-8601 формату (yyyy-MM-ddTHH:mm:ss) без придружене временске зоне. Види се да ова класа обједињује функционалности претходне две класе па се може закључити какав је скуп њених функционалности.

Када је у питању форматирање датума, класа `DateTimeFormatter` може бити од користи. Ова класа пружа могућност коришћења већ постојећих (стандардних) формата.

```
java.time.LocalDateTime datum = java.time.LocalDateTime.of(2021, 11, 30, 15, 31);
String isoDatumTekst = datum.format(DateTimeFormatter.ISO_DATE_TIME);
System.out.println(isoDatumTekst); // 2021-11-30T15:31:00
```

Такође је могуће и форматирање у складу са специфичним потребама програма.

```
java.time.LocalDateTime datum = java.time.LocalDateTime.of(2021, 11, 30, 15, 31);
java.time.format.DateTimeFormatter specificniFormat=
java.time.format.DateTimeFormatter.ofPattern("dd.MM.yyyy|HH:mm:ss");
String specificniDatumTekst = datum.format(specificniFormat);
System.out.println(specificniDatumTekst); //30.11.2021|15:31:00
```

Пример 13. Написати Јава програм који приказује сва датум-времена до минутне прецизности за које важи да је сума редног броја дана у месецу, месеца у години, сата (24 сатни формат) и минута једнака броју који корисник уноси на улазу. Притом узети само датум-времена из текуће године. Користити формат исписа по жељи. □

```
public class FiltrirajVremena {
    public static void main(String[] args) {
        java.util.Scanner skener = new java.util.Scanner(System.in);
        System.out.println("Унесите број");
        int n = skener.nextInt();
        java.time.LocalDateTime sad = java.time.LocalDateTime.now();
        int tekucaGodina = sad.getYear();
        java.time.LocalDateTime vreme
              = java.time.LocalDateTime.of(tekucaGodina, 1, 1, 0, 0);
        while (vreme.getYear() == tekucaGodina) {
               int suma = vreme.getMonthValue() + vreme.getDayOfMonth()
```

```
                                + vreme.getHour() + vreme.getMinute();
                if (suma == n)
                        System.out.println(vreme);
                vreme = vreme.plusMinutes(1);
        }
        skener.close();
    }
}
```

Следи пример једног извршавања програма.

```
Унесите број
123
2021-10-31T23:59
2021-11-30T23:59
2021-12-29T23:59
2021-12-30T22:59
2021-12-30T23:58
2021-12-31T21:59
2021-12-31T22:58
2021-12-31T23:57 ■
```

6.7. Рад са псеудослучајним бројевима, инстанцама класе Random

У секцији 6.5 демонстрирана је употреба статичког метода `Math.random()` који генерише псеудослучајан број из интервала [0, 1]. Поред овог механизма Јава пружа и неке додатне могућности за рад са псеудослучајним бројевима кроз класу `Random`. Ова класа проширује могућности метода `Math.random()` у смеру рада са специфичним типовима псеудослучајних података, на пример, генерисање целих псеудослучајних бројева, псеудослучајних логичких вредности итд. Креирање инстанце ове класе могуће извршити на два начина.

```
java.util.Random gen1 = new java.util.Random();
java.util.Random gen2 = new java.util.Random();
java.util.Random gen3 = new java.util.Random(2131);
java.util.Random gen4 = new java.util.Random(2131);
System.out.println(gen1.nextInt()); // -1610654896 (вероватно другачије сваки пут)
System.out.println(gen2.nextInt()); // 1462713902 (вероватно другачије сваки пут)
System.out.println(gen3.nextInt()); // -385602027
System.out.println(gen4.nextInt()); // -385602027
```

Први и други конструктор у горњем коду немају параметре, што утиче да се тзв. „семе“ генератора (енг. random seed) подешава имплицитно. То даље значи да ће и сама секвенца псеудослучајних бројева, која се генерише помоћу ових генератора, бити вероватно другачија (упоредити прве вредности које враћају `gen1` и `gen2`). Уколико се користи конструктор који прихвата параметар, односно “семе”, секвенце псеудослучајних бројева генерисаних од стране два или више генератора биће идентичне (погледати следећи кôд).

```
java.util.Random gen1 = new java.util.Random(12345);
java.util.Random gen2 = new java.util.Random(12345);
System.out.println(gen1.nextDouble()); // 0.3618031071604718
System.out.println(gen2.nextDouble()); // 0.3618031071604718
```

```
System.out.println(gen1.nextInt()); // -287790814
System.out.println(gen2.nextInt()); // -287790814
System.out.println(gen1.nextInt()); // -355989640
System.out.println(gen2.nextInt()); // -355989640
```

Штавише, ако се на различитим рачунарима и оперативним системима инсталира Јава виртуелна машина и покрену над њима програми који користе генераторе са истим семеном, опет ће се добијати идентичне вредности.
Преглед неких чешће коришћених метода у оквиру класе `Random` је дат испод.

Метода	**Опис**
`nextBoolean()`	Униформна псеудослучајна логичка вредност.
`nextBytes(byte[] bajtovi)`	Уписује униформне псеудослучајне бајтове у прослеђени низ.
`nextDouble()`	Униформни псеудослучајни број из [0, 1] у двострукој прецизности.
`nextFloat()`	Униформни псеудослучајни број из [0, 1] у једнострукој прецизности.
`nextGaussian()`	Псеудослучајни број из нормалне расподеле са очекивањем 0 и стандардном девијацијом 1.
`nextInt()`	Униформни псеудослучјани цео број из целог допустивог опсега.
`nextInt(int maks)`	Униформни псеудослучајан цео број из опсега 0 до `maks` (без `maks`).
`nextLong()`	Униформни псеудослучајан дугачки цео број из допустивог опсега.
`setSeed(long seme)`	Ажурира „семе“ генератора.

Пример 14. Написати Јава програм који рачуна просек вредности које враћа метод `nextGaussian()`. На улазу корисник уписује колико бројева жели да креира, а на излазу се исписује просек. Семе генератора подесити на неку фиксирану вредност. □

```
public class TestirajPseudoNormalneBrojeve {
    public static void main(String[] args) {
        java.util.Scanner skener = new java.util.Scanner(System.in);
        System.out.println("Унесите број бројева ");
        int n = skener.nextInt();

        java.util.Random gen = new java.util.Random(12345);
        double prosek=0;
        for(int i=0; i<n; i++) {
               double sp = gen.nextGaussian();
               prosek+=sp;
        }
        prosek/=n;
        System.out.println("Просек је "+prosek);

        skener.close();
    }
}
```

Следи резултат рада програма за неколико уноса. Може се наслутити да просек заиста тежи ка 0.

```
Унесите број бројева
3
Просек је 0.4498106986322034
```

```
------------------------------
Унесите број бројева
10
Просек је 0.009936086679199475
------------------------------
Унесите број бројева
100
Просек је 0.01648838069826526
------------------------------
Унесите број бројева
100000
Просек је 6.269351781967599E-4
------------------------------
Унесите број бројева
100000000
Просек је 1.0518669106605583E-4■
```

Пример 15. Написати Јава програм који генерише 10 предлога за добру шифру (напомена: у пракси добре шифре имају још сложенију структуру). Добра шифра треба да се састоји од малих или великих слова алфабета и цифара. Притом, слова у просеку треба да чине око 70% карактера, а цифре око 30%. Величина добре шифре је између 8 и 16 карактера. Семе генератора подесити на неку фиксирану вредност. □

```
public class GenerisiPredlogeSifara {
    public static void main(String[] args) {
        java.util.Random gen = new java.util.Random(42121);
        for (int i = 0; i < 10; i++) {
                StringBuilder sbSifra = new StringBuilder();
                int duzina = gen.nextInt(9) + 8; // 8-16
                for (int j = 0; j < duzina; j++) {
                        double tipSp = gen.nextDouble();
                        if (tipSp <= 0.7) {
                                int udaljenost = gen.nextInt(26);
                                char c = (char) ('a' + udaljenost);
                                if (gen.nextBoolean())
                                        c = Character.toUpperCase(c);
                                sbSifra.append(c);
                        } else {
                                sbSifra.append(gen.nextInt(10)); // 0-9
                        }
                }
                System.out.println(sbSifra);
        }
    }
}
```

Неки бољи предлагач шифри би укључивао и употребу слогова због лакоће памћења. Такође би такав предлагач требало да избегава употребу фиксираног семена генератора, јер би то представљало безбедоносни проблем. Овде је то урађено зарад омогућавања репродуковања резултата извршавања.
Следи резултат рада програма за неколико уноса.

```
9bs2v7ZSwLryme
thgTXRtpIJ4k9A4
82pi9u99yJg
F0UA9pLas1a
At387LYEo2u7Z6
```

```
3iOIm5M3
eMDW76TRHVEJ
PgMPS3L167
totYcB7gLZaS
98AKm6gCescY ■
```

6.8. Резиме

У овом поглављу је демонстриран начин употребе Јава функционалности опште намене. За разлику од програмског језика C, ове функционалности су једноставније за коришћење, интуитивније и обично захтевају мањи труд програмера (мање написаних линија кода). Детаљнији приказ ових функционалности, односно начин њихове реализације, у овом моменту није могућ, јер је за то потребно боље разумевање основних и напредних Јава и ООП концепата. Стога ће неке од њих бити детаљније разматране у наредним поглављима – засад је само разумевање начина њихове употребе довољно.

6.9. Питања и задаци

1. Шта је садржано у класи `System` и за шта се користи та класа?
2. Упоредити `System.out` и `System.err`. По чему су слични, по чему се разликују?
3. Како се врши мерење времена извршења неког дела програма или читавог програма написаног у програмском језику Јава? Илустровати примером, који се разликује од примера датог у тексту.
4. Да ли и како програмер у Јави може уклањати податке из меморије самостално? Упоредити то са начином уклањања података из меморије у програмском језику C.
5. Да ли у програмском језику Јава постоји механизам прекида програма попут функције `exit`, која постоји у програмском језику C? Ако постоји, објаснити како се користи тај механизам и илустровати примером.
6. Колико траје животни век објекта креираног у програмском језику Јава? Упоредити оператор `new` из порограмског језика Јава и функцију `malloc` из програмског језика C.
7. Објаснити како оператор `==` пореди променљиве примитивног типа, а како инстанцне променљиве. Илустровати примером.
8. Чему служи оператор `instanceof`, како се користи и каква је веза овог оператора и принципа наслеђивања у Јави? Илустровати примером.
9. Да ли је следећи део кода исправан?

```
String test = "Данас је понедељак";
for(int i=0;i<test.length();i+=2)
      test.charAt(i) = '#';
System.out.println(test);
```

   Образложити одговор. Шта је идеја датог кода?

11. Написати програм који испитује да ли је ниска, која се уноси са тастатуре, палиндром. За ниску се каже да је палиндром ако се исто чита с лева на десно и с десна на лево. На пример, „АНА“, „123-321“ и слично.

12. Примером илустровати и објаснити разлике између метода `equals()` и `equalsIgnoreCase()`.
13. Истражити класу `StringBuilder` и упоредити је са класом `String`. Илустровати примером.
14. Шта су омотачи примитивних типова и за шта се користе? Примером илустровати основне карактеристике омотача типова. Како је функционисало аутоматско обмотавање и одмотавање пре Јаве 5?
15. Написати Јава програм који из текста (ниске) извлачи информације о броју учесника, просечном броју освојених поена, минималном броју освојених поена и максималном броју освојених поена, ако је познато да се текст састоји од реченица где свака има структуирану форму „[Име учесника] [Број поена]“.
16. Написати Јава програм који рачуна колико пута се мења знак целих бројева унетих са стандардног улаза. Бројеви се уносе док се не унесе нула. На пример, за унос 2, 11, -369, 104, 105, 22, -25, -36, -78, -10, 14, 0 резултат је 4 промене знака.
17. Написати Јава програм који за две тачке из простора R^2, чије се координате уносе са тастатуре, одређује међусобно растојање.
18. Написати Јава програм који најпре захтева од корисника да унесе дан, месец и годину за два датума, а потом на излазу исписује да ли први унесени датум претходи другом унесеном датуму.
19. Написати Јава програм који најпре захтева од корисника да унесе сат, минуте и секунде за два временска тренутка, а потом на излазу исписује апсолутну временску разлику, изражену у секундама, између унесених тренутака.
20. Коцкица за игру „Човече не љути се“ баца се 10000 пута, односно 10000 пута се генерише цели број из интервала [1, 6]. Одредити која страна коцкице је „пала“ највише, а која најмање пута. Да ли је ситуација иста и у новом покретању програма? Образложити.

7. Низови у Јави

Низ је један од појмова који се јавља у великом броју програмских језика. Најчешће се дефинише као група променљивих истог типа које се појављују под заједничким именом. У ову општу дефиницију уклапа се и дефиниција низа у програмском језику Јава. За решавање појединих проблема неопходно је коришћење низова или сличних структурних типова података, а велики број задатака се елегантније решава коришћењем низова. Низовни тип података у Јави има следећа својства:

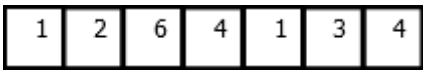
- садржи линеарно уређен, унапред познат, број чланова;
- сви чланови су истог типа и имају заједничко име;
- чланови могу бити примитивног или објектног типа;
- сваком члану приступа се помоћу заједничког имена низа и индекса члана;
- сви индекси су целобројног типа;
- сви чланови низа се третирају као посебне променљиве (називају се и индексним променљивама).

Низовни тип у Јави је увек објектни тип, тј. низ је увек објекат. Ако низ има k индекса (k=1, 2, …), назива се k-димензионални низ. Тако можемо говорити о једнодимензионалним, дводимензионалним, тродимензионалним итд. низовима. Чланови k-димензионалног низа су индексне променљиве са k индекса.

Када је у питању једнодимензионални низ, ту разликујемо две могућности:

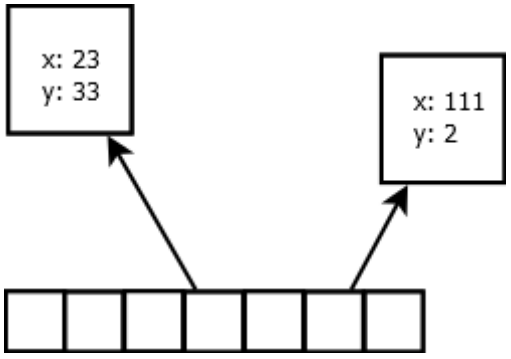
1. једнодимензионални низ примитивних вредности,
2. једнодимензионални низ објеката.

Пример једнодимензионалног низа примитивних вредности од 7 чланова је дат испод.



Дакле, на позицији i се налази баш запис i-те вредности. На пример, на позицији 2 налази се број 6, односно конкретни битови 000...00110.

Са друге стране, једнодимензионални низ објеката има нешто другачију меморијску организацију. У следећем примеру елементи низа су тачке у дводименизионом простору представљене координатама. У овом случају објекти (дводимензионалне тачке) уопште нису на узастопним меморијским локацијама — на узастопним меморијским локацијама су само референце ка њима.



Низови објектних типова у Јави нису исто што и низови структура података у C програмском језику. Наиме, показивачка аритметика језика C, у случају када се ради са низом структура, конципирана је тако да је позната величина структуре (`sizeof` оператор) и усклађена је са истом. То значи да се приликом рачунања позиције i-тог елемента растојање рачуна тако што се на адресу почетка низа дода умножак броја i и величине структуре података. У Јави је величина референце увек иста и износи 4 бајта,

без обзира на величину објекта на који се реферише. Ово је последица раније описаног својства да објекти, на које показују низовне референце, могу бити било где у меморији. Низови објеката у Јави се стога могу пре поистоветити са низовима показивача ка структурама података уместо са низовима структура података у C-у.
Када је у питању временска сложеност приступа елементу низа, једнодимензионални низови примитивних вредности, али и једнодимензионални низови објеката имају O(1) сложеност приступа. Без обзира што је број ефективних рачунских операција код приступа низу објеката нешто већи (рачуна се позиција референце па се потом приступа ономе на шта референца показује), ово не утиче на сложеност изражену у O() нотацији.

7.1. Декларација и иницијализација низа

Низовна променљива се декларише у делу програма за декларацију. (Тада се могу задати и вредности члановима низа.) Број парова угластих заграда у декларацији одређује колико ће индекса имати низ, тј. његову димензионалност. Постоје два начина за декларисање низова – експлицитни и имплицитни.

```
// експлицитни
int pozicija[], a[];
String niska[][], bb[][];
Knjiga naslov[][][];

// имплицитни
int [] pozicija, a;
String [][] niska, bb;
Knjiga [][][] naslov;
```

Код експлицитног начина се угласте заграде задају после назива низовне променљиве. Код имплицитног угласте заграде претходе називима низовних променљивих.
Уочава се разлика између ове две врсте декларација — друга варијанта омогућава декларисање више низова, без потребе за поновним навођењем угластих заграда.

```
int [] a, b, zzz; // све 3 променљиве су низовне
int a, b, zzz[];  /* променљиве су подразумевано цели бројеви осим ако се не дода
заграда, у том случају су низови целих бројева */
```

Креирање низа се може реализовати на два начина:

1. без иницијализације — помоћу оператора `new`,
2. са иницијализацијом — набрајањем чланова.

Ако се декларација врши на први начин, у декларацији се може задати и број елемената низа (по одговарајућој димензији)[6]. Други начин подразумева да су елементи низа познати већ у фази писања/превођења програма, што је у аналогији са статичким низовима у C-у. Разлика у односу на C је у томе што се и иницијализовани низ налази на хипу, попут неицијализованог низа. Задавањем вредности члановима низа на почетку одређена је и величина меморијског простора коју ће низ заузимати.

```
java.util.Scanner scan = new java.util.Scanner(System.in);
int n = scan.nextInt();
scan.close();
String[] niske = new String[n];// први начин - број елемената познат при извршавању
```

[6] Разликовати појмове број елемената (величина) низа од појма његове димензије. Када се прича о броју елемената, може се говорити о укупном броју елемената или о броју елемената по циљној димензији.

```
int [] brojac = new int[100]; // први начин - број елемената познат при превођењу

// други начин
String godDoba[], s="лето";
godDoba ={"пролеће", s, "јесен", "зима"};
```

Из датог примера се може видети да код неицијализованих низова такође имамо два случаја:

1. број елемената низа је одређен током извршавања програма,
2. број елемената низа је познат већ током превођења.

У оба случаја је реч о низовима који су распоређени у хип меморији.
Бекусовом нотацијом исказано, декларација и иницијализација низова може бити представљена следећим металингвистичким формулама:

```
<декларација и иницијализација низа> ::= <тип>{[]} <листа декл/иниц низа>;
<листа декл/иниц низа> ::= <декл/иниц низа>{,<декл/иниц низа>}
<декл/иниц низа> ::= <назив низа>{[]}[=(<алокација низа>|{<листа елемената>})]
<назив низа> ::= <идентификатор>
<алокација низа> ::= new <тип>[[<димензија>]]{[[<димензија>]]}
<димензија> ::= <аритметички израз>
<листа елемената> ::= [<елеменат>{,[<елеменат>}]
<елеменат> ::= <израз>|{<листа елемената>}
```

7.2. Низовна променљива и индексна променљива

Приступ члановима низа је омогућен посредством низовне променљиве `imeNiza` и индекса `indeks` у форми `imeNiza[indeks]` када је реч о једнодимензионалном низу. Конструкт `imeNiza[indeks]` назива се индексна променљива.
Индекс може бити целобројни литерал или израз који производи целобројни литерал. Минимална вредност индекса је 0, а максимална је једнака броју елемената низа умањеном за 1.

```
int [] brojac = new int[100];
String [] godDoba={"пролеће", "лето", "јесен", "зима"};
...
brojac[50]=2;
a=brojac[i+j]; // i и j могу бити познати у фази превођења или у фази извршавања
x = godDoba[3];
```

Са индексним променљивима оперише се на исти начин као и са променљивима без индекса. Вредности индексних променљивих се мењају помоћу наредбе додељивања.
Индексна променљива је Бекусовом нотацијом дефинисана на следећи начин:

```
<индексна променљива> ::= <назив низа>[<индекс>]{[<индекс>]}
<индекс> ::= <аритметички израз>
```

Пример 1. Написати Јава програм који одређује НЗД низа позитивних целих бројева.□
Функција за одређивање НЗД два броја је иста као у решењу 2 из секције која се односи на структурно програмирање у секцији 3.1.1.

```
public class PokreniNzdNiza {
	public static void main(String[] argumenti) {
		int[] niz = { 24, 48, 96, 192, 36, 72, 144 };
		System.out.print("Низ: ");
		for (int elemenat : niz)
```

```
                System.out.print(elemenat + " ");
            System.out.println();

            int nzd = niz[0];
            for (int i=0; i<niz.length; i++) {
                int elemenat = niz[i];
                nzd = StrukturnoNzd.nzd2(nzd, elemenat);
            }
            System.out.print("НЗД низа: " + nzd);
        }
}
```

За приступ сваком члану низа искоришћена је `for` наредба при чему је за ограничавање бројачке променљиве узет број елемената низа (атрибут `length`). Следи резултат рада овог програма.

```
Низ: 24 48 96 192 36 72 144
НЗД низа: 12■
```

7.3. Низови и циклуси

У секцији 5.5.6 приказан је класичан начин употребе `for` наредбе циклуса. Будући да се `for` користи често у комбинацији са низовима, постоји и скраћена нотација (колекцијска `for` наредба) која омогућава избегавање употребе бројачке променљиве. Следећи фрагмент кода приказује пролазак (итерирање) кроз целобројни низ.

```
for (int elemenat : niz)
     System.out.println(elemenat);
```

Синтакса колекцијске `for` наредбе је, према Бекусовој нотацији, дата следећим формулама:

```
<колекцијска наредба for> ::=
                          for(<тип елемента> <променљива>:<назив низа>)<наредба>
<тип елемента> ::= <тип>
<променљива> ::= <идентификатор>
```

Колекцијска наредба `for` се у позадини преводи у тзв. итератор над којим се позивају придружени методи. Више детаља о итераторима ће бити дато у секцији 14.2.

Пример 2. Написати Јава програм који одређује НЗД низа позитивних целих бројева. У реализацији користити колекцијску `for` наредбу. □

Функција за одређивање НЗД два броја је иста као у претходном примеру.

```
public class PokreniNzdNizaKolekcijskiFor {
    public static void main(String[] argumenti) {
        int[] niz = { 24, 48, 96, 192, 36, 72, 144 };
        System.out.print("Низ: ");
        for (int elemenat : niz)
            System.out.print(elemenat + " ");
        System.out.println();

        int nzd = niz[0];
        for (int elemenat : niz)
            nzd = StrukturnoNzd.nzd2(nzd, elemenat);
        System.out.print("НЗД низа: " + nzd);
    }
```

```
}
```

Извршењем овог програма добија се исти разултат као при извршавању претходног програма. ■

Пример 3. Написати Јава програм који омогућује да се оформи низ са задатим бројем елемената (који уноси корисник са стандардног улаза), тако да сваки члан низа добије исту (задату) вредност и да се применом колекцијског `for` циклуса прикажу вредности свих чланова низа. □

```
public class RadSaNizom {
    public static void main(String[] args) {
        java.util.Scanner skener = new java.util.Scanner(System.in);
        System.out.println("Унесите број елемената низа: ");
        int n = skener.nextInt();
        skener.close();
        if (n <= 0) {
               System.err.println("Некоректан број елемената.");
               System.exit(1);
        }

        double[] niz = new double[n];
        double x = -23.34e1;
        for (int i = 0; i < niz.length; i++)
               niz[i] = x;
        for (double d : niz)
               System.out.printf("%8.2f ", d);
        System.out.println();
    }
}
```

Лако се може уочити да се низ креира током извршења програма, а не током превођења или на самом почетку извршавања. Стога ће програм коректно радити и када корисник унесе јако велику вредност за број елемената низа. Следи пример извршавања програма.

```
Унесите број елемената низа:
5
 -233.40  -233.40  -233.40  -233.40  -233.40 ■
```

Пример 4. Написати Јава програм који пребројава елементе у низу бројева из интервала 1-10 и приказује их у бројчаном облику и у облику хистограма.□

У првом случају, програм је реализован монолитно – као јединствени метод.

```
public class FrekvencijeBrojevaMonolitno {
    public static void main(String[] args) {
        int[] rezultati ={7, 3, 4, 9, 7, 6, 3, 10, 5, 6, 4, 3, 3, 3, 2, 5, 7, 9, 1};
        int granica = 10;
        int[] brojPojava = new int[granica];

        for (int i = 0; i < brojPojava.length; i++)
               brojPojava[i] = 0;
        for (int x : rezultati)
               brojPojava[(x - 1)]++;

        for (int i = 0; i < brojPojava.length; i++)
               System.out.printf("%d:%d %s", (i + 1), brojPojava[i],
                              ((i + 1) % 8 == 0) ? "\n" : "\t");
```

```
        System.out.printf("\n\n");

        for (int i = 0; i < brojPojava.length; i++) {
               System.out.printf("%3d:", i + 1);
               for (int j = 0; j < brojPojava[i]; j++)
                      System.out.print("*");
               System.out.println();
        }
    }
}
```

Као што се може видети, низ `brojPojava` чува вредности броја појава за сваки од бројева из интервала. Стога је тип компоненте низа целобројни тип, а број елемената низа је задат дужином интервала `granica`.

Најпре се врши пребројавање, тј. попуњавање низа `brojPojava`, након тога се врши бројчани приказ хистограма и на крају графички приказ.

Извршавањем овог програма добија се следећи резултат.

```
1:1    2:1    3:4    4:3    5:2    6:2    7:3    8:0
9:2    10:1

 1:*
 2:*
 3:****
 4:***
 5:**
 6:**
 7:***
 8:
 9:**
10:*
```

Алтернативно решење је да се поједине функционалности монолитног метода (израчунавање, графички приказ и нумерички приказ) реализују као одвојени методи. С обзиром да је метод `main()`, из којег се позивају ови методи, статички, то и новонаправљени методи морају бити означени кључном речју `static`. Надаље, низ који представља број појава елемената и низ са елементима за пребројавање се новонаправљеним методима прослеђују као параметри/аргументи.

```
public class FrekvencijeBrojevaRazdvojeno {
    static void izracunajFrekfencije(int[] rezultati, int[] frekfencije) {
        for (int i = 0; i < frekfencije.length; i++)
               frekfencije[i] = 0;
        for (int x : rezultati)
               frekfencije[(x - 1)]++;
    }

    static void prikaziFrekfencijeNumericki(int[] frekfencije) {
        for (int i = 0; i < frekfencije.length; i++)
               System.out.printf("%d:%d %s", (i + 1), frekfencije[i],
                              ((i + 1) % 8 == 0) ? "\n" : "\t");
    }

    static void prikaziFrekfencijeGraficki(int[] frekfencije) {
        for (int i = 0; i < frekfencije.length; i++) {
               System.out.printf("%3d:", i + 1);
```

```
                for (int j = 0; j < frekfencije[i]; j++)
                        System.out.print("*");
                System.out.println();
        }
    }

    public static void main(String[] args) {
        int[] rezultati = {7, 3, 4, 9, 7, 6, 3, 10, 5, 6, 4, 3, 3, 3, 2, 5, 7, 9,1};
        int granica = 10;
        int[] brojPojava = new int[granica];
        izracunajFrekfencije(rezultati, brojPojava);
        System.out.println();
        prikaziFrekfencijeNumericki(brojPojava);
        System.out.printf("\n\n");
        prikaziFrekfencijeGraficki(brojPojava);
    }
}
```

Могуће је и другачије решење — у којем методи неће преко аргумената добијати број појава елемената, већ ће променљиве, које чувају елементе који се пребројавају и број појава елемената, бити декларисане тако да су директно доступне сваком од метода. Другим речима, низови `rezultati` и `brojPojava` ће бити декларисани изван тела метода и на тај начин „глобално доступни“, тј. видљиви свим методима у датотеци која садржи Јава кôд. С обзиром да целобројни низови `rezultati` и `brojPojava` треба да буду видљиви за статичке методе `main()`, `izracunajFrekfencije()`, `prikaziFrekfencijeNumericki()` и `prikaziFrekfencijeGraficki()`, то ове две променљиве морају да буду статичке.

Надаље, израчунавање се може осмислити тако да се интервал у којем се налазе бројеви, који се пребројавају, не буде унапред задат, већ да се утврди на основу садржаја низа чији се елементи пребројавају, па да се низ са фреквенцијама алоцира тек по одређивању интервала. На тај начин се постиже додатна флексибилност:

```
public class FrekvencijeBrojevaRazdvojenoGlobalniPodaci {
    static int[] rezultati={7, 3, 4, 9, 7, 6, 3, 10, 5, 6, 4, 3, 3, 3, 2, 5,7,9,1};
    static int[] brojPojava;
    static int min;
    static int max;

    static void izracunajFrekfencije() {
        min = rezultati[0];
        max = rezultati[0];
        for (int x : rezultati)
                if (x > max)
                        max = x;
                else if (x < min)
                        min = x;
        brojPojava = new int[max - min + 1];
        for (int i = 0; i < brojPojava.length; i++)
                brojPojava[i] = 0;
        for (int x : rezultati)
                brojPojava[x - min]++;
    }

    static void prikaziFrekfencijeNumericki() {
        for (int i = 0; i < brojPojava.length; i++)
```

```
                System.out.printf("%d:%d %s", (i + min), brojPojava[i],
                                ((i + 1) % 8 == 0) ? "\n" : "\t");
    }

    static void prikaziFrekfencijeGraficki() {
        for (int i = 0; i < brojPojava.length; i++) {
                System.out.printf("%3d:", i + min);
                for (int j = 0; j < brojPojava[i]; j++)
                        System.out.print("*");
                System.out.println();
        }
    }

    public static void main(String[] args) {
        izracunajFrekfencije();
        System.out.println();
        prikaziFrekfencijeNumericki();
        System.out.printf("\n\n");
        prikaziFrekfencijeGraficki();
    }
}
```

Пример 5. Написати Јава програм који реализује стек помоћу низа. Потом га применити за учитавање секвенце реалних бројева и њихов приказ у обрнутом редоследу.□

Стек је фундаментална структура података коју карактерише тзв. LIFO (енг. Last In First Out) принцип – елеменат који се последњи стави на стек ће, приликом узимања, бити први узет са стека. У неким књигама се оваква структура података још назива и магацин, стог или потисни аутомат. Стек се налази у основи реализације: позива метода, преноса параметара и повратка у позивајући метод код скоро свих данас најпопуларнијих програмских језика: C, Јава, C++, C#, ЈаваСкрипт итд. Основне операције над стеком су гурање/додавање елемента на стек (енг. push), уклањање елемента са стека (енг. pop), те провера да ли је стек празан (у ком случају се не може уклонити елемент са стека) или пун (када нема могућности за додавање елемента на стек).

```
public class RadSaStekom {
    static double[] stek;
    static int vrhSteka;

    static void inicijalizujStek(int dimenzija) {
        stek = new double[dimenzija];
        vrhSteka = -1;
    }

    static void dodaj(double elem) {
        if (vrhSteka == stek.length - 1) {
                System.err.println("Грешка при додавању: стек је пун!");
                return;
        }
        stek[++vrhSteka] = elem;
    }

    static double ukloni() {
        if (vrhSteka == -1) {
                System.err.println("Грешка при уклањању: стек је празан!");
                return -1;
```

```
        }
        return stek[vrhSteka--];
    }

    static int brojElemenata() {
        return (vrhSteka + 1);
    }

    public static void main(String[] args) {
        double[] niz = {13.4, 7.4, 6.3, 3.2, 4, 51, 6.2, 4.7, 3, 14.5, -7.6, 0, 25};
        inicijalizujStek(100);
        int i = 0;
        while (i < niz.length) {
               double x = niz[i];
               dodaj(x);
               i++;
        }
        while (brojElemenata() > 0)
               System.out.print(ukloni() + "\t");
    }
}
```

У овом случају, стек је реализован преко једнодимензионалног низа `stek`. С обзиром да се на стек смештају реални бројеви, то ће се користити једнодимензионални низ реалних бројева. Елементи низа су вредности које су смештене на стек, а посебна променљива `vrhSteka` указује који елеменат је последњи додат на стек.

У примеру је, при реализацији метода `dodaj()` и `ukloni()`, усвојен приступ да се код нерегуларних ситуација (на пример, покушај избацивања са празног стека) прикаже на стандардном излазу порука о грешци и да се повратна вредност метода, у којем је дошло до грешке, постави на неку фиксирану вредност.

Следи илустрација извршавања програма.

```
25.0    0.0    -7.6    14.5    3.0    4.7    6.2    51.0    4.0    3.2    6.3    7.4
        13.4 ■
```

7.4. Аргументи команде линије код улазне тачке програма

Понекад је при покретању програма потребно или пожељно задати разне улазне параметре. Ово се постиже помоћу аргумената команде линије који су у коду наведени као улазни параметри метода `main()` – с обзиром да је метод `main()` улазна тачка програма. Ови параметри су представљени у виду низа и зато их разматрамо у овом поглављу. Задавање аргумената командне линије се постиже путем конзоле при самом позивању програма.

```
java NazivPrograma argument0 argument1 ... argumentN
```

Алтернативно се ово може постићи и путем одговарајућег дијалога у развојном окружењу које програмер користи. Овај начин задавања аргумената има смисла само током развијања и тестирања програма.

Пример 6. Написати Јава програм за сабирање целих бројева који су задати као аргументи командне линије. Претпоставити да су аргументи дати коректно, тј. да ће бити увек цели бројеви. □

Аргументи командне линије су увек представљени као низ ниски. Уколико је програмеру потребно да учита неки други тип података, потребно је да изврши одговарајућу конверзију као што је урађено у следећем програму помоћу метода `Integer.parseInt()`.

```
public class SumirajArgumente {
    public static void main(String[] args) {
        int suma = 0;
        for(String a : args) {
               int broj = Integer.parseInt(a);
               suma+=broj;
        }
        System.out.println("Сума је "+suma);
    }
}
```

Следи илустрација извршавања програма.

```
java SumirajArgumente 14 20 1 -2 13 352
Сума је 398 ■
```

Пример 7. Написати Јава програм који сабира, одузима, множи или дели два броја. Аргументи командне линије су, редом, тип операције (`+`, `-`, `*`, `/`), а потом и два реална броја. □

У програму се најпре проверава да ли је очекивани број аргумената командне линије 3, након чега се, на основу типа операције (који се посматра као ниска — нема потребе за конверзијом у карактер), реализује одговарајућа аритметичка операција.

```
public class OdaberiOperaciju {
    public static void main(String[] args) {
        if (args.length != 3) {
               System.err.println("Очекују се три улазна аргумента.");
               System.exit(1);
        }
        String op = args[0];
        double op1 = Double.parseDouble(args[1]);
        double op2 = Double.parseDouble(args[2]);
        double rez = 0;
        switch (op) {
        case "+":
               rez = op1 + op2;
               break;
        case "-":
               rez = op1 - op2;
               break;
        case "*":
               rez = op1 * op2;
               break;
        case "/":
               if (op2 == 0) {
                      System.err.println("Није дозвољено дељење нулом.");
                      System.exit(1);
               }
               rez = op1 / op2;
               break;
        default:
               System.err.println("Неочекивана операција " + op);
```

```
                System.exit(1);
            }
            System.out.println("(" + op1 + ") " + op + " (" + op2 + ") = " + rez);
        }
    }
```

За разлику од програмског језика C, код Јаве се на нултој позицији аргумената командне линије не записује назив програма, већ прави експлицитно задати аргумент – у овом програму је то кôд операције која се извршава. Илустрација извршавања програма је дата испод.

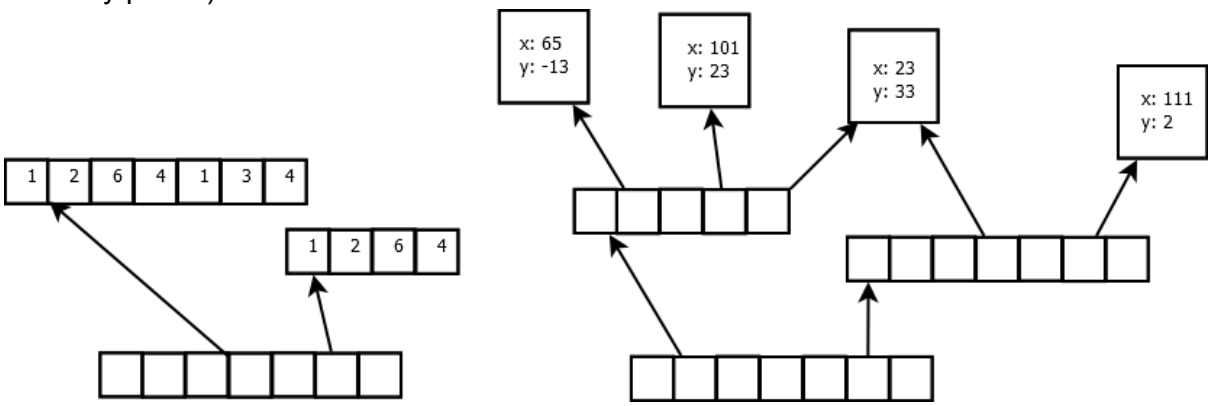
```
java OdaberiOperaciju + 23.2 -2
(23.2) + (-2.0) = 21.2 ■
```

7.5. Вишедимензионални низ

С обзиром да је и сам низ објекат, то значи да референца у оквиру једнодимензионалног низа може показивати на други низ, што представља основ формирања вишедимензионалних низова односно низа низова.
Временска сложеност приступа елементу k-димензионалног низа је иста као и код једнодимензионалног низа, тј. износи O(1). Разлог је то што се k не одређује динамички, већ се фиксира у фази превођења програма, па је у питању константна величина.

7.5.1. Дводимензионални низ

Дводимензионални низ чине елементи истог типа и сваком од њих приступа се помоћу индексне променљиве сачињене од имена низа и два индекса. То значи да је дат низ референци, где свака референца показује на једнодимензиони низ објеката или примитивних вредности. Слика испод представља графички приказ дводимензионалних низова чији чланови су примитивног типа (цели бројеви), односно објекти (координате тачака у равни).



Синтакса креирања дводимензионалног низа је слична као код једнодимензионалног. Користи се `new` да би се најпре креирао „спољашњи“ низ референци ка “унутрашњим” низовима. Иницијална вредност сваке од тих референци је `null`. Након тога је потребно, помоћу `new`, креирати и сваки „унутрашњи“ низ. У случају да имамо дводимензионални низ објеката, потребно је имати још један ниво инстанцирања објеката будући да ће „унутрашњи“ низ у том случају такође бити постављен на `null` референце (погледати претходну слику десно).

```
int n = 10; // n може унети и корисник
```

```
int[][] nizNizova1 = new int[n][];
nizNizova1[4]=new int[20];
nizNizova1[2]=new int[56];
nizNizova1[4][3]=3;
nizNizova1[4][10]=11;

Object[][] nizNizova2 = new Object[10][];
nizNizova2[2]=new Object[n];
nizNizova2[3]=new Object[12];
nizNizova2[2][8]=new Object();
nizNizova2[3][2]=new Object();
```

Претходни фрагмент кода демонстрира рад са дводимензионалним низовима примитивних вредности и објеката. У случају целих бројева је подразумевана вредност за индексну променљиву `nizNizova1[indeks1]` једнака `null`, док је подразумевана вредност за индексну променљиву `nizNizova[indeks1][indeks2]` једнака 0.

Са друге стране, код објектног дводимензионалног низа `nizNizova2` је вредност обе индексне променљиве `nizNizova2[indeks1]` и `nizNizova2[indeks1][indeks2]` једнака `null`.

Пример 8. Написати Јава програм који омогућује да се коришћењем дводимензионалног низа реализују основне операције над матрицама (сабирање, одузимање, множење и израчунавање детерминанте).□

У овом решењу елементи матрице чувају се сложени по редовима као низ једнодимензионалних низова, при чему сваки од једнодимензионалних низова представља један ред матрице. Треба напоменути да матрица представља специјални случај дводимензионалног низа код којег су бројеви елемената у редовима (унутрашњим низовима) једнаки. У општем случају то не мора важити, што је и демонстрирао претходни фрагмент кода.

```
public class RadSaMatricama {
    static void prikazi(double[][] a) {
        if (a == null) {
                System.out.println("Грешка!");
                return;
        }
        System.out.println("Елементи матрице су:");
        for (int i = 0; i < a.length; i++) {
                for (int j = 0; j < a[i].length; j++)
                        System.out.printf("%15.4f", a[i][j]);
                System.out.printf("\n");
        }
    }

    static void prikazi2(double[][] a) {
        if (a == null) {
                System.out.println("Грешка!");
                return;
        }
        System.out.println("Елементи матрице су");
        for (double[] vrsta : a) {
                for (double x : vrsta)
                        System.out.printf("%15.4f", x);
                System.out.printf("\n");
        }
```

```
    }

    static double[][] saberi(double[][] a, double[][] b) {
        if (a.length != b.length)
                return null;
        for (int i = 0; i < a.length; i++)
                if (a[i].length != b[i].length)
                        return null;
        double[][] c = new double[a.length][a[0].length];
        for (int i = 0; i < c.length; i++)
                for (int j = 0; j < c[i].length; j++)
                        c[i][j] = a[i][j] + b[i][j];
        return c;
    }

    static double[][] oduzmi(double[][] a, double[][] b) {
        if (a.length != b.length)
                return null;
        for (int i = 0; i < a.length; i++)
                if (a[i].length != b[i].length)
                        return null;
        double[][] c = new double[a.length][a[0].length];
        for (int i = 0; i < c.length; i++)
                for (int j = 0; j < c[i].length; j++)
                        c[i][j] = a[i][j] - b[i][j];
        return c;
    }

    static double[][] pomnozi(double[][] a, double[][] b) {
        if (a[0].length != b.length)
                return null;
        for (int i = 1; i < a.length; i++)
                if (a[i].length != a[i].length)
                        return null;
        double[][] c = new double[a.length][b[0].length];
        for (int i = 0; i < c.length; i++)
                for (int j = 0; j < c[i].length; j++) {
                        c[i][j] = 0;
                        for (int k = 0; k < a[i].length; k++)
                                c[i][j] += a[i][k] * b[k][j];
                }
        return c;
    }

    static boolean jeKvadratna(double[][] a) {
        for (int i = 0; i < a.length; i++)
                if (a.length != a[i].length)
                        return false;
        return true;
    }

    static double[][] iskljuci(double[][] a, int vrsta, int kolona) {
        int n = a.length;
        double[][] mat = new double[n - 1][n - 1];
        for (int i = 0; i < vrsta; i++)
                for (int j = 0; j < kolona; j++)
```

```
                        mat[i][j] = a[i][j];
        for (int i = vrsta; i < n - 1; i++)
                for (int j = 0; j < kolona; j++)
                        mat[i][j] = a[i + 1][j];
        for (int i = 0; i < vrsta; i++)
                for (int j = kolona; j < n - 1; j++)
                        mat[i][j] = a[i][j + 1];
        for (int i = vrsta; i < n - 1; i++)
                for (int j = kolona; j < n - 1; j++)
                        mat[i][j] = a[i + 1][j + 1];
        return mat;
    }

    static double determinanta(double[][] a) {
        int n = a.length;
        if (n == 1)
                return a[0][0];
        if (n == 2)
                return a[0][0] * a[1][1] - a[1][0] * a[0][1];
        double det = 0;
        double znak = 1;
        for (int j = 0; j < n; j++) {
                det += znak * a[0][j] * determinanta(iskljuci(a, 0, j));
                znak = -znak;
        }
        return det;
    }

    public static void main(String[] args) {
        double[][] a = { { 1.5, 2, 3 }, { 4, 5, 6 } };
        System.out.println("A je: ");
        prikazi(a);
        double[][] b = { { 2, 3, 4 }, { 5, 6.6, 7 } };
        System.out.println("B je: ");
        prikazi(b);
        System.out.println("A+B je: ");
        prikazi2(saberi(a, b));
        System.out.println("A-B je: ");
        prikazi2(oduzmi(a, b));
        System.out.println("A након измене је: ");
        a[1][0] = 0.5;
        prikazi2(a);
        double[][] c = { { 2, 1.5 }, { 3, 0 }, { -1.5, 1 } };
        System.out.println("C je: ");
        prikazi(c);
        System.out.println("A*C je: ");
        double[][] d = pomnozi(a, c);
        prikazi(d);
        if (jeKvadratna(d))
                System.out.println("Детерминанта матрице A*C је: "
                  + determinanta(d));
        else
                System.out.println(
                  "Матрица A*C није квадратна па нема детерминанту.");
    }
}
```

Два метода за приказ матрице у табеларном облику дају исти резултат, само што се у првом случају користи „класични“ `for` циклус, а у другом случају колекцијски `for` циклус. Реализација метода за сабирање, одузимање и множење матрица директно произилази из начина чувања елемената матрице.
Следи пример извршавања програма.

```
A је:
Елементи матрице су:
         1.5000         2.0000         3.0000
         4.0000         5.0000         6.0000
B је:
Елементи матрице су:
         2.0000         3.0000         4.0000
         5.0000         6.6000         7.0000
A+B је:
Елементи матрице су
         3.5000         5.0000         7.0000
         9.0000        11.6000        13.0000
A-B је:
Елементи матрице су
        -0.5000        -1.0000        -1.0000
        -1.0000        -1.6000        -1.0000
A након измене је:
Елементи матрице су
         1.5000         2.0000         3.0000
         0.5000         5.0000         6.0000
C је:
Елементи матрице су:
         2.0000         1.5000
         3.0000         0.0000
        -1.5000         1.0000
A*C је:
Елементи матрице су:
         4.5000         5.2500
         7.0000         6.7500
Детерминанта матрице A*C је: -6.375 ■
```

7.5.2. Тродимензионални низ и низови већих димензија

Када су у питању тродимензионални и низови већих димензија, за њих важе исти концепти и правила као и код дводимензионалних низова. Једина је разлика у додатном нивоу референци, јер унутрашњи низови не реферишу на објекте или примитивне вредности већ на нове низове.

Пример 9. Написати Јава програм који задати текст трансформише у тродимензионални низ карактера такав да се на позицији (i, j, k) налази k-ти карактер j-те речи из i-те реченице. □

```
public class PrebrojKaraktere {
    public static void main(String[] args) {
        String tekst = "Као што се може видети, низ brojPojava чува вредности "
                     + "броја појава за сваки од бројева из интервала. "
                     + "Стога је тип компоненте низа целобројни тип, "
                     + "а број елемената низа је иницијално датом одређен "
                     + "дужином интервала granica."
                     + "Извршавањем овог програма добио би се следећи резултат.";
```

```
        String[] recenice = tekst.split("\\.");
        char[][][] karakteri = new char[recenice.length][][];
        for (int i = 0; i < recenice.length; i++) {
                String[] reci = recenice[i].trim().split(" ");
                karakteri[i] = new char[reci.length][];
                for (int j = 0; j < reci.length; j++) {
                        String rec = reci[j].replace(",", "");
                        karakteri[i][j] = new char[rec.length()];
                        for (int k = 0; k < rec.length(); k++)
                                karakteri[i][j][k] = rec.charAt(k);
                }
        }

        for (int i = 0; i < karakteri.length; i++) {
                System.out.println("--------------------"
                                + System.lineSeparator() + "Реченица " + (i + 1)
                                + System.lineSeparator() + "--------------------");
                for (int j = 0; j < karakteri[i].length; j++) {
                        for (int k = 0; k < karakteri[i][j].length; k++)
                                System.out.print(karakteri[i][j][k]);
                        System.out.println();
                }
        }
    }
}
```

Најпре се улазни текст раздваја на реченице (тачка означава крај реченице). Потом спољна петља пролази кроз низ реченица и сваку од њих раздваја на речи. Свака реч се потом „чисти“ од евентуалних зареза на крају. Након овога се памти карактер у одговарајућој тродимензионалној индексној променљивој, где први индекс означава редни број реченице, други редни број речи у датој реченици, а последњи индекс означава редни број карактера у оквиру дате речи. На крају се врши испис тродимензионалног низа карактера.

Следи пример извршавања програма.

```
--------------------
Реченица 1
--------------------
Као
што
се
може
видети
низ
brojPojava
чува
вредности
броја
појава
за
сваки
од
бројева
из
интервала
```

```
--------------------
Реченица 2
--------------------
Стога
је
...■
```

7.6. Коришћење класе Arrays

За подршку у раду са низовима постоји класа `Arrays`. Неки од популарнијих метода ове класе користе се за сортирање, бинарну претрагу, мешање (енг. shuffle), копирање низа, за попуњавање низа истим елементом и слично.

Сортирање низа

Јава омогућава сортирање низова помоћу метода `Arrays.sort()`. У питању је сортирање учешљавањем (енг. merge sort) које се извршава у времену $O(n \cdot \log(n))$[7]. Кључна предност овог алгоритма у односу на квиксорт (енг. quick sort) је стабилност. За разлику од квиксорт алгоритма, сортирање учешљавањем никад не врши размену једнаких елемената. Наиме, за било која два елемента важи да је један од њих пре у полазној (несортираној) секвенци. Ако су та два елемента према критеријуму сортирања једнака, сортирање учешљавањем никада неће први померити тако да буде после другог у сортираној секвенци.

Бинарна претрага над низом

Бинарна претрага је омогућена методом `Arrays.binarySearch()`. Алгоритам бинарне претраге ради у сложености O(log(n)). Као што је познато, предуслов за примену овог алгоритма је да низ буде сортиран – у супротном алгоритам може да ради некоректно. Метод прихвата следећа четири аргумента: 1) низ; 2) леву границу (позицију) региона претраге; 3) дужину региона претраге; 4) елемент који се тражи. У случају да је пронађен тражени елемент, метод враћа његову позицију у низу. У супротном се враћа негативна вредност његове хипотетичке позиције умањена за 1. Под хипоотетичком позицијом се мисли на позицију на којој би се елемент налазио када би припадао (сортираном) низу. Умањење за 1 се врши како би се избегла неједнозначност при враћању позиције 0, јер у том случају не би било могуће разликовати случај у којем је елемент пронађен и налази се на позицији 0 од случаја у којем је елемент потребно уметнути на позицију -0 (што је исто 0) да би низ остао сортиран.

Пример 10. Написати Јава програм који задати текст трансформише у сортирани низ речи записаних малим словима. Након тога се у петљи уносе речи са стандардног улаза и за сваку унету реч се проверава да ли постоји у претходно уређеном низу речи (применом бинарне претраге). Извршавање се прекида уносом речи „КРАЈ“. □

```
public class SortirajPaTraziRec {
    public static void main(String[] args) {
        String tekst = "Kao što se može videti, niz brojPojava čuva vrednosti "
              + "broja pojava za svaki od brojeva iz intervala. "
```

[7] Више о сортирању учешљавањем се може наћи на адреси https://en.wikipedia.org/wiki/Merge_sort#:~:text=In%20computer%20science%2C%20merge%20sort,in%20the%20input%20and%20output.

```
                + "Стога је тип компоненте низа целобројни тип, "
                + "а број елемената низа је иницијално датом одређен "
                + "дужином интервала granica.";

        String[] reci = tekst.split(" ");
        for(int i=0; i<reci.length; i++) {
                reci[i]=reci[i].toLowerCase().trim().replace(",","").replace(".","");
        }
        java.util.Arrays.sort(reci);
        System.out.println("Сортирани низ речи:"
                        +System.lineSeparator()+"--------------------");
        for(String rec: reci)
                System.out.println(rec);
        System.out.println("--------------------");

        java.util.Scanner skener = new java.util.Scanner(System.in);
        while(true) {
                System.out.println("Унесите реч за претрагу: ");
                String rec = skener.next();
                if(rec.equals("КРАЈ"))
                        break;
                int pozicija = java.util.Arrays.binarySearch(reci, rec);
                if(pozicija>0)
                        System.out.println("Пронађена реч на позицији "+pozicija);
                else
                        System.out.println("Није пронађена реч");
        }
        skener.close();
    }
}
```

Најпре се улазни текст раздваја на речи (празан простор се користи као сепаратор). Потом се свака реч прилагођава тако да у њој фигуришу само мала слова, уклањају се евентуални зарези и тачке. Низ речи се потом сортира применом метода `Arrays.sort()`. Над сортираним низом речи се потом врши претрага тражене речи применом алгоритма бинарне претраге `Arrays.binarySearch()`.

Следи резултат извршавања програма.

```
Сортирани низ речи:
--------------------
brojpojava
granica
а
број
...
низ
низа
низа
од
одређен
појава
сваки
се
стога
тип
тип
```

```
целобројни
чува
што
је
је
--------------------
Унесите реч за претрагу:
од
Пронађена реч на позицији 22
Унесите реч за претрагу:
низ
Пронађена реч на позицији 19
Унесите реч за претрагу:
niz
Није пронађена реч
Унесите реч за претрагу:
КРАЈ ■
```

Пример 11. Написати Јава програм који демонстрира примену алгоритма бинарне претраге над низом целих бројева. Пре примене овог алгоритма потребно је сортирати низ. □

```
public class BinarnaPretraga {
    private static void prikaziCele(int[] celi) {
        for (int x : celi)
               System.out.printf("%d ", x);
        System.out.println();
    }

    public static void main(String[] args) {
        int[] celi = { 23, -7, 5, 6, 4, 34, -23, 8, 17, 0, -2 };
        System.out.println("Пре сортирања");
        prikaziCele(celi);

        java.util.Arrays.sort(celi);
        System.out.println("После сортирања");
        prikaziCele(celi);

        System.out.print("Унесите број који се тражи: ");
        java.util.Scanner skener = new java.util.Scanner(System.in);
        int traziSe = skener.nextInt();
        skener.close();
        int pozicija = java.util.Arrays.binarySearch(celi, 0, celi.length, traziSe);
        if (pozicija >= 0)
               System.out.printf("Тражени број се налази у низу, на позицији %d.",
                              pozicija);
        else {
               System.out.printf("Тражени број %d се не налази у низу.\n", traziSe);
               System.out.printf("Да би се сачувала уређеност низа, "
               + "треба га убацити на (нула базирани) индекс %d.",
                             -(pozicija + 1));
        }
    }
}
```

Решење овог примера је скоро идентично решењу претходног примера. Дакле, најпре се спроводи сортирање за којим следи примена бинарне претраге. Следи илустрација два сценарија извршавања овог програма.

```
Пре сортирања
23 -7 5 6 4 34 -23 8 17 0 -2
После сортирања
-23 -7 -2 0 4 5 6 8 17 23 34
Унесите број који се тражи: 17
Тражени број се налази у низу, на позицији 8.

Пре сортирања
23 -7 5 6 4 34 -23 8 17 0 -2
После сортирања
-23 -7 -2 0 4 5 6 8 17 23 34
Унесите број који се тражи: 9
Тражени број 9 се не налази у низу.
Да би се сачувала уређеност низа, треба га убацити на (нула базирани) индекс 8.■
```

7.7. Методи са аргументима променљиве дужине

Подршка за рад са методима, који имају аргументе променљиве дужине (скраћено varargs), је додата у Јави 5.

Пре тога су алтернативе за постизање сличног ефекта подразумевале:

1. формирање метода са истим називима и различитим бројем аргумената, тзв. преоптерећење метода (описано у секцији 8.4.3) или
2. прослеђивање низа аргумената методу.

Алтернатива 1. је проблематична с обзиром да је већ у фази писања програма потребно предвидети све могуће комбинације аргумената, а то није увек могуће. Обе алтернативе су доводиле до проблема при одржавању кода.

За декларацију метода са аргументом променљиве дужине користе се три тачке у листи аргумената. Пример потписа метода која прихвата аргументе променљиве дужине (нула или више елемената) је следећи:

```
void f(int ... argumenti){ … }
```

Пример 12. Написати Јава програм који користи метод са аргументом променљиве дужине. Метод нема повратну вредност и смисао му је да испише све прослеђене аргументе на стандардни излаз.□

```
public class IspisiArgumente {
    static void ispisiSve(int... argumenti) {
        System.out.println(argumenti.length + " аргумената:");
        for (int a : argumenti)
               System.out.println(a);
    }

    public static void main(String[] args) {
        ispisiSve(20);
        ispisiSve(11, 22, 34, -1);
        ispisiSve();
    }
}
```

Следи резултат рада програма.

```
1 аргумената:
20
4 аргумената:
11
22
34
-1
0 аргумената:
■
```

Приликом компилације се аргумент променљиве дужине преводи у низ. То значи да се са променљивом за аргумент променљиве дужине може радити на исти начин као са низовном променљивом. Предност у односу на приступ у којем програмер самостално креира низ па га проследи методу је мања количина кода, тј. смањивање тзв. понављајућег кода (енг. boilerplate code).

Уколико је потребно креирати метод који поред аргумента променљиве дужине има још неке аргументе, онда се аргумент променљиве дужине мора ставити на крај, као последњи аргумент.

```
void f(String... a, int... b){} // грешка при компилацији
void f(int... a, String b){} // грешка при компилацији
void f(String b, int... a){} // ово је у реду
```

С обзиром да је пре увођења аргумената променљиве дужине већ било написано доста кода (енг. legacy code) коришћењем неке од поменутих могућности, програмери су се у неким случајевима одлучивали да прећу на нову синтаксу, а у неким су задржавали стари приступ. Задржавање старог приступа је потпуно оправдано, посебно у ситуацијама када је редослед аргумената битан. У овом случају би померање аргумента променљиве дужине изазвало проблеме у разумевању кода или би било превише изазовно прилагодити постојећи кôд.

Постоје два принципа за безбедну употребу аргумената променљиве дужине:

1. Аргументи променљиве дужине служе само за читање унутар метода, не и за измену.
2. Не би требало дозволити да референца ка аргументу променљиве дужине „побегне“ из метода у метод која га је позвао, јер то надаље може омогућити индиректно кршење принципа 1.

Поштовање ових принципа онемогућава настанак проблема „прљања“ или „корупције“ хип меморије (енг. heap pollution), који се може јавити при употреби генеричких типова (поглавље 13).

Принцип 1. се заснива на чињеници да се приликом позивања метода не прослеђује именована променљива, већ имплицитна променљива са задатим вредностима. То указује на идеју коју су дизајнери аргумената променљиве дужине имали на уму — употреба у сврху читања, а не и као механизам имплицитних повратних вредности. Стога се доводи у питање смисао измене вредности аргумента променљиве дужине:

```
void f(int... a){
       a[...] = ...
}
```

Наиме, те измене се не преносе у контекст из којег је метод са аргументом променљиве дужине позван. Међутим, постоји начин да се ово заобиђе, што демонстрира наредни пример. (Напомена: следећи кôд представља лош пример употребе аргумената променљиве дужине.)

```
public class IzmenaArgumentaPromenljiveDuzine {
	static int[] f(int ... argumenti) {
		argumenti[0] = 10;
		return argumenti;
	}

	public static void main(String[] args) {
		int[] argumenti = f(1,3,5,2,4);
		for(var a: argumenti)
			System.out.println(a);
		// исписује 10 3 5 2 4
	}
}
```

Дакле, метод `f()` враћа аргумент променљиве дужине као своју повратну вредност. Тиме се промена над аргументом променљиве дужине опажа и ван тела метода `f()`.

7.8. Резиме

Низови су један од фундаменталних рачунарских концепата, па је природно да се они изучавају и у оквиру објектно оријентисаног програмирања. Уз низове се обично изучавају и алгоритми над низовима. Стога се може рећи да је важније алгоритамско разумевање низова од типа парадигме у оквиру којег се са низовима ради. Кључна предност низа је брз случајни приступ, што је и мотивација за програмера да одабере низ као структуру података. У поглављу 14 биће уведене још неке фундаменталне структуре података и описани алгоритми за рад са њима. Тада ће бити могућа нешто детаљнија дискусија о предностима и манама низова, као и упоредни однос низова са другим структурама података, тј. колекцијама и речницима.

7.9. Питања и задаци

1. Која су основна својства низовног типа податка у програмском језику Јава?
2. Примером илустровати проблем који није могуће решити без употребе низова (или неке друге колекције података), односно проблем који није могуће решити ако на располагању имамо само променљиве.
3. Упоредити низове објектних типова у Јави са низовима структура података и низовима показивача ка структурама података у програмском језику C.
4. Објаснити и примером илустровати разлике у декларацији низа са и без иницијализације.
5. Написати Јава програм који за низ целих бројева, који се уноси са тастатуре, на екрану исписује парне бројеве који се налазе на непарним индексима.
6. Написати Јава програм који за низ целих бројева, који се уноси са тастатуре, одређује најмању, највећу, просечну вредност и медијану.
7. Примером 5 дата је илустрација имплементације стека на којем се чувају реални бројеви преко једнодимензионалног низа. Написати Јава програм који илуструје имплементацију реда карактера преко једнодимензионалног низа. Ред је структура података коју карактерише FIFO (енг. First In First Out) принцип.
8. Написати Јава програм који захтева унос два реална броја `a` и `b` са тастатуре, све док се не унесу бројеви такви да је `a<b`, а затим пребројава колико реалних

бројева, који се задају као аргументи командне линије, налази у интервалу `[a,b]`.

9. Упоредити и примером илустровати задавање улазних параметара преко аргумената командне линије у програмском језику Јава и програмском језику C.
10. Написати Јава програм који проверава да ли је матрица дијагонална, горња троугаона или доња троугаона. Матрица се представља дводимензионалним низом.
11. По чему се разликују тродимензионални низови од дводимензионалних низова? Примером илустровати реалну ситуацију када је погодно користити тродимензионалне или четвородимензионалне низове.
12. Истражити који још методи, поред метода приказаних у примеру 10, се налазе у класи `Arrays`. Примером илустровати примену неких од тих метода.
13. Како се пре увођења подршке за рад методима са променљивим бројем аргумената, постизао ефекат који ови методи омогућавају? Шта су предности, а шта недостаци старог и новог приступа?
14. Написати метод која за аргумент узима произвољан број целих бројева и враћа највећи међу њима. Тестирати метод неколико пута задавањем различитог броја целих бројева.
15. Који принципи се требају поштовати да би се постигла безбедна употреба аргумената променљиве дужине? Објаснити и илустровати примерима добру и лошу употребу.
16. Упоредити механизам за рад са методима са променљивим бројем аргумената у програмском језику Јава и програмском језику C.

8. Класе, пакети, поља, методи и објекти у Јави

Као што је истакнуто у секцији 2.3.2, објектно орјентисани приступ код највећег броја популарних програмских језика је реализован уз помоћ класа. Код тих програмских језика (међу којима се налази и Јава) класе, које креира програмер, представљају нове апстрактне типове података.
Креирање апстрактних типова података је фундаментални концепт објектно оријентисаног програмирања. Апстрактни типови података функционишу веома слично као уграђени типови — може се креирати променљива датог типа (која ће референсати на инстанцу дате класе) и преко те променљиве манипуслисати са објектом на који она референсе.

8.1. Класе у Јави

Како у Јави класа описује структуру објекта (податке и понашање), то се свака класа може посматрати као тип података. На овај начин, програмер може дефинисати нови тип који се боље уклапа у проблем који се решава и није принуђен да се приликом решавања проблема ограничи само на већ дефинисане (тзв. предефинисане) типове. Тиме се проширује програмски језик тако што се додају нови типови података који боље одговарају уоченим потребама. Програмско окружење за Јаву омогућује да рад са новим класама/типовима буде једнако удобан и једнако сигуран као што је то случај са предефинсаним класама/типовима.
Гледано са становишта објектно оријентисане парадигме, Јава има следеће карактеристике:

- строго типизиран програмски језик;
- објектно орјентисан програмски језик са хијерархијском структуром класа;
- не допушта да се декларише било шта што није унутар објекта, ово се односи и на атрибуте и на методе;
- програмски кôд је организован по класама и сваки објекат мора бити инстанца неке класе;
- дизајнирана је тако да подржава само једноструко наслеђивање;
- због ефикасности, нису баш сви елементи реализовани као објекти.

Класа се дефинише коришћењем кључне речи `class`. Синтакса за дефиницију класе, претходно дата у секцији 5.3.2 ће сада бити дограђена тако да одсликава чињеницу да тело класе садржи атрибуте и методе.

```
<тип класе> ::= class <назив класе><тело класе>
<назив класе> ::= <идентификатор>
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>)}}
```

Како се буде напредовало приликом усвајања новог знања, тако ће и металингвистичке формуле које описују синтаксне конструкције језика бити проширване и ажуриране, како би могле да укључе новоусвојено знање. Појам `<дефиниција поља>` (о ком ће бити речи у секцији 8.3.1) је сличан дефиницији појма `<декларација и иницијализација променљивих>` из секције 5.4.1, док ће `<дефиниција метода>` бити дата у секцији 8.4.1.

У принципу, једној Јава класи одговара једна датотека са екстензијом `.java`. Притом, назив класе и назив датотеке треба да буду исти.

Следећа класа, обавезно записана у датотеци под називом `Zaposleni.java`, представља запосленог.

```
class Zaposleni {
    String imePrezime;
    double plata;
}
```

У овом случају класа `Zaposleni` садржи само атрибуте који представљају име и презиме запосленог и његову плату. Када се креира класа, онда се може креирати потребан број објеката — примерака дате класе.

8.1.1. Објекат и референца на објекат

Сви објекти у Јави се третирају на униформан начин, коришћењем конзистентне синтаксе. Иако су у Јави све објекти, суштинки се не ради над објектима већ над **референцама** на објекте.
Синтакса за реферисање на инстанцу дате класе је дата следећом металингвистичком формулом:

```
<инстанцна променљива> ::= <идентификатор>
```

Пример 1. Декларисати референце (инстанцне променљиве) за новоуведену класу `Zaposleni`, као и за предефинисану класу `String`. □
У коду који следи се декларишу две референце за класу `Zaposleni` (променљиве `z1` и `z2`) и три референце за класу `String` (променљиве `s1`, `s2` и `s3`).

```
class KreiranjeReferenci {
   public static void main(String[] args) {
       Zaposleni z1;
       Zaposleni z2;

       String s1;
       String s2;
       String s3;
   }
}
```

Међутим, објекти нису креирани већ су само декларисане референце ка њима, без иницијализације. Ово важи и за променљиве класе `Zaposleni` као и за променљиве класе `String`. ▮

8.1.2. Креирање објекта — инстанце дате класе

У реалним ситуацијама, кад год се креира референца, та референцу се повезује са конкретним објектом. Да би се постигло да референца реферише на објекат, неопходно је да се направи (креира) дати објекат. Креирање објекта се релизује помоћу оператора `new`. Кључна реч `new` има следеће значење, „Креирај један нови објекат датог типа/класе и врати референцу на тај новокреирани објекат“.
Наредба за креирање објекта помоћу оператора `new` има следећу синтаксу :

```
<наредба за креирање објката> ::= <израз креирања>;
```

Појам `<израз креирања>` је дефинисан у секцији 5.2.5.

Приликом креирања објекта помоћу оператора `new`, издваја се из хип меморије регион у који ће бити смештене вредности атрибута, тј. поља тог објекта. Информација о адреси тог региона се враћа као резултат примене оператора `new`.

Оператор `new` има доста сличности са функцијом `malloc()` у програмском језику C. У оба случаја је реч о динамичкој алокацији меморије, с тим што `malloc()` дозвољава експлицитно читање повратне вредности, тј. адресе меморијске локације на којој је направљен објекат. Са друге стране, оператор `new` иако налази адресу у меморији и везује је за инстанцну променљиву, програмер не може ту нумеричку вредност читати, нити њоме манипулисати као што је то случају у C-у.

Пример 2. Нека је на располагању већ развијена класа `Zaposleni`. Креирати две инстанце класе `Zaposleni` и креирати три ниске. □

Наредбама које следе се декларишу променљиве `z1` и `z2` и њихове вредности се постављају тако да реферишу на две новокреиране инстанце класе `Zaposleni`, а променљиве `s1`, `s2` и `s3` се постављају тако да реферишу на празну ниску, ниску „Браћа Бамбалић“ и „Браћа Бамбалић“, респективно.

```
class KreiranjeObjekata{
   public static void main(String[] args) {
       Zaposleni z1 = new Zaposleni();
       Zaposleni z2 = new Zaposleni();

       String s1 = new String();
       String s2 = new String("Браћа Бамбалић");
       String s3 = s2;
    }
}
```

Може се приметити да може бити и више променљивих које реферишу на исти објекат — објекат на који реферише `s2` је исти као и објекат на који реферише `s3`. ▮

8.1.3. Објекти класе Object

Као што је већ објашњено у поглављу 2, објекат је кључни појам са којим се повезује низ других значајних појмова објектно оријентисаног програмирања као што је, на пример, наслеђивање. Наслеђивање омогућава креирање нових класа из постојећих уз надограђивање њихове структуре и функционалности. С обзиром да је у Јави подржано једноструко наслеђивање, структура која се добија применом наслеђивања је дрво. Корен овог дрвета је класа под називом `Object` па су све класе директни или индиректни потомци класе `Object`. Ова класа дефинише неколико значајних метода попут `toString()`, `equals()` и `hashCode()`. Њихове дефиниције се могу променити и то је обично пожељно. О њима, као и начину промене њихових дефиниција, биће речи у секцији 8.5.4.

Пример 3. Написати Јава програм који креира објекат класе `Object` и потом га исписује на стандардном излазу. □

```
public class TestirajObject {
    public static void main(String[] args) {
        Object o1;
        o1 = new Object();
        System.out.println(o1);
        System.out.println(o1.toString());
    }
```

```
}
```

Најпре се декларише референца на `Object`, без иницијализације. Потом се креира инстанца класе `Object`. На крају је објекат исписан на конзоли на два начина, први је имплицитним, а други експлицитним позивом метода `toString()`. Може се приметити да је подразумевана текстуална репрезентација објекта класе `Object` поприлично неинформативна. Тај испис је контролисан реализацијом метода `toString()` који је дефинисан на нивоу класе `Object` па је самим тим саставни део и сваке друге класе.

```
java.lang.Object@2f92e0f4
java.lang.Object@2f92e0f4■
```

Пример 4. Написати Јава програм који декларише и инстанцира објекте класе која представља кутију. Кутија се описује са три атрибута: висина, ширина и дубина. □

```
// могло је да пише и
// public class Kutija extends Object{
public class Kutija {
    int visina;
    int sirina;
    int dubina;

    public static void main(String[] args) {
        Kutija kutija1 = new Kutija();
        kutija1.dubina = 10;
        kutija1.sirina = 2;
        kutija1.visina = 10;
        System.out.println(kutija1);
    }
}
```

И у овом случају се, приликом исписа објекта, не добија претерано корисна информација. Било би очекивано да текстуална репрезентација кутије укључује информације о висини, ширини и дубини кутије. Механизам за постизање тог ефекта је превазилажење метода `toString()`. О превазилажењу ће бити речи у секцији 8.5.4.

```
rs.math.oop.g06.p06.kreiranjeKorisnickihObjekata.Kutija@2f92e0f4■
```

8.1.4. Поређењи референци на објекат

Поређење две променљиве применом оператора `==` је увек базирано на истом принципу, а то је регистарско поређење садржаја тих променљивих. У зависности од типа променљивих, зависиће и интерпретација добијеног резултата. Тако, у случају да су променљиве примитивног типа, резултат поређења ће заиста указивати на једнакост садржаја тих променљивих. С друге стране, ако се пореде две инстанцне променљиве, једнакост ће важити само ако указују на исту инстанцу. Уколико ове две променљиве указују на две различите инстанце са истим садржајем, једнакост неће важити. На пример, две кутије потпуно истих димензија, приликом поређења са `==`, неће бити једнаке. Да би се постигао ефекат поређења на једнакост у семантичком смислу (тј. да се две кутије третирају као једнаке ако су истих димензија), потребно је превазићи и користити метод `equals()`. О овоме ће бити речи у секцији 8.5.4.

Пример 5. Написати Јава програм који тестира оператор `==` применом на примитивним целобројним подацима и на инстанцним променљивама класе `Kutija`. □

```
public class UporediPromenljive {
    public static void main(String[] args) {
        int x = 123, y = 123;
        System.out.println(x == y);

        Kutija kutija1 = new Kutija();
        kutija1.dubina = 10;
        kutija1.sirina = 2;
        kutija1.visina = 4;
        Kutija kutija2 = new Kutija();
        kutija2.dubina = 10;
        kutija2.sirina = 2;
        kutija2.visina = 4;
        System.out.println(kutija1 == kutija2);
    }
}
```

Следи очекивани резултат рада програма. Дакле, иако су две кутије суштински једнаке, референце ка њима су различите са становишта оператора `==`, јер реферишу на два меморијски независна објекта.

```
true
false■
```

8.2. Организација класа по пакетима

Програмери групишу сличне, тј. повезане типове, у пакете и на тај начин избегавају конфликте у именима и контролишу приступ.
Пакет је група повезаних типова (класа, интерфејса, енумерисаних типова итд.) за коју се обезбеђује заштита при приступу и управљање простором имена.
На пример, класа `Object` припада пакету `java.lang` из Јава API, класа `Scanner` припада пакету `java.util` итд.
Класе у Јави су организоване по пакетима. Најчешће коришћени пакети су: `java.lang`, `java.util`, `java.io`, `java.net`, `java.awt`, `javax.swing` итд.
Примери из овог уџбеника и решења датих задатака су организована по пакетима ради лакшег сналажења и једноставније организације програмског кода.
Разлози за паковање класа и интерфејса у пакете су:

1. лакше одређивање да ли су типови повезани;
2. лакше се могу пронаћи тражени типови;
3. нема именских конфликта са другим типовима истог назива, јер пакет креира нови простор имена — два типа, на пример, класе могу да имају исто име под условом да припадају различитим пакетима;
4. могућност да типови унутар пакета имају неограничен приступ један другом.

Подаци у оквиру објеката се подразумевано дефинишу тако да могу бити „видљиви“ (тј. доступни) из свих класа које се налазе у истом пакету у којем се налази та класа
Да би могле да се креирају сопствене Јава класе у пакетима, мора се знати где се бајт-код преведених класа налази у оквиру система.
Одређивање места, где Јава окружење за извршавање налази класе, врши се постављањем вредности променљивој `CLASSPATH`. Ова променљива одређује путању до директоријума у којем Јава окружење за извршавање тражи класе. Комбиновањем путање дате помоћу `CLASSPATH` и назива пакета, Јава виртуелна машина проналази

бајт-код класа са којима се оперише. Ако `CLASSPATH` није дефинисан, подразумева се да се класе налазе у поддиректоријуму `lib` директоријума `java` унутар конкретне инсталације Јаве (секција 4.5.2).
Развојна окружења допуштају подешавање путање за класе кроз кориснички интерфејс, у оквиру дефинисања параметара тзв. пројекта.

8.2.1. Дефинисање пакета, наредба package

Процес креирања сопствених пакета се може описати у три корака.

1. Избор имена пакета. Препорука: коришћење назива Интернет домена са елементима поређаним у обрнутом редоследу (на тај начин се постиже да назив пакета буде јединствен). На пример, ако је назив домена `math.rs`, назив пакета би требало да почне са `rs.math`. По конвенцији, називи пакета почињу малим словима.
2. Креирање структуре директоријума (фасцикли, фолдера). Ако је назив пакета из једног дела (нема тачака у називу), назив директоријума поклапа се са називом пакета. Ако се назив пакета састоји из више делова (одвојених тачком), тада за сваки део треба формирати поддиректоријум. На пример, за пакет `rs.math.oop`, главни директоријум треба да се зове `rs`, његов поддиректоријум `math` и у њему треба да постоји директоријум `oop`. У сваки од ових директоријума се могу убацити датотеке, односно класе, интерфејси итд.
3. Постављање `package` наредбе. Ово треба да буде прва наредба Јава програма. На пример, ако је назив пакета `rs.math.oop`, онда на почетку сваке датотеке у том пакету прва наредба мора бити: `package rs.math.oop;`

Бекусовом нотацијом исказано, наредба `package` има следећу синтаксу:

```
<наредба package> ::= package {<назив пакета>.}<назив пакета>;
<назив пакета> ::= <идентификатор>
```

Напомена: интегрисана развојна окружења омогућују да се сва три претходно описана корака реализују једном опцијом, и сама предлажу решење у ситуацијама када неусклађеност путање датотека и назива пакета доведе до грешке.
Сада је јасно да металингвистичка променљива `<назив класе>` дефинисана у секцији 5.3.2 није само идентификатор већ може имати нешто сложенију структуру:

```
<назив класе> ::= {<идентификатор>.}<идентификатор>
<инстанцна променљива> ::= <идентификатор>
```

8.2.2. Увоз класа из пакета, наредба import

Наредба `import` омогућава да се цели пакети или појединачне класе и интерфејси из пакета увезу у циљну класу. Увоз подразумева да кôд свега што се увози постане доступан за коришћење унутар класе/интерфејса у оквиру које је позван увоз — доступност се огледа у реферисању увезене класе/интерфејса путем њиховог назива. У досадашњим програмима није се користила наредба `import` већ се увоз вршио имплицитно задавањем пуног назива сваке класе (који укључује и пуни назив припадајућег пакета). На пример, за креирање инстанце генератора псеудослучајних бројева треба написати:

```
java.util.Random rg = new java.util.Random();
```

У случају да класа/интерфејс користи нешто што се не налази у истом пакету, и то што се користи није увезено наредбом `import` или коришћено уз задавање потпуног назива, компајлер ће пријавити грешку при превођењу.
Следећим кодом је демонстриран увоз свега што је доступно у оквиру пакета `java.util`. Након тога је, на пример, могуће користити класу `Random`, али и класу `Scanner`, без навођења њиховог пуног назива.

```
import java.util.*;
…
Random rg = new Random();
…
Scanner sc = new Scanner(System.in);
```

Увоз целих пакета се не препоручује, јер оптерећује процес компилације. Препорука је да се увози само оно што се заиста користи у оквиру посматране класе или интерфејса. Изузетак од тог правила је употреба тзв. статичког увоза, чији је смисао скраћивање записа приликом позивања статичких метода из неке спољне класе. На пример, ако је потребно интензивно користити математичке функције, доступне већином у облику статичких метода у оквиру класе `Math`, може се извршити статички увоз класе `Math`, након чега ће методи бити позивани без навођења префикса `Math`..

```
import static java.lang.Math.*;
...
cos(a);
sin(b);
...
```

Синтакса наредбе `import` је, дакле, дата следећим металингвистичким формулама:

```
<наредба import> ::= import [static] <назив класе>|{<назив пакета>.}<назив пакета>.*;
<назив пакета> ::= <идентификатор>
```

На пример, да би се увезао генератор псеудослучајних бројева, може се написати:

```
import java.util.Random;
…
Random rg = new Random();
```

Пример 6. Написати Јава програм који генерише 10 псеудослучајних целих бројева. Користити наредбу `import` за увоз класе `Random`. □

```
package rs.math.oop.g08.p06.uvozKlase;

import java.util.Random;

public class IspisiPseudoslucajneCele {
    public static void main(String[] args) {
        Random rg = new Random(12345);
        for(int i=0; i<10; i++)
               System.out.println(rg.nextInt());
    }
}
```

Резултат рада програма је дат испод.

```
1553932502
-2090749135
-287790814
-355989640
```

```
-716867186
161804169
1402202751
535445604
1011567003
151766778 ■
```

8.3. Класе и објекти — поља

Компоненте (чланови) објекта су: променљиве примерка (или поља/атрибути) и методи. У овом поглављу акценат је на пољима објекта.
Свака инстанца дате класе садржи сопствени примерак (сопствену меморијску локацију) за свако од поља које се налази у оквиру објекта. Стога, промена вредности поља датог примерка нема утицаја на друга поља у оквиру датог примерка, нити на исто поље које се налазе у оквиру других примерака.

8.3.1. Дефиниција поља

Дефиниција поља се, засад док још нису уведени модификатори, састоји од две компоненте:

1. типа поља (тип) и
2. имена поља (идентификатор).

```
class PripadajucaKlasa{
       int polje1;
       String polje2;
}
```

Дакле, дефиниција поља је слична декларацији и иницијализацији локалне променљиве. Бекусовом нотацијом се класа може описати на следећи начин:

```
<тип класе> ::= class <назив класе><тело класе>
<назив класе> ::= {<идентификатор>.}<идентификатор>
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>)}}
<дефиниција поља> ::= <декларација и иницијализација променљивих>
```

Дефиниција метода ће бити дата у секцији 8.4.1.
Модификатори омогућују да се подеси видљивост дате променљиве члана – атрибута тј. поља (описано у секцији 8.7), да се одреди да ли се ради о променљивој примерка (инстанцној променљивој) или о класној (статичкој) променљивој (описано у секцији 8.3.3), као и да ли вредност променљиве постаје непроменљива непосредно по креирању (финална тј. константна), што је описано у секцији 8.8.

8.3.2. Приступ пољу у примерку дате класе

Вредности датог поља у оквиру инстанце се може приступити ако се референцира инстанца која садржи ту променљиву. Пољима датог објекта приступа се преко тзв. тачка-нотације.

```
referencaNaObjekat.polje
```

Следећа металингвистичка формула описује како се врши реферисање на поље инстанце тј. примерка одређене класе:

```
<референца на поље инстанце> ::= <инстанцна променљива>.<назив поља>{.<назив поља>}
<назив поља>::= <идентификатор>
```

Претходним формулама је покривен и случај када је неко од поља датог објекта низовног типа. Променљива `<индексна променљива>` је дефинисана у секцији 7.2.

Пример 7. Дефинисати класу `Ucenik` са подацима о имену и презимену и разреду. Креирати објекате ове класе и обезбедити приступ пољима. □

```
package rs.math.oop.g08.p07.pristupPoljima;

public class Ucenik {
    String imePrezime;
    int razred;

    public static void main(String[] args) {
        Ucenik prvi = new Ucenik();
        prvi.imePrezime="Петар Перић";
        prvi.razred=3;
        Ucenik drugi = new Ucenik();
        drugi.imePrezime="Милан Микић";
        drugi.razred=6;
        System.out.println(prvi.razred);
        System.out.println(drugi.imePrezime);
    }
}
```

У наведеном решењу, класа `Ucenik` је искоришћена и за покретање програма јер се у њој се налази `main()` метод. Унутар `main()` метода креирана су два објекта класе `Ucenik`, подешена су интерна стања тих објеката применом тачка-нотације, а касније је примeнтом тачка-нотације омогућено и њихово читање (приступ).

Резултат извршавања је дат испод.

```
3
Милан Микић ■
```

8.3.3. Статичка (класна) поља

Класа садржи само једну копију статичког (класног) поља и то поље је дељено међу свим објектима дате класе. Статичко поље постоји чак иако се не креира ниједан примерак дате класе.

Она припада класи и њу могу сви да референцирају, а не само примерци дате класе.

Статичко поље се декларише коришћењем модификатора `static`.

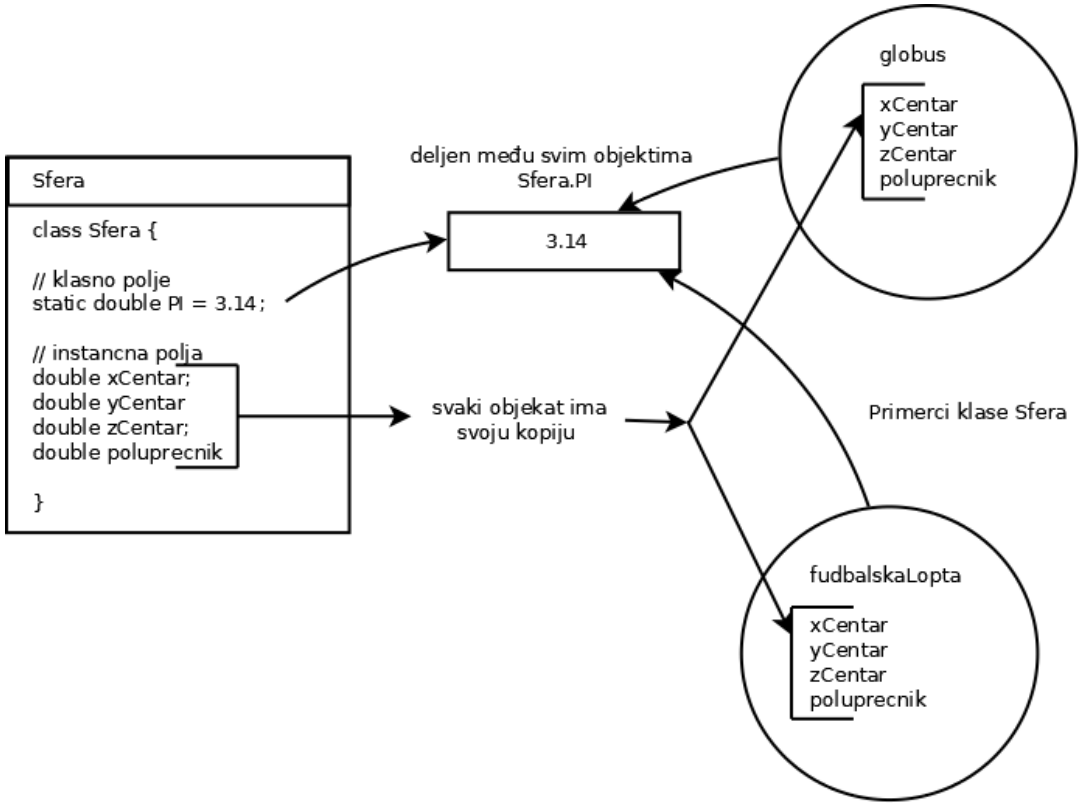
Следећа металингвистичка формула представља проширену дефиницују поља из поглавља 8.3.1 тако да буду укључена и статичка поља:

```
<дефиниција поља> ::= [static ]<декларација и иницијализација променљивих>
```

За приступ статичком пољу се користи тачка-нотација, при чему се као прималац поруке може користити име класе или име неке инстанце класе (препоручује се коришћење имена класе). То се Бекусовом нотацијом може исказати на следећи начин:

```
<референца на статичко поље> ::= <референца на поље инстанце>
                                |<назив класе>.<назив поља>{.<назив поља>}
```

Дијаграм који следи на прегледан начин показује разлику између инстанцних и статичких поља – у начину дефинисања, начину чувања и начину приступа овим пољима.



Статичка и инстанна поља

Пример 8. Класу `Ucenik` из претходног примера проширити статичким пољем које у себи садржи број креираних инстанци класе `Ucenik`. Тестирати промену и читање вредности овог поља. □

```
package rs.math.oop.g08.p08.pristupStatickimPoljima;

public class Ucenik {
    String imePrezime;
    int razred;
    static int brojUcenika = 0;

    public static void main(String[] args) {
        Ucenik prvi = new Ucenik();
       prvi.imePrezime="Петар Перић";
       prvi.razred=3;
       prvi.brojUcenika++; // може се приступити преко инстанцне референце
       Ucenik drugi = new Ucenik();
       drugi.imePrezime="Милан Микић";
       drugi.razred=6;
       Ucenik.brojUcenika++; // али је преко назива класе природније
       System.out.println(prvi.brojUcenika);
    }
}
```

Као што се може видети, поље `brojUcenika` се повећало два пута (испис је 2). То значи да је меморијска локација у којој је записана ова променљива јединствена, што није био случај код инстанцних поља (поља примерка).

```
2 ■
```

8.3.4. Опсег важења за променљиве и поља

Опсег важења променљиве је део програма у којем та променљива може да се користи. Опсег важења локалне променљиве је одређен блоком у којем је она дефинисана. Прецизније, почев од наредбе у којој је променљива декларисана до краја припадајућег блока.

```
{
        int a=1; // овде се може приступити a, али не и b
        { // овде се може приступити a, али не још b
                int b=2; // овде се може приступити и a и b
        } // овде се може приступити a, али не и b!
}
// овде се не може приступити нити a, нити b
```

Статичким пољима се може приступати у сваком тренутку рада програма.
Инстанцним пољима објекта се може приступати све време док постоји тај објекат.
Наредни кôд се надовезује на претходни пример са класом `Ucenik`.

```
public static void main(String[] args) {
        System.out.println(Ucenik.brojUcenika);
        // System.out.println(prvi.imePrezime);
        Ucenik prvi;
        // System.out.println(prvi.imePrezime);
        prvi = new Ucenik();
        prvi.imePrezime="Петар Перић";
        prvi.razred=3;
        System.out.println(prvi.imePrezime);
}
```

Прва коментарисана наредба покушава да користи локалну променљиву која још није декларисана. Друга коментарисана наредба покушава да приступи пољу објекта који још није креиран. Када би се било која од ове две наредбе одкоментарисала, компајлер би пријавио компилациону грешку.

8.4. Класе и објекти — методи

Методи омогућавају да се дугачак кôд разбије у мање целине и на тај начин доприносе прегледности кода. Такође, метод се може позивати већи број пута, што омогућава запис мање количине кода него у случају да методи нису подржани (што је данас веома ретко).

8.4.1. Дефиниција и позив метода

Методи се појављују у телу класе. Метод се састоји из два кључна дела: потписа метода и тело метода.

```
class PripadajucaKlasa{
        ...
      povratni-tip imeMetoda(arg1, arg2, ..., argn){ // потпис метода
              // тело метода
      }
      ...
}
```

Име метода заједно са типом и редоследом параметара чини потпис метода. Потпис два метода у класи мора бити различит како би компајлер могао да одреди који метод се позива.
Дефиниција инстанцног метода се Бекусовом нотацијом може представити на следећи начин:

```
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= <повратни тип> <назив метода>(<параметри>)
<повратни тип> ::= <тип>|void
<назив метода>::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= <тип>
<назив параметра>::= <идентификатор>
<тело метода> ::= <блок>
```

Тело метода може садржати декларације локалних променљивих и друге Јава наредбе које се извршавају при позиву метода. У наредбама у телу метода се могу користити четири потенцијална извора података:

1. формални аргументи метода,
2. инстанцне и класне променљиве,
3. локалне променљиве, дефинисане у телу метода, и
4. вредности које враћају други методи који су позвани у текућем.

Метод враћа вредност преко наредбе `return`. Уколико метод не враћа вредност, његов тип је `void`. Наредба `return` у телу метода омогућује тренутни прекид рада позваног метода, повратак у позивајућу рутину одмах након наредбе позива тог метода, са резултатом рада датим вредношћу израза који следи након наредбе `return`. Ова наредба је Бекусовом нотацијом описана на следећи начин:

```
<наредба return> ::= return [<израз>];
```

Приликом декларације метода у класи, наводи се и тип за сваки од параметара. При позиву метода наводе се стварни аргументи. Синтакса позива метода одређена је на следећи начин:

```
<позив инстанцног метода> ::= <инстанцна променљива>.<назив метода>([<аргументи>])
<аргументи> ::= <аргумент>{,<аргумент>}
<аргумент> ::= <израз>
```

Аргументи метода се наводе у заградама иза имена метода (број аргумената може бити нула). Исто као и код C, параметри и аргументи морају да буду сагласни по броју и по типу.

8.4.2. Позив метода и кључна реч this

Слично као код приступа пољу, инстанцни метод се може позвати коришћењем тачка-нотације тако што се наводи променљива за приступ инстанци, тачка и затим име метода (са одговарајућим аргументима, ако их метод прима).

```
class X{
        void f(){ ... }

        void g(){
                X x = new X();
                x.f();
```

```
        }

        public static void main(String...args){
                X x = new X();
                x.f();
        }
}
```

Када се метод текућег објекта `x` позива унутар другог инстанцног метода објекта `x`, претходно поменута тачка-нотација нема смисла, јер у том контексту не постоји референца `x`. Срећом, постоји специјална кључна реч `this` која управо указује на текући објекат. Тако да се применом тачка нотације у форми `this.nazivMetoda(...)` врши позив метода над текућим објектом. Кључна реч `this` се може и изоставити.

```
class X{
        void f(){ ... }

        void g(){
                ...
                this.f();
                f(); // исти ефекат као и наредба изнад
                ...
        }
}
```

Синтакса за позив метода, која обухавата и позив у оквиру датог примерка класе, описана је следећом металингвистичком формулом:

```
<позив инстанцног метода> ::=
                (<инстанцна променљива>|this).<назив метода>([<аргументи>])
```

Пример 9. Раније дефинисану класу `Ucenik` проширити методом за приказ информација о ученику као и методом којим се, на основу разреда, одређује да ли је ученик у старијој смени (ученици од 5. до 8. разреда су у старијој смени). □

```
package rs.math.oop.g08.p09.pozivanjeMetoda;

public class Ucenik {
    String imePrezime;
    int razred;

    void prikaziInformacije() {
        System.out.println(imePrezime + " " + razred
                        + " " + jeStarijaSmena()); // могло је и this.jeStarijaSmena()
    }

    boolean jeStarijaSmena() {
        return razred > 4;
    }
}
```

```
package rs.math.oop.g08.p09.pozivanjeMetoda;

public class TestirajUcenik {
```

```
    public static void main(String[] args) {
        Ucenik prvi = new Ucenik();
        prvi.imePrezime = "Петар Перић";
        prvi.razred = 3;
        Ucenik drugi = new Ucenik();
        drugi.imePrezime = "Милан Микић";
        drugi.razred = 6;
        prvi.prikaziInformacije();
        drugi.prikaziInformacije();
    }
}
```

У овом примеру је демонстриран начин позивања метода и то у ситуацијама:

1. када је позив направљен унутар другог метода исте класе — метод `jeStarijaSmena()` се позива унутар `prikaziInformacije()` и
2. када метод из којег се позива не припада објекту — позив `prikaziInformacije()` унутар метода `main()`.

У првом случају није искоришћена тачка-нотација, јер је јасно над којим објектом се позива метод. Алтернативно је било могуће у том случају користити и имплицитну референцу на текући објекат `this`.

Следи приказ рада програма.

```
Петар Перић 3 false
Милан Микић 6 true■
```

8.4.3. Преоптерећење метода

Дефинисање метода са истим именом, али различитим аргументима, назива се преоптерећење метода.

У програмском језику C преоптерећивање није дозвољено, што значи да се методи који суштински раде исту ствар, али су другачије параметризовани, обавезно именују различитим називима.

На пример, ако је потребно дефинисати методе који сабирају 2, 3 или 4 броја, у програмском језику C би се, за постизање тог циља, дефинисала 3 метода са различитим именима и различитим списковима аргумената: `saberi2(x, y)`, `saberi3(x, y, z)` и `saberi4(x, y, z, w)`. У Јави би се, применом преоптерећивања, то могло реализовати на следећи начин.

```
double saberi(double x, double y) { return x+y; }

double saberi(double x, double y, double z) { return x+y+z; }

double saberi(double x, double y, double z, double w) { return x+y+z+w; }
```

Овај пример осим што демонстрира могућности Јаве, није од практичног значаја, јер и у Јави и у C-у постоји инфиксни оператор сабирања `+` који омогућава постизање тражене функционалности у краћем запису.

Пример 10. Дефинисати класу `Pravougaonik` описану координатама горњег левог и доњег десног темена. Претпоставка је да су странице правоугаоника паралелне са одговарајућим осама координатног система па је ова репрезентација јединствена. Користити целобројне координате имајући у виду матрицу пиксела на екрану. Омогућити подешавање ових координата на два начина. Разлика између та два начина је у прослеђеним аргументима: 1) прослеђују су се два темена (било која, метод треба да

процени да ли су валидна); 2) прослеђује се само горње лево теме, ширина и висина правоугаоника.☐

```
package rs.math.oop.g08.p10.preopterecivanjeMetoda;

public class Tacka {
    int x;
    int y;
}
```

За потребе реализације програма креирана је и класа `Tacka` задата помоћу две целобројне координате.

```
package rs.math.oop.g08.p10.preopterecivanjeMetoda;

public class Pravougaonik {
    Tacka gl, dd;

    void podesi(int x1, int y1, int x2, int y2) {
        gl = new Tacka();
        dd = new Tacka();
        gl.x = x1 <= x2 ? x1 : x2;
        dd.x = x1 >= x2 ? x1 : x2;
        dd.y = y1 <= y2 ? y1 : y2;
        gl.y = y1 >= y2 ? y1 : y2;
        if(gl.x==dd.x || gl.y==dd.y) {
               System.err.println(
                              "Грешка, висина и ширина морају бити позитивне.");
               System.exit(1);
        }
    }

    void podesi(Tacka gl, int s, int v) {
        if(s<=0 || v<=0) {
               System.err.println(
                              "Грешка, висина и ширина морају бити позитивне.");
               System.exit(1);
        }
        this.gl = gl;
        gl = gl;
        dd = new Tacka();
        dd.x = gl.x + s;
        dd.y = gl.y - v;
    }

    void prikazi() {
        System.out.println("gl=(" + gl.x + ", "
                       + gl.y + ") dd=(" + dd.x + ", " + dd.y + ")");
    }

    public static void main(String[] args) {
        Pravougaonik p1 = new Pravougaonik();
        p1.podesi(10,  20,  50, 30);
        p1.prikazi();
        Pravougaonik p2 = new Pravougaonik();
        Tacka t = new Tacka();
        t.x = 10;
```

```
        t.y = 30;
        p1.podesi(t, 40, 10);
        p1.prikazi();
        Pravougaonik p3 = new Pravougaonik();
        p1.podesi(10, 100, 10, 20);
    }
}
```

Правоугаоници `p1` и `p2` су суштински исти, али се креирају задавањем различитих параметара, што мотивише употребу преоптерећивања.

Овде указујемо на још једну употребу кључне речи `this`. Наиме, у телу метода `podesi(Tacka gl, int s, int v)` `this` је искоришћен како би се избегао именски конфликт између аргумента метода `gl` и истоименог поља класе. Предност у именовању има локална променљива или у овом случају аргумент метода, што значи да је за идентификацију поља потребно користити `this.gl`.

Следе резултати рада програма.

```
gl=(10, 30) dd=(50, 20)
gl=(10, 30) dd=(50, 20)
Грешка, висина и ширина морају бити позитивне.■
```

8.4.4. Статички (класни) методи

Статички методи су налик статичким пољима. Они не припадају инстанци класе већ самој класи. То значи да је и њихов животни циклус независан од постојања конкретне инстанце неке класе — попут функција у програмском језику C.

Са статичким методима смо се већ добро упознали, пошто је већина досадашњих програма реализована помоћу њих. На пример, метод `main(String args[])`, који је неопходан у апликацијама, је увек статички, јер пре његовог позивања не постоји ни једна инстанца па самим тим нема начина да он буде инстанцни метод.

У Јави статички методи (изузев метода `main()`) углавном служе за реализацију услужних класа. Мноштво услужних класа смо упознали у поглављу 6, на пример `Random`, `Math`, `System` итд.

Статички метод се разликује од обичног по присуству модификатора `static`, који претходи повратној вредности у потпису метода. Следећа металингвистичка формула представља проширену дефиницију метода из секције 8.4.1 тако да буду укључени и статички методи:

```
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= [static ]<повратни тип> <назив метода>(<параметри>)
```

Статички метод може да приступа само статичким члановима класе, дакле, другим статичким методама или статичким пољима.

```
int polje1;
static int polje2;

void metod1() { ... }

static void metod2() { ... }

static void metod3() {
        // polje1 = 10; // не компајлира се
```

```
        polje2 = 20;
        // metod1(); // не компајлира се
        metod2();
}
```

Приступ инстанцним (нестатичким) члановима није могућ, јер је животни век статичког метода независтан од постојања инстанци дате класе. Стога у случају приступа нестатичким члановима не би било гаранције да они уопште постоје. Друго, чак и да су пре позива статичког метода креиране инстанце дате класе, није јасно на коју би се конкретну инстанцу класе приступ односио.

Позив статичког метода се може представити Бекусовом нотацијом на следећи начин:

```
<позив статичког метода> ::= <позив инстанцног метода>
                           |<назив класе>.<назив метода>(<аргументи>)
<аргументи> ::= <аргумент>{,<аргумент>}
<аргумент> ::= <израз>
```

8.5. Класе – наслеђивање

За претходно креиране класе, могуће је креирати њихове потклассе помоћу кључне речи `extends`. Потклаcа на тај начин постаје надскуп своје наткласе, односно садржи све наслеђене атрибуте и методе те их може користити или чак мењати. Такође, потклаcа може додавати и нове атрибуте, односно методе.
Проширена дефиниција класе сада има следећи облик:

```
<тип класе> ::= class <назив класе> <проширивање><тело класе>
<назив класе> ::= {<идентификатор>.}<идентификатор>
<проширивање> ::= extends <назив класе>
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>)}}
<дефиниција поља> ::= [static ]<декларација и иницијализација променљивих>
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= [static ]<повратни тип> <назив метода>(<параметри>)
<повратни тип> ::= <тип>|void
<назив метода>::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= <тип>
<назив параметра>::= <идентификатор>
<тело метода> ::= <блок>
```

Наредни пример демонстрира наслеђивање.
Пример 11. Дефинисати класу `Srednjoskolac` као потклаcу класе `Ucenik`. Од додатних атрибута, средњошколац поседује и врсту средње школе. Поред тога, поседује и метод за узимање вредности атрибута врсте средње школе.□

```
package rs.math.oop.g08.p11.nasledjivanje;

public class Srednjoskolac extends Ucenik {
       String vrstaSkole;

       String uzmiVrstuSkole() {
          return vrstaSkole;
       }

       void proveriRazred() {
```

```
            if(razred>4 || razred<1)
                System.out.println("Разред "+razred+" није могућ у средњој школи.");
            else
                System.out.println("Разред "+razred+" је у реду.");
        }

        public static void main(String[] args) {
            Srednjoskolac sred = new Srednjoskolac();
            sred.imePrezime="Марко Родић";
            sred.razred=2;
            sred.vrstaSkole="Техничка школа";
            sred.prikaziInformacije();
            System.out.println(sred.vrstaSkole);
            System.out.println(sred.uzmiVrstuSkole());
            sred.proveriRazred();
            sred.razred=5;
            sred.proveriRazred();
        }
}
```

Наслеђене атрибуте и методе је могуће користити на два начина у поткласи: 1) применом тачка-нотације над референцом или 2) позивањем унутар неког од метода поткласе. У другом случају није неопходна тачка нотација, већ само именовање атрибута, односно, метода. Начин 1) се види унутар тела метода `main)=` где је најпре креиран објекат поткласе, а потом се у наредним наредбама манипулисало над атрибутима наткласе, али и атрибутима класе. Начин 2) се примећује у методу `proveriRazred()`, где се, на основу наслеђеног атрибута `razred`, проверава да ли је његова вредност дозвољена за средњу школу. Ова провера је могла да постоји и у класи `Ucenik`, али би у том случају класа `Srednjoskolac` морала да измени (превазиђе) понашање наслеђеног метода, што представља нешто напреднији концепт (биће обрађен у секцији 8.5.4).

Следи резултат извршавања програма.

```
Марко Родић 2. razred
Техничка школа
Техничка школа
Разред 2 је у реду.
Разред 5 није могућ у средњој школи.■
```

8.5.1. Приступање пољима натклаcа и њихово сакривање

У претходном примеру демонстрирано је наслеђивање. Оно се манифестује у два кључна аспекта: 1) структуралном, што подразумева да су подаци (поља) наслеђени и могу се користити у поткласама и 2) аспекту понашања, што подразумева да се наслеђени методи могу користити у поткласама.

У секцији 8.7 биће додатно појашњена могућност употребе поља и метода у поткласама у зависности од тзв. модификатора видљивости.

Приступ пољима натклаcе у методима, референца super

Када је у питању приступање пољима натклаcа, оно је могуће употребом кључне речи `super`. Ова кључна реч може (има смисла) да се позове само унутар инстанцног метода

посматране класе, јер у случају да је метод статички не постоји веза са текућим објектом класе па самим тим ни са његовом наткласом.
Након кључне речи `super` у стандардној тачка-нотацији се надаље приступа траженом пољу наткласе. То поље мора бити искључиво инстанцно – статичко поље није везано за објекат класе, па самим тим ни за његову наткласу.
У претходном примеру се приступало пољу `razred` које формално припада наткласи `Ucenik`. Приметимо да се за тај приступ није користила реч `super`, што је оправдано у ситуацијама када нема именског конфликта са пољем текућег објекта. Међутим, није грешка да се, чак и у тој ситуацији, користи реч `super`, на пример, ако желимо да у коду буде јасније да то поље није дефинисано на нивоу класе текућег објекта.

```
void proveriRazred() {
       if(razred>4 || super.razred<1)
            System.out.println("Разред "+razred+" није могућ у средњој школи.");
       else
            System.out.println("Разред "+razred+" је у реду.");
}
```

Кључна реч `super` омогућава приступ траженом пољу из директне или индиректне наткласе (све до класе `Object`). То значи да када се напише `super.nazivPolja`, Јава ће потражити поље са називом `nazivPolja` у директној наткласи, потом, ако га ту не пронађе, у наредној, итд.
Иако `super` синтаксно подсећа на раније уведену кључну реч `this` треба имати у виду да се ове две кључне речи доста семантички разликују. Наиме, `this` је заиста референца на објекат, па има своју улогу и у фази извршавања Јава програма. Са друге стране, `super` је искључиво програмска конструкција битна у фази писања кода. Она не представља, по аналогији са `this`, референцу ка неком објекту (наткласе). То указује на чињеницу да је наслеђивање механизам користан у фази програмирања који олакшава писање кода и смањује његово дуплирање. Приликом компилације се, међутим, класе конкретизују у виду уније свих поља и метода на линији наслеђивања. У супротном би се, приликом креирања објекта неке класе, креирао по један пратећи објекат сваке наткласе, што не би имало смисла и било би меморијски неефикасно.

Сакривање поља

Сакривање поља (енг. field hiding) се односи на постојање два или више истоимених поља у оквиру хијерархије наслеђивања конкретне класе.

```
class A{
       String naziv;
}

class B extends A{
       String naziv;

       void prikazi(){
              System.out.println(naziv);
              System.out.println(super.naziv);
       }
}
```

Приметити да је у овом случају употреба `super` неопходна, за разлику од претходног, где није било сакривања поља. Наиме, први приступ се односи на поље `naziv` класе `B`,

док се други приступ `super.naziv` односи на поље класе `A`. Сакривање поља компликује програмски кôд и није препоручена његова употреба у Јави.

8.5.2. Испитивање да ли је објекат припада класи

Применом наслеђивања остварује се релација између поткласе и наткласе. Будући да је релација транзитивна, важи да ако је A потклаcа B и B је потклаcа C, онда је и A потклаcа од класе C. У Јави постоји специјални оператор `instanceof` којим се испитује да ли конкретна инстанца припада некој класи. За објекат класе A се може рећи да припада класи A, али и свим њеним наткласама. Такав објекат има све што имају и објекти његових наткласа (када је реч о пољима и методима), и поред тога, још неке евентуалне додатне елементе. Последица је да су све класе у Јави директне или индиректне потклаcе класе `Object` па и испитивање да ли је неки објекат потклаcа класе `Object` увек даје позитиван одговор.

Пример 12. Написати Јава програм који тестира оператор `instanceof` над објектима класе `Object` и класе `Kutija` у разним комбинацијама. □

```
package rs.math.oop.g08.p12.pripadnostKlasi;

public class IspitajPripadnostKlasi {

    public static void main(String[] args) {
        Object o=new Object();
        Kutija k = new Kutija();
        k.dubina=10;
        k.visina=2;
        k.sirina=11;

        System.out.println("Објекат класе Object припада класи Object");
        System.out.println(o instanceof Object);
        System.out.println("Објекат класе Object припада класи Kutija");
        System.out.println(o instanceof Kutija);
        System.out.println("Објекат класе Kutija припада класи Object");
        System.out.println(k instanceof Object);
        System.out.println("Објекат класе Kutija припада класи Kutija");
        System.out.println(k instanceof Kutija);
    }
}
```

Следи резултат рада програма.

```
Објекат класе Object припада класи Object
true
Објекат класе Object припада класи Kutija
false
Објекат класе Kutija припада класи Object
true
Објекат класе Kutija припада класи Kutija
true ■
```

8.5.3. Конверзија између објеката

Објекат неке класе је могуће безбедно конвертовати (кастовати) у објекат било које његове наткласе, укључујући и корену класу `Object`. На пример, ако бисмо из класе `Kutija` имали и изведену класу `ObojenaKutija`, било би могуће написати следећи кôд:

```
Kutija k;
Object o;
ObojenaKutija ok = new ObojenaKutija();
k = ok;
o = new ObojenaKutija();
```

Јасно је да је у питању имплицитна конверзија, јер свака `ObojenaKutija` јесте `Kutija`, или још општије гледано, `Object`.
Ако је потребно вршити конверзију у супротном смеру, од општије ка специфичнијој класи, онда је у питању тзв. експлицитна конверзија:

```
Object o = new ObojenaKutija();
ObojenaKutija ok = (ObojenaKutija) o;
Object o2 = new Kutija();
ObojenaKutija ok2 = (ObojenaKutija) o2; // грешка приликом извршавања
Object o3 = new ObojenaKutija();
Kutija k = (Kutija) o3;
```

У овом смеру на програмеру је одговорност за евентуалну грешку у конверзији. У примеру изнад види се најпре уопштавање инстанце `ObojenaKutija` у `Object`, а потом и њено враћање у инстанцу типа `ObojenaKutija`. Овде је јако важно приметити да се приликом конверзије у општији тип `Object` ништа суштински не мења у оквиру објекта. Дакле, меморија на хипу остаје нетакнута. То надаље омогућава да процес буде повратан, тј. да се инстанца типа `Object` може вратити у инстанцу `ObojenaKutija`.
Други део примера демонстрира ситуацију у којој се врши некоректна конверзија. Наиме, инстанца `Kutija` се конвертује у `Object`, а потом се покушава враћање у тип `ObojenaKutija` који је специфичнији од `Kutija` – свака `ObojenaKutija` јесте `Kutija`, али није свака `Kutija` `ObojenaKutija`. Приликом извршавања овај кôд ће произвести грешку. Да би се овакве ситуације избегле, програмери морају добро анализирати све конверзије које се дешавају у оквиру кода. Додатно, може се користити оператор `instanceof` пре сваке експлицитне конверзије.
Трећи део примера демонстрира адекванту конверзију у којој се општији тип `Object` конвертује назад у мање општи `Kutija`, који је и даље општији од оригиналног типа `ObojenaKutija`.

8.5.4. Позивање метода натклаcа и њихово превазилажење

Позивање метода натклаcа и њихово превазилажење представљају концепте аналогне приступању пољима натклаcа и њиховом сакривању. За разлику од сакривања поља, чија употреба није препоручена, превазилажење метода је један од кориснијих концепата у Јави.

Позивање методе натклаcе, референца super

Да би се приступило методу натклаcе, као и у случају поља, понекад је довољно само навести назив метода – ово се односи на ситуацију када актуелна класа не садржи истоимени метод. Ако класа садржи истоимени метод, потребно је користити кључну

реч `super`. У оба случаја приступа се (позива се) метод прве директне или индиректне наткласе која садржи метод са задатим називом.

```
class A{
        void prikazi(){ ... }
}

class B extends A{
        ...
}

class C extends B{
        void metod(){
                prikazi();
                super.prikazi();
        }

        static void main(String[] args){
                C c = new C();
                c.prikazi(); // подразумева позив метода дефинисаног у A
        }
}
```

У претходном примеру се први позив метода `prikazi()`, као и други `super.prikazi()`, односе на метод дефинисан у класи `A`, јер ни у класи `B` нити класи `C` није дефинисан метод `prikazi()`. Трећи позив, `c.prikazi()`, такође се односи на метод класе `A`.

```
class C extends B{
        void prikazi(){ ... }

        void metod(){
                prikazi(); // позив метода дефинисаног у C
                super.prikazi(); // позив метода дефинисаног у A
        }

        static void main(String[] args){
                C c = new C();
                c.prikazi(); // подразумева позив метода дефинисаног у C
        }
}
```

Са друге стране, ако и класа `C` дефинише метод `prikazi()`, позивањем `prikazi()` се извршава метод из класе `C` (јер је то најближа дефиниција), док `super.prikazi()` игнорише дефиницију из актуелне класе и позива прву најближу из наткласа, што је дефиниција у класи `A`. Позив `c.prikazi()` овде изазива извршавање метода дефинисаног у класи `C`.

Синтакса за позив метода, која обухавата и позив у оквиру датог примерка класе и позив методе натклсе, описана је следећом металингвистичком формулом:

```
<позив инстанцног метода> ::=
                (<инстанцна променљива>|this|super).<назив метода>([<аргументи>])
```

Превазилажење метода

Превазилажење (редефинисање) је дефинисање метода у класи када тај метод већ постоји у некој његовој наткласи. Претходна дефиниција метода и даље постоји, али јој се сада обавезно приступа уз употребу кључне речи `super`. (Ово је већ приказано у претходном фрагменту кода.)
Како би се у коду нагласило да је реч о превазилажењу, а не о првој дефиницији метода са датим називом, користи се специјална мета-језичка ознака `@Override`. Овакве ознаке (започињу знаком @) се користе са циљем допуњавања значења програмског кода Јава и називају се анотације.

```
class C extends B{
        @Override
        void prikazi(){ ... }

        void metod(){
                        prikazi();
                        super.prikazi();
        }

        static void main(String[] args){
                        C c = new C();
                        c.prikazi();
        }
}
```

Када компајлер прочита анотацију `@Override`, он проверава да ли метод са датим називом већ постоји у некој од наткласа. Ако не постоји, пријавиће грешку, јер није могуће превазићи метод који не постоји. Постојање ове анотације смањује евентуалне грешке у којима програмер бива уверен да је редефинисао неки метод, а уствари га по први пут дефинише. На пример, да нема `@Override`, програмер би због словне грешке у називу метода могао да помисли да је у питању превазиђени метод, док би компајлер то посматрао као потпуно нови метод.

```
class C extends B{
        void prikati(){ ... }
        ...
}
```

Постоји много разлога због којих би програмер превазишао понашање метода натклассе. Наводимо у наставку примере три метода који се најчешће превазилазе, сва три дефинисана већ у класи `Object`:

1. `toString()` метод за текстуални приказ објекта;
2. `equals()` метод за поређења два објекта на једнакост;
3. `hashCode()` метод који реализује хеш-функцију, тј. враћа целобројну репрезентацију датог објекта.

Превазилажење методе за испис објекта – toString()

Потпис метода `toString()` је следећи:

```
public String toString()
```

Дакле, повратна вредност је типа `String`. Овај метод има донекле специјалан статус у процесу компилације кода будући да се имплицитно позива приликом позивања неких метода за испис текста. На пример, следеће наредбе су еквивалентне:

```
System.out.println(objekat);
System.out.println(objekat.toString());
```

За раније уведену класу `Ucenik` из Примера 8, редефиниција ове методе би могла да изгледа овако:

```
public class Ucenik {
    String imePrezime;
    int razred;

    @Override
    public String toString() {
        return imePrezime+" "+razred;
    }
}
```

Превазилажење може омогућити да се користе и претходно уведене дефиниције (из наткласа). На пример, у случају класе `Srednjoskolac`, из Примера 11, могуће је искористити ранију (ре)дефиницију из наткласе `Ucenik`.

```
public class Srednjoskolac extends Ucenik {
        String vrstaSkole;

        @Override
        public String toString() {
                return super.toString()+" "+vrstaSkole;
        }
}
```

Позивом `super.toString()` враћа се текстуална репрезентација класе `Ucenik` након чега се на тако добијени текст надовезује још поље `vrstaSkole`, које је карактеристично за средњошколца, а не и за било ког ученика.

Превазилажење методе за поређење једнакости објеката – equals()

Раније је било речи о неадекватности употребе оператора `==` за поређење објеката. Метод `equals()` се обично користи када је потребно „суштински“ упоредити два објекта, тј. упоредити их, не према референцама, већ према ономе што они семантички представљају, тј. према њиховом садржају. Потпис овог метода је следећи:

```
public boolean equals(Object obj)
```

Ако се овај метод не превазиђе, његово подразумевано понашање је исто као и понашање оператора `==`.

Једна могућа редефиниција метода `equals()` за тип `String` је следећа:

```
@Override
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (!(obj instanceof String))
                return false;
        String sObj = (String) obj;
        if (length() != sObj.length())
                return false;
         for (int i = 0; i < length(); i++) {
                if (charAt(i) !=sObj[i])
                                return false;
```

```
        }
        return true;
}
```

Најпре се проверава да ли су референце `this` и прослеђена референца једнаке – ако јесу, онда објекти морају бити и суштински једнаки, јер је реч о истом објекту. У супротном се испитује да ли је прослеђена референца уопште референца на објекат типа `String`. Ако то није испуњено, нема смисла вршити даље поређење. Иначе се врши експлицитна конверзија прослеђеног објекта у `String` тип. Након тога се врши суштинско поређење тако што се прво пореде дужине ниски, а потом и њихов садржај, тј. карактери.

Превазилажење метода за хеш-код објекта

Хеш-код представља целобројну репрезентацију објекта. Примена хеш-кода ће постати јасна у секцији 14.4.2, када се буде говорило о тзв. хеш скуповима и хеш табелама. Хеш-код није нужно јединствен, тј. може постојати више објеката који се сликају у исти хеш-код, али је овакве ситуације пожељно минимизовати адекватном реализацијом метода за рачунање хеш-кода `hashCode()`. Метод `hashCode()` се може звати и хеш функција.

Да би се конзистетно дефинисала хеш функција за неки објекат, програмер мора редефинисати два методе: `hashCode()` и `equals()`. На пример, приликом поређења два уређена пара, метод `equals()` ће вратити `true` само ако су парови једнаки по вредностима обе координате. Метод `hashCode()` је, попут метода `equals()`, дефинисан у кореној класи `Object` и његова подразумевана имплементација обично није адекватна — две објектне променљиве (референце) имају исти хеш-код само ако показују на идентичан објекат. На пример, за два уређена пара са истим координатама `hashCode()` ће вратити различите вредности уколико је реч о објектима на различитим меморијским локацијама (два пута позван оператор `new`). Стога мора бити испуњено да ако важи `a.equals(b)`, онда `a` и `b` морају имати исти хеш-код. У супротном смеру импликација не важи, тј. уколико два објекта имају исти хеш-код онда метод `equals()` не мора вратити `true`. Међутим, ако два објекта имају различите хеш-кодове онда су и сами објекти сигурно различити.

У пракси је погодно да објекти са различитим вредностима хеш-кода најчешће представљају различите објекте. Размотримо тип `String` и потенцијалне дефиниције `hashCode()`. Пример лоше реализације је да метод враћа константну вредност. Нешто боља реализација за тип `String` би сумирала целобројне вредности појединачних карактера (на пример, ASCII[8] вредности). Иако би у већини случајева хеш-код био различит за различите ниске, оваква дефиниција није довољно добра. На пример, за различите речи, које користе исте карактере, хеш-кодови би били исти. Имплементација `hashCode()` за ниске написане у кодној страни Latin1 може изгледати овако:

```
public int hashCode() {
        int h = 0;
        for (byte v : value) {
                h = 31 * h + (v & 0xff);
        }
```

[8] Кодна страница ASCII се може наћи на следећој адреси: https://www.asciitable.com

```
        return h;
}
```

Уместо низа карактера користи се интерна репрезентација ниске на нивоу бајтова (низ `byte[] value`). За сваки бајт ниске ажурира се тренутна вредност хеш-кода (променљива `h`) тако што се на њега дода претходна вредност хеш-кода помножена са 31 и неозначена вредност бајта (`v&0xff`). Оваква дефиниција значајно смањује могућност појаве колизија.

Пример 13. Раније уведену класу `Tacka`, која представља тачку у дводимензионалном простору целобројних координата, прилагодити тако да се у класи превазилази подразумевано понашање метода `toString()`, `equals()` и `hashCode()`. Потом дефинисати класу `Duz`, дефинисану помоћу две тачке, и за њу такође редефинисати поменуте методе. □

```
package rs.math.oop.g08.p13.prevazilazenje;

public class Tacka {
    int x, y;

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof Tacka))
                return false;
        Tacka tObj = (Tacka) obj;
        if(this == tObj)
                return true;
        return x == tObj.x && y == tObj.y;
    }

    @Override
    public int hashCode() {
        int h = x;
        h = 31 * h + y;
        return h;
        // return Objects.hash(x, y);
        // алтернативно постоји и услужни метод hash() у класи Objects
    }
}
```

Две тачке су једнаке уколико су им једнаке обе координате. Хеш функција је реализована слично као и у претходном примеру хеш функција за ниске.

С обзиром да ће хеш функције најчешће бити рачунате на сличан начин, у класи `Objects` је дефинисан статички метод `hash()` који прихвата аргумент променљиве дужине. Код тачке има смисла као аргументе проследити вредности обе координате.

```
package rs.math.oop.g08.p13.prevazilazenje;

import java.util.Objects;

public class Duz {
```

```
    Tacka a;
    Tacka b;

    @Override
    public String toString() {
        return "{" + a.toString() + ", " + b + "}";
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof Duz))
                return false;
        Duz dObj = (Duz) obj;
        if(this == dObj)
                return true;
        return (a.equals(dObj.a) && b.equals(dObj.b))
                        || (a.equals(dObj.b) && b.equals(dObj.a));
    }

    @Override
    public int hashCode() {
        if(a.hashCode()<b.hashCode())
                return Objects.hash(a, b);
        else
                return Objects.hash(b, a);
    }

    public static void main(String[] args) {
        Tacka t1 = new Tacka();
        Tacka t2 = new Tacka();
        Tacka t3 = new Tacka();
        t1.x = 10;
        t1.y = 20;
        t2.x = 40;
        t2.y = 60;
        t3.x = 10;
        t3.y = 20;
        System.out.println(t1.equals(t2)); // false
        System.out.println(t1.equals(t3)); // true
        Duz d1 = new Duz();
        Duz d2 = new Duz();
        Duz d3 = new Duz();
        d1.a = t1;
        d1.b = t2;
        d2.a = t2;
        d2.b = t1;
        d3.a = t1;
        d3.b = t3;
        System.out.println(t1.equals(d1)); // false
        System.out.println(d1.equals(d2)); // true (само редослед другачији)
        System.out.println(d1.equals(d3)); // false
        System.out.println(t1 +" "+ t1.hashCode());
        System.out.println(t2 +" "+ t2.hashCode());
        System.out.println(t3 +" "+ t3.hashCode());
        System.out.println(d1 + " "+ d1.hashCode());
        System.out.println(d2 + " "+ d2.hashCode());
```

```
        System.out.println(d3 + " "+ d3.hashCode());
    }
}
```

Приликом превазилажења метода у класи `Duz` није написано превише кода, јер се методи `toString()`, `equals()` и `hashCode()` ослањају на одговарајуће реализације метода из класе `Tacka`.

У методу `equals()` класе `Duz` није од значаја редослед задавања тачака које описују дуж. Да би се добило конзистентно понашање у вези са методом `hashCode()`, при формирању хеш-кода класе `Duz`, хеш-кодови тачака су искомбиновани у јединственом редоследу (на пример, прво тачка са мањим хеш-кодом па потом са већим). За формирање комбинованог хеш-кода овде је искоришћен претходно поменути метод `Objects.hash()`.

Следи резултат извршавања.

```
false
true
false
true
false
(10, 20) 330
(40, 60) 1300
(10, 20) 330
{(10, 20), (40, 60)} 12491
{(40, 60), (10, 20)} 12491
{(10, 20), (10, 20)} 11521 ■
```

8.5.5. Полиморфизам

Полиморфизам је својство објектно оријентисаног програмирања које омогућава примену истог ентитета (метода, оператора) на примерке различитих класа. Значи, исти ентитети јављају се у више облика, тј. могу извршити различите операције зависно од сценарија. Полиморфизам је уско повезан са наслеђивањем и доприноси вишеструкој искористивости (рејузабилности) софтвера.

Пример 14. Класа `Ljubimac` има потклисе `Macka` и `Pas`. У свакој од ових класа налази се метод `trci()`. Класа `TestLjubimac` служи за тестирање ових класа. У њој се креирају три инстанце класе `Ljubimac` и на сваку инстанцу се примени метод `trci()`. Потребно је коректно приказати начин трчања сваке животиње, без обзира што су све декларисане као `Ljubimac`. Овде је метод `trci()` полиморфан.□

```
package rs.math.oop.g08.p14.polimorfizam;

class Ljubimac{
        public void trci(){
           System.out.println("Неки љубимци не могу да трче!");
        }
}
class Macka extends Ljubimac{
        public void trci(){
           System.out.println("Мачка трчи елегантно и гипко!");
        }
}
```

```
class Pas extends Ljubimac{
        public void trci(){
           System.out.println("Пас трчи брзо и скоковито!");
        }
}

class TestLjubimac{
        public static void main(String args[]){
           Ljubimac zuca = new Pas();
           Ljubimac maca = new Macka();
           Ljubimac kornjaca = new Ljubimac();
           zuca.trci();
           maca.trci();
           kornjaca.trci();
        }
}
```

Приметити да је целокупан кôд смештен у једну датотеку под називом `TestLjubimac.java`. Овакав запис није претерано заступљен у Јави, али је ипак могуће у исту датотеку унети дефиниције више класа, при чему се назив бар једне од њих мора поклопити са називом датотеке.

Објекти класе конкретног љубимца су конвертовани у општију класу `Ljubimac` (појашњено у секцији 8.5.3). Након тога је над свим објектима позиван метод `trci()`, који сигурно постоји у свакој од потклаcа, јер је дефинисан у њиховој наткласи `Ljubimac`. Конверзијом у општији тип није дошло до суштинске измене објекта (на хипу) – сваки објекат је задржао своје основно понашање као и структуру (методи и поља). Ово се може закључити на основу добијених исписа.

```
Пас трчи брзо и скоковито!
Мачка трчи елегантно и гипко!
Неки љубимци не могу да трче! ■
```

8.6. Подешавање почетног стања објекта

Након што се објекат креира на хипу, могуће је подесити његово почетно стање, односно вредности припадајућих поља.

(Уколико се не наведе подешавање стања, примениће се подразумеване вредности: за објектна поља је то референца `null`, за бројевне типове вредност 0, за логички тип вредност `false` итд.)

Могуће је подесити стање применом следећа четири механизма:

1. иницијализација поља при декларацији (било да је статичко или не);
2. иницијализација инстанцних поља у инцијализационом блоку;
3. иницијализација статичких поља у статичком иницијализационом блоку;
4. иницијализација поља у оквиру конструктора (у конструктору се могу вршити и остале активности, а не само подешавање вредности поља).

Када је у питању хронологија ових активности, најпре се извршава иницијализација статичких поља (било путем иницијализације при декларацији или помоћу статичког иницијализационог блока), потом следи иницијализација (инстанцних) поља на исти начин и на крају извршавање наредби конструктора.

Иницијализација приликом декларације подразумева да се у продужетку декларације поља наведе и његова вредност, на пример:

```
class Tacka{
	int x = 10, y = 10;
}
```

8.6.1. Иницијализациони блок

Иницијализациони блок, као што и само име говори, служи да се у оквиру њега иницијализују вредности променљивих, тачније поља.
(Локалне променљиве се иницијализују унутар тела припадајућих метода.)
За разлику од иницијализације приликом декларације променљиве, у оквиру иницијализационог блока је могуће користити контролне структуре, што омогућава иницијализацију на нетривијалан начин, на пример помоћу наредби циклуса или наредби гранања.
Једина разлика између обичног иницијализационог блока и статичког је у томе да се први користи за подешавање вредности инстанцних поља, а други за подешавање статичких поља.
Следећи кôд поставља вредност поља `suma` на суму првих `n` псеудослучајних бројева. Вредност статичког поља `n` подешава у оквиру статичког иницијализационог блока.

```
static int n;
int suma;

static {
	n = 10;
}

{
	suma = 0;
	Random rg = new Random();
	for(int i=0; i<n; i++)
		suma+=rg.nextInt();
}
```

Иницјализациони блок се може описати Бекусовом нотацијом на следећи начин:

```
<иницијализациони блок> ::= [static ]<блок>
```

Проширена дефиниција тела класе, таква да су у њу укључени и иницијализациони блокови, сада има следећи облик:

```
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>|<иницијализациони блок>)}}
```

8.6.2. Конструктор

Приликом креирања конкретног примерка неке класе, увек се позива конструктор те класе. Конструктор се у функционалном смислу може сврстати у методе. За разлику од других метода, конструктор се имплицитно позива, приликом употребе оператора `new`. Друга разлика је у томе што конструктор нема повратну вредност. Потпис конструктора укључује евентуални модификатор, назив класе и на крају листу аргумената, која може бити празна. После тога следи тело конструктора у којем се наводе наредбе. Према томе, конструктор се Бекусовом нотацијом може описати на следећи начин:

```
<конструктор> ::= <име класе>(<параметри>)<блок>
```

```
<име класе> ::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= <тип>
<назив параметра>::= <идентификатор>
```

Позив конструктора се реализује помоћу оператора `new`, тј. израза за креирање објекта, описаног у секцији 5.2.5. Одговарајући конструктор се активира тако што се обезбеди поклапање по броју и типу аргумената у позиву оператора `new` и параметара конструктора.

Сада треба прошитири и дефиницију тела класе, тако да обухвати и конструкторе:

```
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>
                   |<иницијализациони блок>|<конструктор>)}}
```

На пример, овако би могао да изгледа конструктор у класи `Tacka`:

```
class Tacka{
      int x, y;

      public Tacka(){
          x = 10;
          y = 100;
      }

      public static void main(String[] args){
             Tacka t = new Tacka();
      }
}
```

Уколико програмер није дефинисао конструктор за дату класу, преводилац позива подразумевани конструктор. Подразумевани конструктор нема аргумената, и иницијализује све инстанцне и класне променљиве на подразумеване вредности.

```
class Tacka{
      int x, y;

      public static void main(String[] args){
             Tacka t = new Tacka();
      }
}
```

Ако је програмер дефинисао бар један конструктор за дату класу, онда подразумевани конструктор више не постоји.

```
class Tacka{
      int x, y;

      public Tacka(int xp, int yp){
          x = xp;
          y = yp;
      }

      public static void main(String[] args){
             // Tacka t = new Tacka();
             Tacka t = new Tacka(10, 20);
      }
}
```

Позив конструктора без аргумената је у претходном примеру коментарисан – да није, десила би се грешка приликом компилације.

Препоптерећење конструктора и употреба референце this

Конструктори могу бити преоптерећени, исто као и остали методи. Ако постоји додатни конструктор, који има неке нове особине, у њему се може позвати већ постојећи конструктор употребом `this`.

Пример 15. Прилагодити класу `Tacka` тако да садржи два конструктора, један без аргумената, други са вредностима за `x` и `y` координате. Потом преоптеретити и конструкторе за класу `Duz`. Потребно је да постоји празан конструктор, конструктор који прихвата две тачке и конструктор који прихвата четири цела броја (редом координате тачака). □

```
package rs.math.oop.g08.p15.konstruktori;

public class Tacka {
    int x, y;
    static int brojKreiranihTacaka = 0;

    public Tacka() {
        this.x = 0;
        this.y = 0;
        brojKreiranihTacaka++;
    }

    public Tacka(int x, int y) {
        this.x = x;
        this.y = y;
        brojKreiranihTacaka++;
    }

    @Override
    public String toString() {
        return "("+x+", "+y+")";
    }
}
```

```
package rs.math.oop.g08.p15.konstruktori;

public class Duz {
    Tacka a, b;

    public Duz() {
        a = new Tacka(0, 0);
        b = new Tacka(0, 0);
    }

    public Duz(Tacka a, Tacka b) {
        this.a = a; // као и код метода, this разрешава именских конфликт
        this.b = b;
    }

    public Duz(int ax, int ay, int bx, int by) {
```

```
        this(new Tacka(ax, ay), new Tacka(bx, by)); // позив конструктора изнад
    }

    @Override
    public String toString() {
        return "{"+a+", "+b+"}";
    }

    public static void main(String[] args) {
        Duz d1 = new Duz(new Tacka(2, 3), new Tacka(5, 4));
        Duz d2 = new Duz(2, 3, 5, 4);
        System.out.println(d1);
        System.out.println(d2);
        System.out.println(Tacka.brojKreiranihTacaka);
    }
}
```

Дакле, `this` има намену у вези са конструкторима. Наиме, могуће је позивати неки од постојећих конструктора класе коришћењем `this()`. Задавањем одговарајућих аргумената приликом позива `this()` идентификује се одговарајући конструктор. За класу `Duz` то је контруктор `public Duz(Tacka a, Tacka b)`.
Следи испис произведен радом програма.

```
{(2, 3), (5, 4)}
{(2, 3), (5, 4)}
4
```

Референцијална зависност објеката и копирајући конструктор

Објекат чије унутрашње стање, тј. вредност неког поља, може да се промени назива се мутирајући објекат. У супротном се ради о немутирајућем објекту. Стога треба пажљиво радити са конструкторима мутирајућих објеката. Наиме, може се догодити да се, при извршавању конструктора, вредност поља новокреираног објекта постави тако да садржи вредност аргумента конструктора.

- У том случају стварни аргумент конструктора и поље новокреираног објекта постају „везани“ и реферишу на исти објекат (ово се још зове и референцијална зависност).
- Због тога промена објекта аргумента конструктора доводи до промене поља новокреираног објекта и обрнуто.

У примеру 15 је постојала референцијална зависност између дужи и одговарајућих тачака на основу којих је дуж дефинисана. Променом неке од тачака мења се и дуж.

```
Tacka t1 = new Tacka(2, 3);
Tacka t2 = new Tacka(4, 5);
Duz d = new Duz(t1, t2);
System.out.println(d); // {(2, 3), (4, 5)}
t1.x = 20;
System.out.println(t1); // (20, 3)
System.out.println(d); // {(20, 3), (4, 5)}
```

Тачка `t1` описује дуж `d`. Након што се тачка `t1` измени (промена `x` координате), измена се рефлектује и на унутрашње стање дужи `d`. Разлог томе је дефиниција конструктора дужи из претходног примера:

```
public Duz(Tacka a, Tacka b){
```

```
        this.a = a;
        this.b = b;
}
```

Овде се референца ка пољима `this.a` и `this.b`, која описују дуж, поставља тако да показује на прослеђене објекте тачака `a` и `b`, редом.
Овакво понашање често није пожељно, јер нарушава принцип учауривања података у оквиру објекта. Како би се разрешио проблем референцијалне зависности, користи се копирајући конструктор (или конструктор копије).
Копирајући конструктор, по конвенцији, има следећи потпис:

```
public NazivKlase(NazivKlase objekatKojiSeKopira)
```

На пример, копирајући конструктор за класу `Tacka` може бити реализован на следећи начин:

```
public Tacka(Tacka t){
        x = t.x; // на нивоу примитивних типова нема референцијалне зависности
        y = t.y;
}
```

Пошто су координате декларисане као примитивни типови `int`, овде не може постојати референцијална зависност.
Са друге стране, поља класе `Duz` су објектног типа па се копирајући конструктор дефинише на следећи начин:

```
public Duz(Duz d){
        a = new Tacka(d.a);
        b = new Tacka(d.b);
}
```

Овде се види кључна разлика у односу на претходну дефиницију овог конструктора. Уместо да се вредност референце `d.a` додели пољу `a`, овде се креира потпуно нова инстанца класе `Tacka`, која је копија прослеђене `d.a`, и као таква додељује се пољу `a`.
Пример 16. Нека је цртеж у векторској графици представљен низом полигона (претпоставити да их неће бити више од 10). Полигон је описан низом тачака (претпоставити да их неће бити више од 10). Тачка је, као и у претходним примерима, представљена помоћу две целобројне координате. Креирати копирајући конструктор за цртеж.
Да би се ово постигло потребно је направити копирајуће конструкторе за сваку од наведених класа. Свака класа треба сама брине о креирању сопствене копије.□

```
package rs.math.oop.g08.p16.kopirajuciKonstruktor;

public class Tacka {
    int x, y;

    public Tacka(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // копирајући конструктор
    public Tacka(Tacka t) {
        this.x = t.x;
        this.y = t.y;
```

```
    }

    @Override
    public String toString() {
        return "("+x+", "+y+")";
    }
}
```

```
package rs.math.oop.g08.p16.kopirajuciKonstruktor;

public class Poligon {
    Tacka[] tacke = new Tacka[10];

    public Poligon(Tacka...tacke) {
        if(tacke.length>this.tacke.length) {
                System.err.println("Полигон има више од 10 тачака.");
                System.exit(1);
        }
        for(int i=0; i<tacke.length; i++)
                if(tacke[i]==null)
                        break;
                else
                        this.tacke[i]=new Tacka(tacke[i]);
    }

    // копирајући конструктор
    public Poligon(Poligon poligon) {
        this(poligon.tacke);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("[");
        for(Tacka t: tacke)
                if(t==null)
                        break;
                else
                        sb.append(t+" ");
        sb.append("]");
        return sb.toString();
    }
}
```

Полигон може да буде сачињен од највише 10 тачака – ако се проследи више од тога, излази се из програма са поруком о појави грешке. Конструктор који прихвата `Tacka...tacke` омогућава креирање референцијално независних унутрашњих тачака полигона. Конструктор копије се надаље ослања на овај конструктор.

```
package rs.math.oop.g08.p16.kopirajuciKonstruktor;

public class Crtez {
    Poligon[] poligoni;
    int brojPoligona;

    public Crtez() {
```

```
        poligoni = new Poligon[10];
        brojPoligona = 0;
    }

    public Crtez(Crtez crtez) {
        this();
        for(Poligon p: crtez.poligoni)
                if(p==null)
                        break;
                else
                        dodajPoligon(p); // додајемо исте полигоне прављењем копија
    }

    void dodajPoligon(Poligon p) {
        if(brojPoligona>=poligoni.length) {
                System.err.println("Нема више места за нови полигон.");
                System.exit(1);
        }
        poligoni[brojPoligona++]= new Poligon(p); // елиминација реф. зависности
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for(Poligon p:poligoni)
                if(p==null)
                        break;
                else
                        sb.append(p+System.lineSeparator());
        return sb.toString();
    }

    public static void main(String[] args) {
        Tacka t1 = new Tacka(10, 20);
        Poligon p1 = new Poligon(t1, new Tacka(20, 30), new Tacka(100, 200));
        Poligon p2 = new Poligon(new Tacka(100, 50), t1, new Tacka(50, 50));
        Crtez c1 = new Crtez();
        c1.dodajPoligon(p1);
        c1.dodajPoligon(p2);
        System.out.println("c1 пре измене спољашње тачке.");
        System.out.println(c1);
        t1.x = 2000;
        System.out.println("c1 након измене спољашње тачке.");
        System.out.println(c1);
        Crtez c2 = new Crtez(c1);
        c2.poligoni[0].tacke[0].x = 5000;
        System.out.println("c2 као копија c1 са измењеном унутрашњом тачком.");
        System.out.println(c2);
        System.out.println("c1 након што се c2 изменио.");
        System.out.println(c1);
    }
}
```

Класа `Crtez` је мало другачије конципирана, са циљем демонстрације другачије организације кода и употребе конструктора за иницијализацију података. Уместо конструктора, који прихвата полигоне, овде се полигони додају позивањем метода

`dodajPoligon()`. Без обзира на то, и у класи `Crtez` се елиминише референцијална зависност тако што метод `dodajPoligon()` позива копирајући конструктор за сваки од полигона који се додаје у низ.

На основу резултата извршавања програма се може видети да је цртеж `c1` референцијално независан од промене спољашње тачке `t1`. Такође се може видети да је копија цртежа `c1`, под називом `c2`, референцијално независна од стварних промена унутрашњих тачака цртежа `c1`.

```
c1 пре измене спољашње тачке.
[(10, 20) (20, 30) (100, 200) ]
[(100, 50) (10, 20) (50, 50) ]

c1 након измене спољашње тачке.
[(10, 20) (20, 30) (100, 200) ]
[(100, 50) (10, 20) (50, 50) ]

c2 као копија c1 са измењеном унутрашњом тачком.
[(5000, 20) (20, 30) (100, 200) ]
[(100, 50) (10, 20) (50, 50) ]

c1 након што се c2 изменио.
[(10, 20) (20, 30) (100, 200) ]
[(100, 50) (10, 20) (50, 50) ]■
```

Позив конструктора наткласе, референца super

Понекад је понашање конструктора класе надскуп понашања конструктора наткласе, што представља проблем, јер се не жели дуплирање кода. Слично као што је то био случај са позивањем метода наткласе (пример позива `super.toString()`), могуће је позвати и конструктор наткласе, наредбом `super()`.

Пример 17. Дефинисати потклasu класе `Tacka` под називом `OznacenaTacka`. Ова класа има додатно поље `oznaka` типа `String`. Приликом дефиниције конструктора класе `OznacenaTacka` искористити раније дефиниције конструктора класе `Tacka`. □

```java
package rs.math.oop.g08.p17.konstruktorNadklase;

public class Tacka {
    int x, y;

    public Tacka() {
        x = 0;
        y = 0;
    }

    public Tacka(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "("+x+", "+y+")";
    }
}
```

```java
package rs.math.oop.g08.p17.konstruktorNadklase;

public class OznacenaTacka extends Tacka{

    String oznaka;

    public OznacenaTacka() {
        super();
        oznaka = "";
    }

    public OznacenaTacka(int x, int y, String oznaka) {
        super(x, y);
        this.oznaka = oznaka;
    }

    @Override
    public String toString() {
        return oznaka+super.toString();
    }

    public static void main(String[] args) {
        OznacenaTacka ot = new OznacenaTacka(10, 20, "A");
        System.out.println(ot);
        Tacka t = new OznacenaTacka(30, 40, "B");
        System.out.println(t);
    }
}
```

Употреба конструктора `super()` у овом примеру није омогућила велику уштеду у количини кода, јер су две наредбе замењене једном. У пракси се, међутим, дешава да конструктори буду доста сложени и да поред директног подешавања поља на основу аргумената укључују и неке пропратне активности, на пример подешавање вредности зависних поља.

Следи резултат извршавања програма.

```
c1 пре измене спољашње тачке.
[(10, 20) (20, 30) (100, 200) ]
[(100, 50) (10, 20) (50, 50) ]

c1 након измене спољашње тачке.
[(10, 20) (20, 30) (100, 200) ]
[(100, 50) (10, 20) (50, 50) ]

c2 као копија c1 са измењеном унутрашњом тачком.
[(5000, 20) (20, 30) (100, 200) ]
[(100, 50) (10, 20) (50, 50) ]

c1 након што се c2 изменио.
[(10, 20) (20, 30) (100, 200) ]
[(100, 50) (10, 20) (50, 50) ] ■
```

8.7. Модификатори видљивости

Класе могу бити третиране од стране програмера који креира нове типове података (дизајнер класа) или од стране програмера који креирану класу користи у развоју своје апликације (корисник класе). Циљ корисника класе је да прикупи и искористи функционалност класа које је развио дизајнер и на тај начин убрза развој својих апликација. Циљ дизајнера класе је да креира класе преко којих ће изложити спољашњости само оно што је неопходно кориснику, а да све остало буде сакривено.
Када дизајнер креира класу у којој су сви чланови (поља и методе) доступни свима, тада корисник класе у свом програму може да ради шта жели – нема могућности за постављање правила и за обезбеђивање да постављена правила важе.
Контролом видљивости се обезбеђује да корисници класе не могу приступити унутрашњим операцијама и осетљивим деловима имплементације. Поред тога, правилно постављена контрола приступа омогућује дизајнеру класе да промени њен унутрашњи начин рада, а да при том не мора да брине како ће те промене утицати на корисника класе.
Модификатори видљивости (заштите) су специјалне кључне речи које мењају понашање класа, метода и поља. Постоје четири нивоа видљивости:

1. `public`
2. `default` (имплицитни – не записује се)
3. `protected`
4. `private`

У неким објектно оријентисаним језицима други ниво заштите се експлицитно назива `package` па се овако организована заштита назива „4P – заштита“.
То се Бекусовом нотацијом може записати на следећи начин:

```
<модификатор видљивости> ::= private|protected|public
```

Модификатори видљивости се могу применити на саму класу, на поља, методе и контрукторе, што утиче и на синтаксу ових програмских конструкција:

```
<тип класе> ::= [модификатор видљивости> ]class <назив класе>
                              <проширивање><тело класе>
<назив класе> ::= {<идентификатор>.}<идентификатор>
<проширивање> ::= extends <назив класе>
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>
                    |<иницијализациони блок>|<конструктор>)}}
<дефиниција поља> ::=  [<модификатор видљивости> ][static ]
                               <декларација и иницијализација променљивих>
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= [<модификатор видљивости> ][static ]<повратни тип>
                                <назив метода>(<параметри>)
<повратни тип> ::= <тип>|void
<назив метода>::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= <тип>
<назив параметра>::= <идентификатор>
<тело метода> ::= <блок>
<иницијализациони блок> ::= [static ]<блок>
<конструктор> ::= <име класе>(<параметри>)<блок>
<име класе> ::= <идентификатор>
```

8.7.1. Модификатор public

Модификатор `public` даје пољу/методу/класи најширу (јавну) видљивост. То значи да се елементима језика декларисаним `public` модификатором може приступити не само из тела актуелне класе (у којој се налази сам кôд), већ из било које друге класе, која не мора нужно припадати истом пакету као и актуелна класа.

Пример 18. Демонстрирати примену модификатора `public` при раду са пољима, методима и класама. Креирати јавну класу `A` која има јавно поље, метод и конструктор. Након тога, приказати приступ истима из тела класе `A`, затим из тела класе `B`, која се налази у истом пакету где и класа `A`, и на крају из тела класе `C`, која се не налази у истом пакету где и класе `A` и `B`. □

```
package rs.math.oop.g08.p18.publicModifikator.podpaket1;

public class A {
    public int polje;

    public A() { }

    public void metod() { this.polje = 10; }

    void testiraj() {
        polje = 10;
        metod();
        A a = new A();
        a.polje = 20;
        a.metod();
    }

    public static void main(String[] args) {
        A a = new A();
        a.polje = 20;
        a.metod();
    }
}
```

Као што се може видети, јавном пољу `polje` се може приступити помоћу било којег метода припадајуће класе. У случају да је реч о инстанцном методу – `testirajA()`, може се приступити јавном пољу текућег објекта, а може се крерирати и нови објекат и потом приступити његовом јавном пољу, применом тачка-нотације. Слична је ситуација и са јавним методом `metod()`.

Приступ јавном пољу и методу из тела статичког метода `main()` је могућ применом тачка-нотације након што се прво креира инстанца класе `A`.

```
package rs.math.oop.g08.p18.publicModifikator.podpaket1;

public class B {
    void testirajA() {
        A a = new A();
        a.polje = 20;
        a.metod();
    }

    public static void main(String[] args) {
        A a = new A();
        a.polje = 20;
        a.metod();
    }
}
```

У оквиру класе `B` могуће је на скоро исти начин манипулисати објектима класе `A`, односно њеним јавним пољем и методом.

```
package rs.math.oop.g08.p18.publicModifikator.podpaket2;

import rs.math.oop.g08.p18.publicModifikator.podpaket1.A;

public class C {
    void testirajA() {
        A a = new A();
        a.polje = 20;
        a.metod();
    }

    public static void main(String[] args) {
        A a = new A();
        a.polje = 20;
        a.metod();
    }
}
```

Коначно, исто је могуће урадити и из тела класе `C`, која се налази у одвојеном пакету. Уколико класа `A` не би имала јавни конструктор или ако у дефиницији класе не би стајала кључна реч `public` (`public class A`…), не би било могуће креирати објекат класе `A` у оквиру класе `C` па самим тим не би се могло приступати ни пољу нити методу ове класе. Ако се приликом дефинисања класе A изостави дефиниција конструктора, подразумевани конструктор ће користити модификатор `public`. ■

8.7.2. Модификатор package

Модификатор `package`, ако други модификатор није наведен, се подразумева и не наводи се експлицитно. Видљивост је, применом овог модификатора, ограничена на пакет у којем је дефинисана класа.

Пример 19. На исти начин као у примеру 18 демонстрирати употребу модификатора `package`. □

```
package rs.math.oop.g08.p19.packageModifikator.podpaket1;
```

```
class A {
    int polje;

    A() { }

    void metod() { this.polje = 10; }

    void testiraj() {
        polje = 10;
        metod();
        A a = new A();
        a.polje = 20;
        a.metod();
    }

    public static void main(String[] args) {
        A a = new A();
        a.polje = 20;
        a.metod();
    }
}
```

Ова промена нема никакав утицај на понашање у оквиру тела исте класе, што се може видети из кода изнад. Све се уредно компајлира.
Иста је ситуација и у оквиру тела класе `B` (не приказујемо кôд због уштеде простора).

```
package rs.math.oop.g08.p19.packageModifikator.podpaket2;

// import rs.math.oop.g08.p19.packageModifikator.podpaket1.A;

public class C {
    void testirajA() {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
    }

    public static void main(String[] args) {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
    }
}
```

Класу `C` сада није могуће компајлирати, јер се покушава приступ класи која није доступна. Два су разлога недоступности: 1) сама класа `A` има видљивост ограничену на пакет и 2) конструктор класе `A` има видљивост ограничену на пакет. Уколико би се видљивост класе `A` повећала на јавну, `import` директиву би било могуће извршити. Међутим, због недоступности конструктора и даље не би било могуће креирати објекте класе `A`. Обратно, ако би се конструктор прогласио јавним, а класа остала на тренутној видљивости, опет не би било могуће компајлирати кôд, јер није могуће креирати објекат класе која није видљива. ■

8.7.3. Модификатор protected

Модификатор `protected` подразумева видљивост која је и подразумевана (`package`). Додатно, `protected` омогућава видљивост и ван пакета уколико класа из које приступамо наслеђује класу којој се приступа. У овом случају поља и методи којима приступамо нису чланови класе `A` већ изведене класе `C`.

Пример 20. На сличан начин као у примеру 18 демонстрирати употребу модификатора `protected`. Класа `C` сада наслеђује класу `A`. □

```
package rs.math.oop.g08.p20.protectedModifikator.podpaket1;

public class A {
    protected int polje;

    protected A() { }

    protected void metod() { this.polje = 10; }

    void testiraj() {
        polje = 10;
        metod();
        A a = new A();
        a.polje = 20;
        a.metod();
    }

    public static void main(String[] args) {
        A a = new A();
        a.polje = 20;
        a.metod();
    }
}
```

У оквиру класе `A` нема никаквих промена у начину употребе `protected` поља и метода. Када је у питању дефиниција класе, није могуће користити модификатор `protected` – дозвољени модификатори на овом месту су `public` и подразумевани.

С обзиром да `protected` укључује и подразумевану видљивост, видљивост у оквиру класе `B` је иста као и раније па класа `B` није приказана.

```
package rs.math.oop.g08.p20.protectedModifikator.podpaket2;

import rs.math.oop.g08.p20.protectedModifikator.podpaket1.A;

public class C extends A{
    void testirajA() {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
        polje = 20;
        metod();
    }

    public static void main(String[] args) {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
         C c = new C();
         c.polje = 20;
         c.metod();
    }
}
```

Класа `C` сада наслеђује класу `A`, а да би ово било могуће, у дефиницији класе `A` је задржан модификатор видљивости `public`.
Видљивост поља и метода овде се не односи на нове инстанце класе `A`, већ на инстанце класе `C`. Неки делови кода су коментарисани, јер да нису, програм не би било могуће компајлирати.■

8.7.4. Модификатор private

Модификатор `private` нуди најрестриктивнију видљивост и често је ово најпогоднији начин заштите поља, метода или класа.
Употребом овог модификатора видљивост је ограничена на тело класе.
Пример 21. На сличан начин као у примеру 18 демонстрирати употребу модификатора `private`. □

```
package rs.math.oop.g08.p21.privateModifikator.podpaket1;

class A {
    private int polje;

    private A() { }

    private void metod() { this.polje = 10; }

    void testiraj() {
        polje = 10;
        metod();
        A a = new A();
        a.polje = 20;
        a.metod();
    }

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

```
        A a = new A();
        a.polje = 20;
        a.metod();
    }
}
```

У оквиру класе `A` нема никаквих промена у начину употребе `private` поља и метода. Класу `B` је сада немогуће компајлирати, јер приватни конструктор класе `A` не дозвољава креирање објекта, док приватна поља и метод онемогућавају употребу објекта (чак и да конструктор није приватан).

```
package rs.math.oop.g08.p21.privateModifikator.podpaket1;

public class B {
    void testirajA() {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
    }

    public static void main(String[] args) {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
    }
}
```

Ефекат на класу `C` је исти као и на класу `B`, стога је не приказујемо.■

Заштита поља применом модификатора private

У секцији 2.3.5 говорило се о концепту учауривања као једном од кључних ООП концепата. Модификатор `private` има значајну улогу у реализацији учауривања поља јер онемогућава неконтролисани приступ пољима из „спољног света“.
(Када је реч о приступу методима, за њих се користи модификатор `private` кад год је улога метода стриктно локална, тј. друге класе не морају или не треба да га користе.)
Како користити приватна поља ван тела класе?
Одговор је: помоћу метода за узимање и постављање вредности (енг. getter и setter методи).

```
public class A{
        private int polje;

        public int uzmiPolje(){
                return polje;
        }

        public void postaviPolje(int polje){
                this.polje = polje;
        }
}
```

Модификатор метода `uzmiPolje()` и `postaviPolje()` не мора бити `public` – може бити и неки нижи ниво видљивости (с тим што нема смисла декларисати га као `private`).

Сада се намеће ново питање: у чему је разлика између претходно наведене класе `A` и алтернативе дате следећим кодом?

```
public class A{
       public int polje;
}
```

Дефиниција која користи јавно поље је очигледно краћа, а притом нуди идентичну функционалност:

- поље `polje` је могуће прочитати из било којег дела кода;
- пољу `polje` је могуће променити вредност из било којег дела кода.

У првом случају је могуће приступити или мењати поље класе `A` на следећи начин:

```
A a = new A();
a.postaviPolje(10);
System.out.println(a.uzmiPolje());
```

У другом случају је то доста компактније записано:

```
A a = new A();
a.polje = 10;
System.out.println(a.polje);
```

И поред ових предности примене модификатора `public`, у већини ситуација је пожељније користити варијанту која користи приватна поља и придружене методе за узимање и постављање вредности, јер се тиме омогућава заштита унутрашњег стања објекта. То демонстрира наредни пример.

Пример 22. Дефинисати класу `Krug` која је описана центром и полупречником. Додатно, ова класа садржи и поља `obim` и `povrsina`. Потребно је дефинисати конструктор који прихвата као аргументе центар и полупречник. Такође, редефинисати метод `toString()`. Реализовати две варијанте ове класе, једну која не користи заштиту поља и другу која користи. Тестирати и упоредити ове две варијанте. □

```
package rs.math.oop.g08.p22.zastitaPolja;

public class KrugJavni {

    double cx, cy, r, p, o;

    public KrugJavni(double cx, double cy, double r) {
        this.cx = cx;
        this.cy = cy;
        this.r = r;
        p = r*r*Math.PI;
        o = 2*r*Math.PI;
    }

    @Override
    public String toString() {
        return String.format("C=(%.2f, %.2f) r=%.2f P=%.2f O=%.2f",
                        cx, cy, r, p, o);
    }
}
```

Због директног приступа јавним пољима, појављује се грешка. Наиме, након постављаља полупречника на 10 није дошло до прерачунавања површине и обима. Тиме је стање објекта постало неконзистентно. Ова реализација демонстрира лош

начин расподеле одговорности међу објектима – стање објекта је промењено из „спољног света“ који није свестан међузависности поља.

```
package rs.math.oop.g08.p22.zastitaPolja;

public class KrugZasticeni {
    private double cx, cy, r, p, o;

    public KrugZasticeni(double cx, double cy, double r) {
        this.cx = cx;
        this.cy = cy;
        this.r = r;
        preracunaj();
    }

    private void preracunaj() {
        p = r*r*Math.PI;
        o = 2*r*Math.PI;
    }

    public double uzmiCx() { return cx; }

    public void postaviCx(double cx) { this.cx = cx; }

    public double uzmiCy() { return cy; }

    public void postaviCy(double cy) { this.cy = cy; }

    public double uzmiR() { return r; }

    public void postaviR(double r) {
        this.r = r;
        preracunaj();
    }

    public double uzmiP() { return p; }

    public double uzmiO() { return o; }

    @Override
    public String toString() {
        return String.format("C=(%.2f, %.2f) r=%.2f P=%.2f O=%.2f",
                        cx, cy, r, p, o);
    }
}
```

У класи `Krug`, где се користи индиректан приступ пољима, након сваке промене полупречника позива се метод за прерачунавање унутрашњег стања, што производи конзистентне вредности обима и површине.

Друга корисна последица употребе класе `KrugZasticeni` је могућност дефинисања делимичног приступа одређеним пољима. На пример, за поља обим и површина је одлучено да им се омогући само узимање вредности, а не њихова промена (енг. read-only). Ово није могуће постићи у класи `KrugJavni`.

```
package rs.math.oop.g08.p22.ucaurivanjePolja;
```

```
public class TestirajKrugove {

    public static void main(String[] args) {
        KrugJavni kj = new KrugJavni(10, 20.4, 7.3);
        KrugUcaureni kz = new KrugZasticeni(10, 20.4, 7.3);
        System.out.println(kj);
        System.out.println(kz);
        kj.r = 10;
        kz.postaviR(10);
        System.out.println(kj);
        System.out.println(kz);
    }
}
```

Следи приказ резултата рада програма.

```
C=(10.00, 20.40) r=7.30 P=167.42 O=45.87
C=(10.00, 20.40) r=7.30 P=167.42 O=45.87
C=(10.00, 20.40) r=10.00 P=167.42 O=45.87
C=(10.00, 20.40) r=10.00 P=314.16 O=62.83 ■
```

8.8. Модификатор ограничавања – final

Модификатор `final` се користи за ограничавање понашања класа, поља и метода:

- `final` класе не могу бити наслеђене;
- `final` поља не могу променити вредност након иницијализације – ефективно постају константна;
- `final` методи не могу бити редефинисани.

Сада се синтакса класе може дефинисати на следећи начин:

```
<тип класе> ::= [модификатор видљивости> ][final ]class <назив класе>
                              <проширивање><тело класе>
<назив класе> ::= {<идентификатор>.}<идентификатор>
<проширивање> ::= extends <назив класе>
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>
                    |<иницијализациони блок>|<конструктор>)}}
<дефиниција поља> ::=  [<модификатор видљивости> ][static ][final ]
                                <декларација и иницијализација промењивих>
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= [<модификатор видљивости> ][static ][final ]<повратни тип>
                                <назив метода>(<параметри>)
<повратни тип> ::= <тип>|void
<назив метода>::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= [final ]<тип>
<назив параметра>::= <идентификатор>
<тело метода> ::= <блок>
<иницијализациони блок> ::= [static ]<блок>
<конструктор> ::= <име класе>(<параметри>)<блок>
<име класе> ::= <идентификатор>
```

Спречавање наслеђивања

Пример `final` класе је `String`. Због тога се следећи кôд не компајлира:

```
public class MojString extends String{
        ...
}
```

Програмер може креирати и сопствене класе које ће имати овакво понашање.
За класу `String` део дефиниције изгледа овако:

```
public final class String ...{
        ...
}
```

Константна поља

Вредност поља је могуће одржати константном употребом модификатора `final`. Да би се постигло овакво понашање компајлер анализира изворни кôд и утврђује да ли се додела вредности пољу (оператор `=`) применила сам једном. Ако у коду постоји више додела, компајлер пријављује грешку.
Уотреба константних (финалних) поља је реализована у примеру 22, где је искоришћено константно (и статичко) поље `PI` класе `Math`.

```
...
private void preracunaj() {
        p = r*r*Math.PI;
        o = 2*r*Math.PI;
}
...
```

Спречавање редефинисања

У оквиру секције 8.5.4 дефинисано је превазилажење (редефинисање) метода. У питању је моћан концепт који омогућава да се на нижим нивоима хијерархије изведених класа прилагоди понашање метода.
Понекад је ову могућност, ипак, потребно спречити. На пример, ако је корисник (програмер) сигуран да је дефиниција метода у текућој класи тачна за све евентуалне потклассе.

```
public class Kvadrat {
    private double a;
    ...
    final double povrsina() {
        return a*a;
    }
}
```

8.9. Неке динамичке структуре података

Динамичке структуре података су оне структуре података које су у стању да аутоматски прилагођавају своју величину (повећавају или смањују) потребама корисника (програмера). У досадашњем излагању фокус је био на употреби низова, који се не могу подвести под ову групу структура. Низ представља изузетно корисну структуру података чија је основна предност брз случајан приступ (задавањем позиције елемента). Ова брзина је последица логичке организације низа – низ заузима континуирани простор у меморији, што за последицу има да је на основу информације о почетку низа могуће у

O(1) времену приступити члану низа на произвољној позицији. Са друге стране, континуираност простора изазива проблем при раду са низом. Наиме, у случају да је потребно проширити низ, мора се извршити његова реалокација. Под реалокацијом се подразумева налажење новог континуираног дела меморије који је довољно велик да у њега стане проширени низ.

Следеће две подсекције су посвећене двема динамичким структурама података које нуде решење овог проблема. У питању су саморастући низ и повезана листа.

8.9.1. Саморастући низ

Саморастући низ је структура података која своју величину динамички прилагођава потребама у току рада програма. Суштински је реч о класи која енкапсулира стандардни низ и притом реализује разне корисне методе за рад са тим низом. На пример, корисник (програмер) може да затражи да се на позицију 1000 постави одређена вредност. У том моменту енкапсулирани низ може:

1. имати величину већу од 1000, па се постављање вредности на дату позицију врши исто као и при раду са стандардним низом или
2. имати величину мању или једнаку 1000, што ће довести до тога да саморастући низ интерно повећа своју величину и након тога изврши постављање вредности.

Аутоматско повећавање величине низа је пожељно реализовати на ефикасан начин, тј. на начин који смањује број спроведених реалокација меморије. Ово се обично реализује тако што се величина низа повећава за неки процентуални фактор (умножак) уместо за адитивни фактор (сабирак).

Пример 23. Реализовати саморастући низ ниски и демонстрирати његову употребу. Класа `SamorastuciNizNiski` треба да енкапсулира низ ниски и да омогући следеће операције за рад над истим: 1) иницијализацију низа на неку подразумевану величину, на пример, 8; 2) метод којим се поставља вредност на задату позицију; 3) метод којим се приступа вредности на задатој позицији; 4) метод који враћа број елемената низа; 5) метод који враћа капацитет енкапсулираног низа – капацитет није исто што и број елемената низа, дакле, у питању је алоцирани број елемената низа. Сматрати да је број елемената низа једнак позицији последњег постављеног елемента у низу увећаној за један. Елементи који су између, уколико им није постављена вредност експлицитно, представљени су `null` референцама.

Демонстрацију употребе класе `SamorastuciNizNiski` реализовати у засебној класи. ☐

```
package rs.math.oop.g08.p23.samorastuciNiz;

import java.util.Arrays;

public class SamorastuciNizNiski {
    private String[] elementi;
    private int brojElemenata;

    {
        elementi = new String[8];
        brojElemenata = 0;
    }

    private void obezbediKapacitet(int noviKapacitet) {
        if (noviKapacitet <= trenutniKapacitet())
                return;
```

```
        String[] pomocni = elementi;
        elementi = Arrays.copyOf(pomocni, noviKapacitet);
    }

    public void postaviNa(int indeks, String vrednost) {
        if (indeks >= trenutniKapacitet()) {
                int noviKapacitet = Integer.max(indeks, 2 * elementi.length);
                obezbediKapacitet(noviKapacitet);
        }
        elementi[indeks] = vrednost;
        if (indeks + 1 > brojElemenata)
                brojElemenata = indeks + 1;
    }

    public String uzmiSa(int indeks) {
        if (indeks < 0) {
                System.err.println("Грешка: индекс је негативан!");
                return null;
        }
        if (indeks >= elementi.length) {
                System.err.println("Грешка: индекс је већи од величине низа!");
                return null;
        }
        return elementi[indeks];
    }

    public int brojElemenata() {
        return brojElemenata;
    }

    public int trenutniKapacitet() {
        return elementi.length;
    }
}
```

Класа `SamorastuciNizNiski`, поред енкапсулираног низа, садржи и поље `brojElemenata` које одржава информацију о текућем броју елемената. Информацију о капацитету низа није неопходно одржавати у засебном пољу пошто је та информација већ садржана као поље низа `elementi.length`.

Узимање вредности са дате позиције (индекса) је једноставно – проверава се да ли је позиција у оквиру дозвољених граница. Ако јесте, једноставно се враћа вредност на датој позицији у низу. У супротном се исписује грешка и враћа `null`. Ово није оптимална реализација, јер није могуће разликовати случај када је позиција у дозвољеним оквирима, али је вредност `null`, од случаја у којем је позиција ван дозвољених граница – боље је генерисати тзв. изузетак, о којем ће бити речи у поглављу 11.

Реализација метода `postaviNa()` је нешто захтевнија, јер је могуће да позиција на коју се поставља вредност не постоји, тј. позиција је већа или једнака од величине енкапсулираног низа. У тој ситуацији се позива приватни метод `obezbediKapacitet()` који реалоцира низ – дуплира његову величину, након чега преписује елементе из старог низа. Дуплирањем величине низа (или употребом било ког другог множећег фактора) се драматично смањује број потребних реалокација – број реалокација ће бити приближан $\log_2(N)$-3, где је N величина низа (одузима се 3, јер је иницијална величина енкапсулираног низа 8).

```
package rs.math.oop.g08.p23.samorastuciNiz;

import java.util.Scanner;

public class CitajPrikaziParNepar {
    private static void napuni(Scanner sc, SamorastuciNizNiski niz) {
        int i = 0;
        while (sc.hasNext()) {
               String rec = sc.next();
               if (rec.equals("KPAJ"))
                      break;
               niz.postaviNa(i++, sc.next());
        }
    }

    private static void prikaziParniNeparni(SamorastuciNizNiski niz) {
        for (int i = 1; i < niz.brojElemenata(); i += 2)
               System.out.printf("%s\t", niz.uzmiSa(i));
        System.out.println();
        for (int i = 0; i < niz.brojElemenata(); i += 2)
               System.out.printf("%s\t", niz.uzmiSa(i));
        System.out.println();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        SamorastuciNizNiski niz = new SamorastuciNizNiski();
        System.out.println("Унеси текст (или реч KPAJ ћирилицом за крај уноса):");
        napuni(sc, niz);
        sc.close();
        System.out.println("Број речи: " + niz.brojElemenata());
        prikaziParniNeparni(niz);
    }
}
```

Класа `CitajPrikaziNeparPar` демонстрира рад са класом `SamorastuciNizNiski`. Унос вредности у листу је омогућен методом `napuni()` која чита реч по реч са конзоле. Ако је унета реч “KPAJ”, унос се завршава, у супротном се реч додаје на крај повезане листе. Метод `prikaziNeparniParni()` приказује на конзоли прво све речи (ниске) на парним позицијама (ако се позиције броје од 0), након чега у новом реду приказује све речи на непарним позицијама.

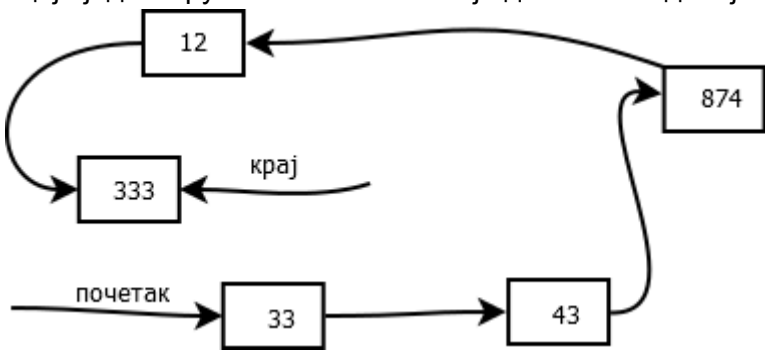
Следи пример једног извршавања програма.

```
Унеси текст (или реч KPAJ ћирилицом за крај уноса):
Ово
су
унете
речи
KPAJ
Број речи: 4
Ово    унете
су    речи ■
```

8.9.2. Повезана листа

Повезана листа припада широј групи тзв. самореферентних или рекурзивних структура података. За њих је карактеристично да одређени део њихове структуре реферише на неки други део те исте структуре. У секцији 5.2.5 аритметички израз је дефинисан на рекурзиван начин – аритметички израз може на пример бити збир два аритметичка израза. Неке од најчешће коришћених самореферентних структура података су дрво и повезана листа.

Логичка организација једноструко повезане листе је дата на следећој слици.



Чворови (елементи) повезане листе могу бити распоређени на произвољним меморијским локацијама. Сваки чвор, поред податка који енкапсулира (на пример, целог броја), има и референцу према следећем чвору. Ово имплицира да је довољно поседовати информацију (референцу) о почетку листе, на основу чега се даље може, праћењем референци, приступити свим осталим чворовима листе. Може се закључити да повезана листа решава претходно поменути проблем реалокације меморије, који постоји при раду са низом. Штавише, реалокација код повезане листе уопште не постоји, јер чворови повезане листе могу бити распоређени на произвољним локацијама у меморији. Повезана листа има и недостатке – најважнији недостатак је O(n) сложеност случајног приступа. Тако да је битно да програмер, приликом избора структуре података, процени шта му је битније: 1) брз случајни приступ или 2) флексибилност у раду са меморијом.

Пример 24. Реализовати једноструко повезану листу ниски. Најпре дефинисати класу која описује појединачни елемент/чвор листе. Ова класа енкапсулира ниску (садржај) и референцу ка следећем чвору. Редефинисати метод за испис чвора. Након тога дефинисати класу која описује једноструко повезану листу која енкапсулира информације о почетку и крају листе. Реализовати три конструктора: 1) конструктор који не прихвата аргумент; 2) конструктор који прихвата једну ниску и 3) конструктор који прихвата низ ниски.

Од метода омогућити следеће: 1) додавање на крај; 2) додавање на почетак; 3) уклањање са краја; 4) уклањање са почетка; 5) дужина (број чворова) листе. У одвојеној класи тестирати све поменуте методе.□

```
package rs.math.oopPitanjaZadaci.g08.p24.jednostukoPovezanaLista;

public class Cvor {
   private String sadrzaj;
   private Cvor sledeci;

   public Cvor(String elem) {
       sadrzaj = elem;
       sledeci = null;
```

```
    }

    public String uzmiSadrzaj() {
        return sadrzaj;
    }

    public void postaviSadrzaj(String sadrzaj) {
        this.sadrzaj = sadrzaj;
    }

    public Cvor uzmiSledeci() {
        return sledeci;
    }

    public void postaviSledeci(Cvor sledeci) {
        this.sledeci = sledeci;
    }

    @Override
    public String toString() {
        return "Елеменат листе: " + sadrzaj;
    }
}
```

```
package rs.math.oopPitanjaZadaci.g08.p24.jednostukoPovezanaLista;

public class PovezanaListaNiski {
    private Cvor pocetak = null;
    private Cvor kraj = null;
    private Cvor tekuci = null;

    public PovezanaListaNiski() {
    }

    public PovezanaListaNiski(String niska) {
        if (niska != null)
                tekuci = kraj = pocetak = new Cvor(niska);
    }

    public PovezanaListaNiski(String[] niske) {
        if (niske == null)
                return;
        for (int i = 0; i < niske.length; i++)
                dodajNaKraj(niske[i]);
        tekuci = pocetak;
    }

    public void dodajNaKraj(String niska) {
        Cvor noviKraj = new Cvor(niska);
        if (pocetak == null)
                pocetak = kraj = noviKraj;
        else {
                kraj.postaviSledeci(noviKraj);
                kraj = noviKraj;
        }
```

```
    }

    public String ukloniSaKraja() {
        if (kraj == null)
                return null;
        if (pocetak == kraj) {
                Cvor jedini = kraj;
                tekuci = pocetak = kraj = null;
                return jedini.uzmiSadrzaj();
        }
        Cvor poslednji = kraj;
        Cvor pretposlednji = pocetak;
        while (pretposlednji.uzmiSledeci() != poslednji)
                pretposlednji = pretposlednji.uzmiSledeci();
        pretposlednji.postaviSledeci(null);
        kraj = pretposlednji;
        return poslednji.uzmiSadrzaj();
    }

    public void dodajNaPocetak(String niska) {
        Cvor noviPocetak = new Cvor(niska);
        if (kraj == null)
                tekuci = pocetak = kraj = noviPocetak;
        else {
                noviPocetak.postaviSledeci(pocetak);
                pocetak = noviPocetak;
                if (tekuci == null)
                        tekuci = pocetak;
        }
    }

    public String ukloniSaPocetka() {
        if (pocetak == null)
                return null;
        if (pocetak == kraj) {
                Cvor jedini = kraj;
                pocetak = kraj = null;
                return jedini.uzmiSadrzaj();
        }
        Cvor prvi = pocetak;
        pocetak = prvi.uzmiSledeci();
        return prvi.uzmiSadrzaj();
    }

    public boolean stigaoDoKraja() {
        if (tekuci == null)
                return true;
        if (tekuci == kraj)
                return true;
        return false;
    }

    public String uzmiPrvi() {
        tekuci = pocetak;
        return (tekuci == null) ? null : tekuci.uzmiSadrzaj();
    }
```

```
    public String uzmiSledeci() {
        if (!stigaoDoKraja())
                tekuci = tekuci.uzmiSledeci();
        return (tekuci == null) ? null : tekuci.uzmiSadrzaj();
    }

    public int brojCvorova() {
        int n = 1;
        Cvor tek = pocetak;
        if (tek == null)
                return 0;
        while (tek != kraj) {
                tek = tek.uzmiSledeci();
                n++;
        }
        return n;
    }
}
```

Поред референци на почетак и крај листе корисно је имати и додатну референцу ка текућем чвору за потребе ажурирања листе.

Конструкција празне листе подразумева да су све референце `null`. Слично, ако се листа иницијализује једном вредношћу, све референце се постављају тако да реферишу на новокреирани чвор који садржи ту вредност. Ако се листа иницијализује низом ниски, онда се пролази кроз низ и за сваку његову наредну вредност позива метод за додавање вредности на крај листе.

Додавања на почетак и крај су слично реализована: 1) у случају да је листа празна, почетак, крај и текући реферишу на новододати чвор; 2) у супротном се ажурирају референце почетка и краја тако да новододати чвор постаје нови почетак односно крај.

Уклањање са почетка има три случаја: 1) ако је листа празна, уклањање није дозвољено, тј. нема ефекта на листу – остаје празна; 2) ако листа има један чвор, све раније поменуте референце постају `null`; 3) иначе се ажурира почетак листе тако да претходни други елемент листе постаје нови почетак.

Уклањање са краја такође има три случаја. Прва два су иста као и код уклањања са почетка. Трећи случај је нешто сложенији, јер је реч о једноструко повезаној листи у којој су референце усмерене од почетних чворова ка крајњим, али не и обратно. Стога је потребно почети од референце `pocetak` и померати се праћењем референце `sledeci` све док се не стигне до чвора чији је `sledeci` једнак референци `kraj`. Другим речима, све док се не стигне до претпоследњег чвора листе. Потом је потребно тај претпоследњи чвор прогласити за нови крај листе. Сложеност овог метода је O(n), за разлику од претходних који су имали сложеност O(1).

За рачунање броја чворова листе потребно је проћи кроз све чворове, што значи да је сложеност O(n). Убрзање може бити постигнуто одржавањем вредности целобројног поља за тренутни број чворова листе – приликом уклањања или додавања чворова би се овај број смањивао односно повећавао.

```
package rs.math.oopPitanjaZadaci.g08.p24.jednostukoPovezanaLista;

import java.util.Scanner;

public class CitajPrikaziNeparPar {
```

```
    private static void napuni(Scanner sc, PovezanaListaNiski lista) {
        while (sc.hasNext()) {
                String rec = sc.next();
                if(rec.equals("KPAJ"))
                        break;
                lista.dodajNaKraj(rec);
        }
    }

    private static void prikaziNeparniParni(PovezanaListaNiski lista) {
        System.out.printf("%s\t", lista.uzmiPrvi());
        while (!lista.stigaoDoKraja()) {
                lista.uzmiSledeci();
                if (lista.stigaoDoKraja())
                        break;
                System.out.printf("%s\t", lista.uzmiSledeci());
        }
        System.out.println();
        lista.uzmiPrvi();
        if (!lista.stigaoDoKraja())
                System.out.printf("%s\t", lista.uzmiSledeci());
        while (!lista.stigaoDoKraja()) {
                lista.uzmiSledeci();
                if (lista.stigaoDoKraja())
                        break;
                System.out.printf("%s\t", lista.uzmiSledeci());
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        PovezanaListaNiski lista = new PovezanaListaNiski();
        System.out.println("Унеси текст (или реч KPAJ ћирилицом за крај уноса):");
        napuni(sc, lista);
        sc.close();
        System.out.println("Број речи: " + lista.brojCvorova());
        prikaziNeparniParni(lista);
    }
}
```

Класа `CitajPrikaziNeparPar` демонстрира рад са класом `PovezanaListaNiski` на исти начин као што је то био случај у примеру 23.
Следи пример једног извршавања програма.

```
Унеси текст (или реч KPAJ ћирилицом за крај уноса):
Ово
су
унете
речи
KPAJ
Број речи: 4
Ово     унете
су      речи ■
```

8.10. Резиме

У овом поглављу могло се видети да Јава представља моћан програмски језик доминантно вођен објектно оријентисаним принципима. Ипак, неки елементи Јаве, попут статичких поља и метода, представљају наслеђе процедуралног језика C, које доприноси ефикасности Јаве.
Кроз паметно осмишљену хијерархију класа (а видећемо касније и интерфејса), Јава је интуитивна и релативно једноставна за програмирање. Велики потенцијал поновне искористивости (рејузабилности) омогућава да се у Јави кôд ретко кад понавља, што чини Јава програме лакшим за одржавање и исправљање грешака. Модификатори видљивости побољшавају учауривање података и понашања у оквиру објеката, што доприноси бољој организованости кода, расподели одговорности међу класама и најбитније, смањивају логичких грешака приликом програмирања.
Богате библиотеке предефинисаних класа омогућавају програмерима бржи развој, али и демонстрирају принципе доброг дизајна кода. Библиотеке класа и концепти који су њиховој основи ће детаљније бити уведени у наредни поглављима – оне ће демонстрирати ширину, тј. општост језика Јава, али и дубину, тј. фину разрађеност неких фундаменталних концепата.

8.11. Питања и задаци

1. Детаљно објаснити и по могућности илустровати примером основне карактеристике програмског језика Јава са становишта објектно оријентисане парадигме.
2. Објаснити и примером илустровати креирање објекта у програмском језику Јава. Да ли постоје класе чије је објекте могуће креирати без употребе оператора `new`?
3. Упоредити оператор `new` за креирање новог објекта у програмском језику Јава са функцијом `malloc()` из програмског језика C.
4. Направити класу која представља студента. Сваки студент има име и презиме, број индекса и годину студија (могуће вредности су прва, друга, трећа, четврта и апсолвент). Затим креирати референце на три студента.
5. Који су најзначајнији методи класе `Object`? Зашто је ова класа важна у програмском језику Јава?
6. Шта је резултат поређења две променљиве које су примитивног типа, а шта је резултат поређења две променљиве које су референце? Илустровати примером.
7. Објаснити и примером илустровати употребу оператора `instanceof`.
8. Како се врши конверзија између објеката основне и наслеђених класа? Илустровати примером ситуације у којима треба бити посебно опрезан при конверзијама.
9. Шта се подразумева под пакетима? Који су разлози за паковање класа и интерфејса у пакете? Навести неке од најчешће коришћених пакета програмског језика Јава и истражити које класе и интерфејси су садржани у тим пакетима.
10. Како се увозе класе из пакета у програм, а како се може приступати класама ако се претходно не увезе пакет? Како Јава виртуелна машина проналази бајт-код класа са којима се оперише?
11. Објаснити процес креирања сопствених пакета.

12. Како се дефинишу поља, како се приступа пољима и колики је опсег важења поља?
13. Која је разлика између статичких поља класе и поља класе која нису статичка? Илустровати примером.
14. Како се дефинишу, а како позивају методе класе? Илустровати примером.
15. Истражити и примером илустровати употребу неких од значајнијих статичких метода класа `Random, Math, System`. Којим члановима класе може приступати статички метод?
16. Објаснити како се остварује и шта се постиже наслеђивањем класа у програмском језику Јава.
17. Упоредити употребу кључних речи `this` и `super` у програмском језику Јава. Кроз примере показати различите употребе ових кључних речи.
18. Шта се подразумева под преопетрећењем метода, а шта под превазилажењем метода? Илустровати примерима када је корисно преоптеретити, а када превазићи неки метод.
19. Објаснити и примером илустровати својство полиморфизма програмског језика Јава.
20. По чему се конструктор разликује од осталих метода класе? У чему је разлика између конструктора, подразумеваног конструктора и копирајућег конструктора? Илустровати примерима.
21. Објснити принцип „4P – заштите“ у програмском језику Јава. Примерима илустровати употребе различитих модификатора видљивости.
22. За шта се користи модификатор `final` у програмском језику Јава?

9. Напредни рад са класама и објектима

Постоје ситуације у којима програмер зна да класа треба да поседује одређени метод, али нема јасну идеју како би се тај метод реализовао – апстрактан је у датом моменту, на датом нивоу хијерархије класа. У зависности од степена апстрактности програмер се може одлучити за коришћење апстрактних класа или интерфејса.
Поред изучавања апстрактних класа и интерфејса, ово поглавље ће бити посвећено још неким напредним темама попут принципа и препорука за објектно оријентисани дизајн, организацији и начину употребе неких значајних интерфејса у оквиру ЈДК итд.

9.1. Апстрактне класе

Апстрактне класе се користе у случају када постоји заједничка наткласа за више класа, али је та наткласа веома општа, тако да се у појединим аспектима њено понашање уопште не може дефинисати. Уз то, истовремено се захтева да свака од поткласа мора да има своју конкретну реализацију тог општег понашања наткласе.
Претпоставимо, на пример, да треба реализовати класу `GeometrijskiObjekat` за геометријски објекат у равни. Та класа је општа и различите врсте геометријских објеката (троугао, квадрат, круг и слично) ће бити изведене из класе `GeometrijskiObjekat`. Као што је познато, геометријски објекти имају обим и површину. С обзиром да се за различите врсте геометријских објеката обим и површина рачунају на различите начине, није могуће на нивоу класе `GeometrijskiObjekat` дефинисати конкретан метод за одређивање обима и површине, већ је потребно да се ти методи дефинишу на нивоу поткласа ове класе. Надаље, пожељно би било да се на нивоу синтаксе језика дефинише обавеза да у свакој од поткласа класе `GeometrijskiObjekat` мора постојати реализација метода за рачунање обима и површине. Ако у некој поткласи не буде таквог метода, већ при превођењу (а не при извршавању) треба да буде пријављена грешка.
Ако у некој класи постоји бар један метод који нема дефиницију (тело) већ само његову декларацију (потпис), тада таква класа мора бити апстрактна класа, а метод без тела је апстрактан метод.
Проглашавање неког метода апстрактним у датој класи ствара програмеру обавезу да метод са тим потписом постоји у свим неапстрактним поткласама те класе, а провера да ли је та обавеза испуњена се врши приликом превођења програма.

9.1.1. Дефинисање апстрактне класе

Апстрактни метод се означава кључном речи `abstract`. Апстрактан метод нема тело, тј. иза декларације апстрактног метода не следи блок, већ сепаратор тачка-зарез.
Ако нека класа садржи апстрактан метод, тада она мора бити апстрактна (мада могу постојати апстрактне класе које не садрже апстрактне методе). За означавање апстрактне класе користи се иста кључна реч `abstract`.
Пример 1. Креирати Јава класу која представља геометријски објекат у равни, тако да иста омогући једноставну надградњу и проширивање. Сваки такав геометријски објекат у перспективи треба да обезбеди могућност за проверу конвексности, проверу ограничености и мерење обима и површине. □

С обзиром на општост побројаних захтева, класа `GeometrijskiObjekat` ће бити реализована као апстрактна класа. Програмски кôд за ту класу се налази у Јава датотеци `GeometrijskiObjekat.java`.

```
package rs.math.oop.g09.p01.apstraktnaKlasa;

public abstract class GeometrijskiObjekat {
   private String oznaka;

   public GeometrijskiObjekat(String oznaka) { this.oznaka = oznaka; }

   public String uzmiOznaku() { return oznaka; }

   public void postaviOznaku(String oznaka) { this.oznaka = oznaka; }

   public abstract boolean jeKonveksan();

   public abstract boolean jeOgranicen();

   public abstract double obim();

   public abstract double povrsina();
}
```

Како сваки геометријски објекат, независно од типа, има ознаку, и како ће се са свим тим ознакама у примерцима класа изведених из `GeometrijskiObjekat` ради на потпуно исти начин, то ће приватно поље за ознаку и јавни конкретни (неапстрактни) методи за узимање и постављање ознаке бити дефинисани у овој класи. Методи за проверу конвексности и ограничености, те за одређивање обима и површине геометријског објекта су превише општи – нема никавог смисленог начина њихове реализације на овом нивоу општости. Стога је њихова реализација делегирана потклласама ове класе, па ће они овде бити дефинисани као апстрактни методи. То даље значи да свака класа изведена из ове класе, а која сама није апстрактна, има обавезу да реализује сва четири апстрактна метода.■

Проширена дефиниција класе, таква да обухвата и апстрактне класе, исказана преко Бекусове нотације, има следећи облик:

```
<тип класе> ::= [модификатор видљивости> ][final |abstract ]class <назив класе>
                              <проширивање><тело класе>
<назив класе> ::= {<идентификатор>.}<идентификатор>
<проширивање> ::= extends <назив класе>
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>
                    |<иницијализациони блок>|<конструктор>)}}
<дефиниција поља> ::=  [<модификатор видљивости> ][static ][final ]
                               <декларација и иницијализација промењивих>
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= [<модификатор видљивости> ][static ][final |abstract ]
                                <повратни тип><назив метода>(<параметри>)
<повратни тип> ::= <тип>|void
<назив метода>::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= [final ]<тип>
<назив параметра>::= <идентификатор>
<тело метода> ::= <блок>
```

```
<иницијализациони блок> ::= [static ]<блок>
<конструктор> ::= <име класе>(<параметри>)<блок>
<име класе> ::= <идентификатор>
```

Апстрактна класа се не може директно инстанцирати, тј. не може се направити инстанца апстрактне класе применом оператора `new` директно на апстактну класу. Међутим, ако се има у виду да је свака инстанца неке класе истовремено и инстанца сваке од наткласа те класе (како конкренте, тако и апстрактне), јасно да се креирана инстанца конкретне потклассе неке апстрактне класе може посматрати као инстанца те апстракне класе.

9.1.2. Наслеђивање између апстрактних и конкретних класа

Синтакса наслеђивања је иста као код претходно описаног наслеђивања између (конкретних/обичних) класа, коришћењем кључне речи `extends`.

Поткласа апстрактне класе:

1. Може реализовати све апстракнте методе и у том случају она постаје конкретна класа.
2. Не мора реализовати апстрактне методе и у том случају и поткласа остаје апстрактна.

Као што је раније истакнуто, програмер користи апстрактне класе ако жели да све потклассе дате класе морају да реализују неко понашање. Како је инстанца конкретне класе истовремено и инстанца апстрактне наткласе (исто као код наслеђивања регуларних класа) овим је омогућено да се, у одређеним случајевима, разне врсте објеката третирају на униформни начин.

Пример 2. Написати Јава класе за представљање геометријских објеката у равни: тачке, дужи и праве. У класи за тачку, поред апстрактних геометријских метода, треба да се реализује и метод за рачунање удаљености до задате тачке. У класи за дуж треба да се реализују методи за рачунање дужине и проверу да ли дуж садржи задату тачку. У класи за праву треба да су реализују методи за проверу да ли права садржи задату тачку, као и методи за проверу да ли су две прослеђене тачке са истих или различитих страна праве. □

Ове класе ће, наравно, бити изведене из класе `GeometrijskiObjekat` описане у претходном примеру. С обзиром да се ради о конкретним класама изведним из апстрактне, потребно је у свакој од ових класа реализовати све апстрактне методе дефинисане у класи `GeometrijskiObjekat`. Дакле, свака од ових класа треба да обезбеди проверу конвексности, проверу ограничености и одређивање обима и површине.

Тачка је геометријски објекат који представља градивни блок за друге геометријске објекте, тј. други објекти ће садржавати тачку. Програмски кôд за класу `Tacka` се налази у Јава датотеци `Tacka.java`.

```
package rs.math.oop.g09.p02.apstraktnaTackaDuzPrava;

import java.util.Objects;
import static java.lang.Math.*;

public class Tacka extends GeometrijskiObjekat {
   private double x;
   private double y;
```

```
   public Tacka(String oznaka, double x, double y) {
      super(oznaka);
      this.x = x;
      this.y = y;
   }

   public Tacka(double x, double y) { this("", x, y); }

   public Tacka(String oznaka) { this(oznaka, 0, 0); }

   public Tacka() { this("O", 0, 0); }

   public Tacka(Tacka t) { this(t.uzmiOznaku(), t.x, t.y); }

   public double uzmiX() { return x; }

   public void postaviX(double x) { this.x = x; }

   public double uzmiY() { return y; }

   public void postaviY(double y) { this.y = y; }

   public double rastojanje(Tacka t) {
      return sqrt(pow(t.x - x, 2) + pow(t.y - y, 2));
   }

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || !(o instanceof Tacka)) return false;
      Tacka tacka = (Tacka) o;
      return Double.compare(tacka.x, x) == 0 && Double.compare(tacka.y, y) == 0;
   }

   @Override
   public int hashCode() { return Objects.hash(x, y); }

   @Override
   public String toString() { return uzmiOznaku() + "(" + x + "," + y + ")"; }

   @Override
   public boolean jeKonveksan() { return true; }

   @Override
   public boolean jeOgranicen() { return true; }

   @Override
   public double obim() { return 0; }

   @Override
   public double povrsina() { return 0; }
}
```

Класа `Tacka` има два атрибута типа `double`, који представљају њене координате у Декартовом координатном систему у равни. У оквиру ове класе је, поред преоптерећених конструктора, дефинисан и метод за рачунање еуклидског растојања

од дате тачке до неке друге. С обзиром да ће овај метод бити позиван и из спољашњости, он је проглашен јавним.
Класа `Tacka` превазилази методе класе `Object` за добијање ниска-репрезентације тачке, за поређење тачака (сматра се да су две тачке једнаке ако су им координате исте) и за одређивање хеш-кода дате тачке.
Класа `Tacka` је неапстрактна (конкретна) па мора садржати реализацију свих метода дефинисаних у апстрактној наткласи `GeometrijskiObjekat` – метода за проверу конвексности, ограничености, као и за израчунавање обима и површине. Ако би недостајао било који од ових метода, и то са истим потписом како је дефинисано у наткласи, тада би преводилац пријавио грешку и програм не би могао да се извршава. У случају тачке, донесена је одлука да (иако се строго гледано на тачку не дефинишу ни обим ни површина) функције за рачунање врате вредност 0.
Дуж је геометријски објекат дефинисан помоћу две тачке. Програмски кôд за класу `Duz` се налази у Јава датотеци `Duz.java`.

```
package rs.math.oop.g09.p02.apstraktnaTackaDuzPrava;

import java.util.Objects;

public class Duz extends GeometrijskiObjekat {
   private Tacka a;
   private Tacka b;

   public Duz(String oznaka, Tacka a, Tacka b) {
      super(oznaka);
      this.a = new Tacka(a);
      this.b = new Tacka(b);
   }

   public Duz(Tacka a, Tacka b) { this("", a, b); }

   public Duz(Duz d) {
      this(d.uzmiOznaku(), d.a, d.b);
   }

   public double duzina() {
      return a.rastojanje(b);
   }

   public boolean sadrzi(Tacka t) {
      boolean kolinearne = ((t.uzmiY() - a.uzmiY()) * (b.uzmiX() - a.uzmiX())
                        == (b.uzmiY() - a.uzmiY()) * (t.uzmiX() - a.uzmiX()));
      if( !kolinearne ) return false;
      if(a.uzmiX() < b.uzmiX()) {
         if (t.uzmiX() < a.uzmiX() || t.uzmiX() > b.uzmiX()) return false;
      }
      else {
         if (t.uzmiX() < b.uzmiX() || t.uzmiX() > a.uzmiX()) return false;
      }
      if(a.uzmiY() < b.uzmiY()) {
         if (t.uzmiY() < a.uzmiY() || t.uzmiY() > b.uzmiY()) return false;
      }
      else {
         if (t.uzmiY() < b.uzmiY() || t.uzmiY() > a.uzmiY()) return false;
```

```
      }
      return true;
   }

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || !(o instanceof Duz)) return false;
      Duz duz = (Duz) o;
      return (a.equals(duz.a) && b.equals(duz.b))
            || (a.equals(duz.b) && b.equals(duz.a));
   }

   @Override
   public int hashCode() {
     if( a.hashCode() <= b.hashCode() ) return Objects.hash(a, b);
     return Objects.hash(b, a);
   }

   @Override
   public String toString() { return uzmiOznaku() + ":[" + a + ";" + b + "]"; }

   @Override
   public boolean jeKonveksan() { return true; }

   @Override
   public boolean jeOgranicen() { return true; }

   @Override
   public double obim() { return a.rastojanje(b); }

   @Override
   public double povrsina() { return 0; }
}
```

Поред преоптерећеног конструктора (укључујући и конструктор копирања), класа `Duz` садржи јавне методе за одређивање дужине дужи и за проверу да ли дуж садржи дату тачку. Због ограничености простора, у конструктору класе `Duz`, где се дуж креира помоћу две тачке, није вршена провера коректности прослеђених параметара, већ се претпоставља да се прослеђене тачке међусобно разликују.

У овој класи су превазиђени методи класе `Object` за поређење дужи (две дужи су једнаке ако им се крајеви поклапају, при чему треба водити рачуна о томе да крајеви две исте дужи не морају бити наведени у истом редоследу) и за одређивање хеш-кода дате дужи, као и метод за добијање ниска-репрезентације дужи.

Конкретна класа `Duz` садржи и реализацију апстрактних метода своје натклase `GeometrijskiObjekat`. Иако, строго математички гледано, дуж нема обим ни површину, донесена је одлука да се за обим узме дужина дужи, а да површина буде 0.

Пример обухвата и геометријски објекат праву, која је интерно представљена коефицијентима своје једначине у имплицитном облику, тј. вредностима коефицијената a, b, c у једначини $ax + by + c = 0$. Програмски кôд за класу `Prava` се налази у Јава датотеци `Prava.java`.

```
package rs.math.oop.g09.p02.apstraktnaTackaDuzPrava;
```

```
public class Prava extends GeometrijskiObjekat {
   private double a;
   private double b;
   private double c;

   public Prava(String oznaka, double a, double b,
                double c) {
      super(oznaka);
      this.a = a;
      this.b = b;
      this.c = c;
   }

   public Prava(double a, double b, double c) { this("", a, b, c); }

   public Prava(String oznaka, Tacka t1, Tacka t2) {
      super(oznaka);
      a = t2.uzmiY() - t1.uzmiY();
      b = t1.uzmiX() - t2.uzmiX();
      c = t1.uzmiY() * (t2.uzmiX() - t1.uzmiX())
                - t1.uzmiX() * (t2.uzmiY() - t1.uzmiY());
   }

   public Prava(Tacka t1, Tacka t2) { this("", t1, t2); }

   public Prava(Prava p) { this(p.uzmiOznaku(), p.a, p.b, p.c ); }

   private double uvrstiKoordinate(Tacka t) {
      return a * t.uzmiX() + b * t.uzmiY() + c;
   }

   public boolean sadrzi(Tacka t) { return (uvrstiKoordinate(t) == 0); }

   public boolean suSaIsteStranePrave(Tacka t1, Tacka t2) {
      return uvrstiKoordinate(t1) * uvrstiKoordinate(t2) > 0;
   }

   public boolean suSaRaznihStranaPrave(Tacka t1, Tacka t2) {
      return !suSaIsteStranePrave(t1, t2);
   }

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null) return false;
      if (!(o instanceof Prava)) return false;
      Prava q = (Prava) o;
      if (a / q.a == b / q.b && a / q.a == c / q.c) return true;
      return false;
   }

   @Override
   public int hashCode() {
      double m = a;
      if (abs(b) > abs(m)) m = b;
      if (abs(c) > abs(m)) m = c;
```

```
        return Objects.hash(a/m, b/m, c/m);
    }

    @Override
    public String toString() {
        return uzmiOznaku() + ":[a=" + a + ";b=" + b + ";c=" + c + "]";
    }

    @Override
    public boolean jeKonveksan() { return true; }

    @Override
    public boolean jeOgranicen() { return false; }

    @Override
    public double obim() { return Double.POSITIVE_INFINITY; }

    @Override
    public double povrsina() { return 0; }
}
```

Ова класа садржи преоптерећене конструкторе (којима се допушта креирање праве помоћу коефицијената и креирање праве помоћу две тачке), као и превазилажење метода класе `Object` за проверу једнакости (две праве су једнаке ако су им коефицијенти пропорционални), за одређивање ниска-репрезентације и за одређивање хеш-кода праве. Због ограничености простора, у конструктору класе `Prava`, где се права креира помоћу две тачке, није вршена провера коректности прослеђених параметара, већ се претпоставља да су прослеђене две међусобно различите тачке.

Поред тога, класа садржи методе за одређивање односа између тачака и праве. Методи за одређивање односа између тачака и праве се ослањају на приватни помоћни метод `uvrstiKoordinate()` за враћање вредности која се добије када се координате дате тачке уврсте у једначину праве.

На крају, класа `Prava` садржи и реализацију свих метода своје апстрактне наткласе, где је усвојена конвенција да обим праве буде бесконачан, а површина 0. ■

Пример 3. Написати Јава класе за геометријске објекте у равни: троугао, четвороугао и круг. У свакој од ових класа треба да се реализује и метод за испитивање да ли објекат класе садржи задату тачку. □

Све претходно побројане класе ће бити изведене из класе `GeometrijskiObjekat` описане у примеру 1. Дакле, свака од ових класа треба да реализује све апстрактне методе дефинисане у њеној наткласи – проверу конвексности, проверу ограничености и рачунање обима и површине.

Овде се круг дефинише помоћу центра круга (типа `Tacka`) и полупречника (типа `double`). Програмски кôд за класу `Krug` се налази у Јава датотеци `Krug.java`.

```
package rs.math.oop.g09.p03.apstraktnaKrugTrougaoCervorougao;

import java.util.Objects;
import static java.lang.Math.*;

public class Krug extends GeometrijskiObjekat {
    private Tacka o;
    private double r;
```

```
    public Krug(String oznaka, Tacka o, double r) {
        super(oznaka);
        this.o = new Tacka(o);
        this.r = r;
    }

    public Krug(Tacka o, double r) { this("", o, r); }

    public Krug(Krug kr) { this(kr.uzmiOznaku(), kr.o, kr.r); }

    public boolean sadrzi(Tacka t) { return t.rastojanje(o) <= r; }

    @Override
    public boolean equals(Object o1) {
        if (this == o1) return true;
        if (o1 == null || getClass() != o1.getClass()) return false;
        Krug krug = (Krug) o1;
        return o.equals(krug.o) && Double.compare(krug.r, r) == 0;
    }

    @Override
    public int hashCode() { return Objects.hash(o, r); }

    @Override
    public String toString() { return uzmiOznaku() + ":[" + o + ";" + r + "]"; }

    @Override
    public boolean jeKonveksan() { return true; }

    @Override
    public boolean jeOgranicen() { return true; }

    @Override
    public double obim() { return 2 * r * PI; }

    @Override
    public double povrsina() { return pow(r, 2) * PI; }
}
```

Класа `Krug` садржи преоптерећене конструкторе (укључујући и копирајући констуктор). У њој се превазилазе методи класе `Object` за проверу једнакости (два круга су једнака ако им се поклапају координате центара и полупречници), за одређивање ниска-репрезентације и за одређивање хеш-кода.

Поред тога, ова класа садржи методе за одређивање припадности тачке кругу (тачка припада кругу ако њено растојање до центра круга није веће од полупречника).

На крају, класа `Krug` садржи и реализацију свих метода своје апстрактне натклase – круг је ограничен и конвексан, а његов обим и површина се рачунају по познатим формулама.

Троугао ће у овом примеру бити одређен са своја три темена. Програмски кôд за класу `Trougao` се налази у Јава датотеци `Trougao.java`.

```
package rs.math.oop.g09.p03.apstraktnaKrugTrougaoCervorougao;

import java.util.Objects;
import static java.lang.Math.sqrt;
```

```
public class Trougao extends GeometrijskiObjekat {
   private Tacka a;
   private Tacka b;
   private Tacka c;

   public Trougao(String oznaka, Tacka a, Tacka b, Tacka c) {
      super(oznaka);
      Tacka o = new Tacka(0,0);
      Tacka[] temena = {a, b, c};
      int ind = 0;
      for(int i=1; i< temena.length; i++)
         if( temena[ind].rastojanje(o) > temena[i].rastojanje(o)
           || (temena[ind].rastojanje(o) == temena[i].rastojanje(o)
              && temena[ind].uzmiX()>temena[i].uzmiX()))
                  ind = i;
      this.a = new Tacka(temena[ind]);
      this.b = new Tacka(temena[(ind+1) % temena.length]);
      this.c = new Tacka(temena[(ind+2) % temena.length]);
   }

   public Trougao(Tacka a, Tacka b, Tacka c) { this("", a, b, c); }

   public Trougao(Trougao tr) { this(tr.uzmiOznaku(), tr.a, tr.b, tr.c); }

   public boolean sadrzi(Tacka t) {
      Duz ivica = new Duz(a, b);
      if (ivica.sadrzi(t)) return true;
      ivica = new Duz(b, c);
      if (ivica.sadrzi(t)) return true;
      ivica = new Duz(c, a);
      if (ivica.sadrzi(t)) return true;
      Prava p = new Prava(a, b);
      if (p.suSaRaznihStranaPrave(c, t)) return false;
      p = new Prava(b, c);
      if (p.suSaRaznihStranaPrave(a, t)) return false;
      p = new Prava(c, a);
      if (p.suSaRaznihStranaPrave(b, t)) return false;
      return true;
   }

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || !(o instanceof Trougao)) return false;
      Trougao trougao = (Trougao) o;
      return a.equals(trougao.a) && b.equals(trougao.b) && c.equals(trougao.c);
   }

   @Override
   public int hashCode() { return Objects.hash(a, b, c); }

   @Override
   public String toString() {
      return uzmiOznaku() + ":[" + a + ";" + b + ";" + c + "]";
   }
```

```
   @Override
   public boolean jeKonveksan() { return true; }

   @Override
   public boolean jeOgranicen() { return true; }

   @Override
   public double obim() {
      return a.rastojanje(b) + b.rastojanje(c) + c.rastojanje(a);
   }

   @Override
   public double povrsina() {
      double ab = a.rastojanje(b);
      double bc = b.rastojanje(c);
      double ca = c.rastojanje(a);
      double s = (ab + bc + ca) / 2;
      return sqrt(s * (s - ab) * (s - bc) * (s - ca));
   }
}
```

Класа `Trougao` садржи преоптерећене конструкторе (укључујући и копирајући констуктор). За разлику од претходно описаних класа, овде се у оквиру најдетаљнијег конструктора врши нетривијална манипулација, којом се координате тачака које су прослеђене конструктору циклично померају тако да прво теме троугла буде она тачка која је најближа координатном почетку. Уколико се, приликом креирања троугла, обезбеди „стандардизован“ редослед навођења темена, тада се у великој мери поједностављује реализација провере једнакости троуглова. Због ограничености простора, у конструктору где се троугао креира помоћу две тачке није вршена провера коректности прослеђених параметара, већ се претпоставља да су сваке две, међу прослеђеним трима тачкама, међусобно различите.

Наравно, класа `Trougao` садржи и превазиђене методе класе `Object` за проверу једнакости, за одређивање ниска-репрезентације датог троугла и његовог хеш-кода.

Као што је познато, сваки троугао је конвексан и ограничен. Обим троугла је сума растојања између суседних темена, а површина троугла је овде одређена по Хероновом обрасцу $P = \sqrt{s(s-a)(s-b)(s-c)}$, где је s полуобим троугла $s = (a+b+c)/2$.

Четвороугао ће, аналогно томе како је урађено код претходне класе, бити дефинисан помоћу своја четири темена. Програмски кôд за класу `Cetvorougao` се налази у Јава датотеци `Cetvorougao.java`.

```
package rs.math.oop.g09.p03.apstraktnaKrugTrougaoCervorougao;

public class Cetvorougao extends GeometrijskiObjekat {
   private Tacka a;
   private Tacka b;
   private Tacka c;
   private Tacka d;

   public Cetvorougao(String oznaka, Tacka a, Tacka b, Tacka c, Tacka d) {
      super(oznaka);
      Tacka o = new Tacka(0,0);
      Tacka[] temena = {a, b, c, d};
```

```
      int ind = 0;
      for(int i=1; i< temena.length; i++)
         if( temena[ind].rastojanje(o) > temena[i].rastojanje(o)
                 || (temena[ind].rastojanje(o) == temena[i].rastojanje(o)
                 && temena[ind].uzmiX()>temena[i].uzmiX()))
            ind = i;
      this.a = new Tacka(temena[ind]);
      this.b = new Tacka(temena[(ind+1) % temena.length]);
      this.c = new Tacka(temena[(ind+2) % temena.length]);
      this.d = new Tacka(temena[(ind+3) % temena.length]);
   }

   public Cetvorougao(Tacka a, Tacka b, Tacka c, Tacka d) { this("", a, b, c, d); }

   public Cetvorougao(final Cetvorougao cetv) {
      this(cetv.uzmiOznaku(), cetv.a, cetv.b, cetv.c, cetv.d);
   }

   public boolean sadrzi(Tacka t) {
      Prava p = new Prava(a, c);
      if (p.suSaRaznihStranaPrave(b, d)) {
         Trougao t1 = new Trougao(a, c, b);
         Trougao t2 = new Trougao(a, c, d);
         return t1.sadrzi(t) || t2.sadrzi(t);
      } else {
         Trougao t1 = new Trougao(b, d, a);
         Trougao t2 = new Trougao(b, d, c);
         return t1.sadrzi(t) || t2.sadrzi(t);
      }
   }

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
      Cetvorougao cet = (Cetvorougao) o;
      return a.equals(cet.a) && b.equals(cet.b) && c.equals(cet.c)
                && d.equals(cet.d);
   }

   @Override
   public int hashCode() { return Objects.hash(a, b, c, d); }

   @Override
   public String toString() {
     return uzmiOznaku() + ":[" + a + ";" + b + ";" + c + ";" + d + "]";
   }

   @Override
   public boolean jeKonveksan() {
      Prava p = new Prava(a, b);
      if (p.suSaRaznihStranaPrave(c, d)) return false;
      p = new Prava(b, c);
      if (p.suSaRaznihStranaPrave(a, d)) return false;
      p = new Prava(c, d);
      if (p.suSaRaznihStranaPrave(a, b)) return false;
```

```
      p = new Prava(d, a);
      if (p.suSaRaznihStranaPrave(b, c)) return false;
      return true;
   }

   @Override
   public boolean jeOgranicen() { return true; }

   @Override
   public double obim() {
      return a.rastojanje(b) + b.rastojanje(c) + c.rastojanje(d) + d.rastojanje(a);
   }

   @Override
   public double povrsina() {
      Prava p = new Prava(a, c);
      if (p.suSaRaznihStranaPrave(b, d)) {
         Trougao t1 = new Trougao(a, c, b);
         Trougao t2 = new Trougao(a, c, d);
         return t1.povrsina() + t2.povrsina();
      } else {
         Trougao t1 = new Trougao(b, d, a);
         Trougao t2 = new Trougao(b, d, c);
         return t1.povrsina() + t2.povrsina();
      }
   }
}
```

И ова класа садржи преоптерећене конструкторе (укључујући и копирајући констуктор). И овде се, у оквиру најдетаљнијег конструктора (којег позивају сви остали констуктори) координате тачака прослеђене конструктору циклично померају тако да прво теме четвороугла буде она тачка која је најближа координатном почетку. Тиме се поједностављује провера једнакости четвороуглова.

Класа `Cetvorougao` садржи и превазиђене методе класе `Object` за проверу једнакости, те за одређивање ниска-репрезентације и хеш-кода датог троугла.

Сваки четвороугао је, наравно, ограничен. Међутим, није сваки четвороугао конвексан, па је потребно за сваку од ивица четвороугла проверити да ли се преостала два темена налазе са исте стране те ивице. Обим четвороугла се рачуна по дефиницији, а рачунање површине се реализује триангулацијом, при чему начин поделе на троуглове зависи од конвексности четвороугла.■

Пример 4. Написати Јава програм којим се, коришћењем претходно развијених класа, креирају различити геометријски објекти у равни (тачке, дужи, праве, кругови, троуглови и четвороуглови), који се затим третирају на униформан начин – прикажу се њихови подаци, њихова конвексност и ограниченост, као и њихова површина.□

Програмски кôд са `main()` методом програма се налази у Јава датотеци `GeometrijskiObjekti.java`, која има следећи садржај.

```
package rs.math.oop.g09.p04.apstraktnaGeometrija;

public class GeometrijskiObjekti {
   public static void main(String[] args) {
      Tacka a = new Tacka("A", 14.5, 15);
      Tacka b = new Tacka(11, 11.5);
      Tacka c = new Tacka(10.45, 22);
```

```
      Tacka d = new Tacka(22.3, 17.5);
      Tacka e = new Tacka("E", 25, 25.5);
      Duz ab = new Duz("AB", a, b);
      Prava p = new Prava("AB", a, d);
      Trougao cdb = new Trougao("CDB", b, c, d);
      Cetvorougao bcde = new Cetvorougao("BCDE", b, c, d, e);
      Krug k1 = new Krug("B_18", b, 18);

      System.out.println("Геометријски објекти ");
      GeometrijskiObjekat[] svi = { a, b, c, d, e, ab, p, cdb, bcde, k1 };
      for (GeometrijskiObjekat go : svi) {
         System.out.print("|" + go);
         if (go.jeKonveksan())
            System.out.print(" конвексан ");
         else
            System.out.print(" неконвексан ");
         if (go.jeOgranicen())
            System.out.println(" ограничен ");
         else
            System.out.println(" неограничен ");
         System.out.print(" обим: " + go.obim());
         System.out.println(" површина: " + go.povrsina() + "|");
      }
   }

}
```

Следи резултат рада програма.

```
Геометријски објекти
|A(14.5,15.0) конвексан  ограничен
 обим: 0.0 површина: 0.0|
|(11.0,11.5) конвексан  ограничен
 обим: 0.0 површина: 0.0|
|(10.45,22.0) конвексан  ограничен
 обим: 0.0 површина: 0.0|
|(22.3,17.5) конвексан  ограничен
 обим: 0.0 површина: 0.0|
|E(25.0,25.5) конвексан  ограничен
 обим: 0.0 површина: 0.0|
|AB:[A(14.5,15.0);(11.0,11.5)] конвексан  ограничен
 обим: 4.949747468305833 површина: 0.0|
|AB:[a=2.5;b=-7.800000000000001;c=80.75000000000001] конвексан  неограничен
 обим: Infinity површина: 0.0|
|CDB:[(11.0,11.5);(10.45,22.0);(22.3,17.5)] конвексан  ограничен
 обим: 35.984199840894505 површина: 60.975000000000016|
|BCDE:[(11.0,11.5);(10.45,22.0);(22.3,17.5);E(25.0,25.5)] неконвексан  ограничен
 обим: 51.43239100196128 површина: 130.82500000000002|
|B_18:[(11.0,11.5);18.0] конвексан  ограничен
 обим: 113.09733552923255 површина: 1017.8760197630929|
```

Уочава се да су разнородни објекти, тј. разне врсте геометријских објеката, третирани на униформни начин (позивани су исти методи), а да резултат тог третирања бива различит, тј. да зависи од стварног типа објекта у времену извршавања. У овом случају је тај заједнички најопштији тип апстрактна класа. Дакле, и код апстрактних класа функционише полиморфизам, описан у секцији 8.5.5, при чему треба истаћи да је његов значај у контексту апстрактних класа додатно повећан. ■

9.2. Интерфејси

У развоју софтвера је често важно да се различите групе програмера договоре око “уговора“ о интеракцији приликом заједничког рада. Свака од тих група треба да буде у могућности да напише свој део кода, а да при томе нема информације како је писан кôд друге стране.
Програмски језик Јава у ту сврху користи интерфејсе. Интерфејс се може третирати као нека врста уговора између класе која га користи (тј. из које се позивају методи интерфејса) и класе која га имплементира. Интерфејси представљају уговоре (они описују понашање), а класе које их имплеметирају и класе које их користе представљају уговорне стране. Наиме, класа која имплементира уговор одређује како се то понашање реализује, а класа која користи имплементирану функционалност то ради позивом метода дефинисаних у интерфејсу, а не директним позивом метода дефинисаних у класи.
Слично апстрактним класама, интерфејси обезбеђују шаблоне за неко понашање, а које ће друге класе користити. За разлику од апстрактних класа које шаблоне за апстрактно понашање прослеђују само поткласама, код интерфејса се шаблони за апстрактно понашање могу проследити било којој класи, независно од ланца наслеђивања.
Интерфејс је референтни тип, сличан апстрактној класи – где није допуштено да неки од метода буду са телом, а да други буду апстрактни, већ су сви методи апстрактни, тј. без реализације. Дакле, у оквиру интерфејса се, по правилу, могу наћи само потписи метода и дефиниције константи. Почев од верзије 8 у Јави је допуштен изузетак од овог правила, којим се, ради елегантне имплементације концепта функционалног програмирања, дозвољава да у интерфејсе буду укључени и тзв. подразумевани методи и приватни методи. Међутим, основни концепти рада са интерфејсима су остали непромењени – овде ће бити описани само ти основни концепти.

9.2.1. Дефинисање интерфејса

Интерфејс се дефинише слично класи, само што се уместо кључне речи `class` користи кључна реч `interface`.
Синтакса за дефинисање интерфејса описана је следећим формулама:

```
<тип интерфејса> ::= interface <назив интерфејса><тело интерфејса>
<назив интерфејса> ::= {<идентификатор>.}<идентификатор>
<тело интерфејса> ::= {{(<опис метода интерфејса>|<дефиниција константе интерфејса>)}}
<опис метода интерфејса> ::= <повратни тип> <назив метода>(<параметри>);
<дефиниција константе интрефејса> ::=  final <декларација и иницијализација променљивих>
```

Декларисање променљиве типа интерфејса је дато формулом:

```
<декларација променљиве типа интерфејса> ::= <тип интерфејса> <инстанцна променљива>
```

Пример 5. Написати Јава интерфејс `Radoznao` који ће садржати два метода.□
Програмски кôд за интерфејс `Radoznao` се налази у Јава датотеци `Radoznao.java`.

```
package rs.math.oopPitanjaZadaci.g09.p05.interfejs;

public interface Radoznao {
	void prikaziUpit();
	String tekstUpita();
}
```

Уочава се да методи интерфејса немају тело. Даље, с обзиром да су по правилу методи интерфејса апстрактни, то нема потребе да се та чињеница додатно наглашава, па не треба наводити `abstract` у оквиру декларације метода у интерфејсу.■

Слично као код апстрактних класа, иако постоје променљиве типа интерфејса, није могуће директно креирати инстанцу интерфејса помоћу оператора `new`. Међутим, могуће је креирати инстанцу конкретне класе која имплементира дати интерфејс, па њу посматрати као инстанцу интерфејса, тј. референцу на креирани објекат доделити променљивој типа интерфејса.

9.2.2. Имплементирање интерфејса од стране класе

Дефинисани интерфејси се имплементирају од стране Јава класа. Када класа имплементира интерфејс, тада се морају имплементирати сви методи интерфејса – не могу се имплементирати само неки од њих, а неки да се оставе неимплементираним.

Саопштавање да дата класа имплементира интерфејс се реализује коришћењем кључне речи `implements` у дефиницији класе, напосредно пре почетка блока. То би се Бекусовом нотацијом могло записати на следећи начин:

```
<тип класе> ::= [модификатор видљивости> ][final |abstract ]class <назив класе>
                              <проширивање><имплементација><тело класе>
<проширивање> ::= extends <назив класе>
<имплементација> ::= implements <листа назива интерфејса>
<листа назива интерфејса> ::= {<назив интерфејса>,}<назив интерфејса>
<назив интерфејса> ::= {<идентификатор>.}<идентификатор>
```

Из горњих формула се јасно види да, ако је то потребно, класа може имплементирати и више од једног интерфејса.

Пример 6. Написати Јава класу која имплементира интерфејс из претходног примера и програм који користи импелементиране методе, и то преко интерфејса и директно преко класе.□

Програмски кôд класе `Strucnjak` која имплементира интерфејс `Radoznao` се налази у Јава датотеци `Strucnjak.java`, која има следећи садржај.

```
package rs.math.oop.g09.p06.interfejsImplementacija;

public class Strucnjak implements Radoznao {
   @Override
   public void prikaziUpit() {
      System.out.println(
           "Реализација метода prikaziUpit() интерфејса Radoznao у класи Strucnjak");
   }

   @Override
   public String tekstUpita() {
      return "Реализација метода tekstUpita() интерфејса Radoznao у класи Strucnjak";
   }

   public String prikaziUpit2(){
      return "Реализација метода prikaziUpit2() у класи Strucnjak";
   }

}
```

Уочава се да класа `Strucnjak` садржи реализацију свих метода дефинисаних у интерфејсу `Radoznao`. Ако је програмер дефинисао да (неапстрактна) класа имплементира интерфејс, онда та класа мора да садржи (било директно, било преко својих наткласа у ланцу наслеђивања) реализацију свих метода интерфејса. Ако не би садржавала реализацију неког од метода, преводилац би приликом генерисања извршног кода пријавио грешку.
Уочава се да класа `Strucnjak` садржи и јавни метод `prikaziUpit2()` који не постоји у интерфејсу `Radoznao`. Овом методу се неће моћи приступити преко интерфејса, јер он у том интерфејсу није дефинисан.
Програмски кôд са улазном тачком програма се налази у Јава датотеци `Pitanja.java`, која има следећи садржај.

```
package rs.math.oop.g09.p06.interfejsImplementacija;

public class Pitanja {
    public static void main(String[] arg)
    {
        Strucnjak p1 = new Strucnjak();
        p1.prikaziUpit();
        System.out.println(p1.tekstUpita());
        System.out.println(p1.prikaziUpit2());

        Radoznao p2 = new Strucnjak();
        p2.prikaziUpit();
        System.out.println(p2.tekstUpita());
    }
}
```

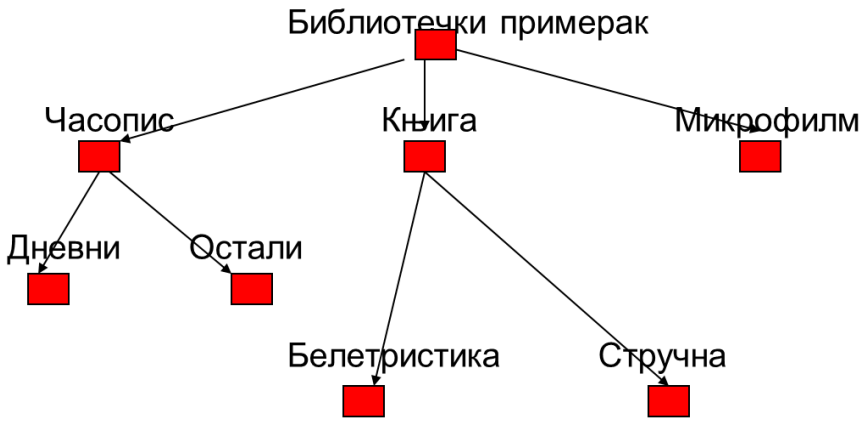
Следи резултат рада програма.

```
Реализација метода prikaziUpit() интерфејса Radoznao у класи Strucnjak
Реализација метода tekstUpita() интерфејса Radoznao у класи Strucnjak
Реализација метода prikaziUpit2() у класи Strucnjak
Реализација метода prikaziUpit() интерфејса Radoznao у класи Strucnjak
Реализација метода tekstUpita() интерфејса Radoznao у класи StrucnjaK
```

Уочава се да је, слично као код апстрактних класа, променљива `p2` типа интерфејса `Radoznao` реферише објекат класе `Strucnjak` (која имплементира тај интерфејс). У том случају, преко промењиве `p2`, не може бити позван метод `prikaziUpit2()`. ■
Када је интерфејс имплементиран од стране неке класе, свака њена поткласа наслеђује све методе и може их, по потреби, превазићи (редефинисати). Самим тим, свака поткласа класе која имплементира неки интерфејс такође имплементира тај интерфејс – није неопходно (нити се препоручује) да се кључна реч `implements` јави и у дефиницији поткласе.

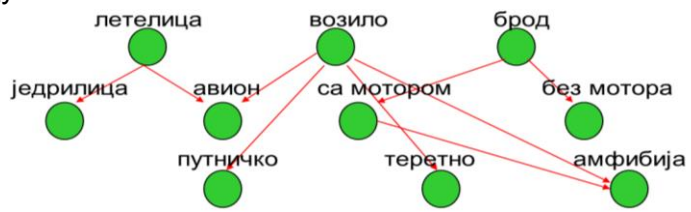
9.2.3. Вишеструко наслеђивање и интерфејси

Велика предност интерфејса, у односу на апстрактне класе, потиче из чињенице да у Јави свака класа има тачно једну (било конкретну, било апстрактну) натклasу, али да може имплементирати више од једног интерфејса. У том случају хијерархија класа дефинисана наслеђивањем има дрвоидну структуру, као што је и приказано на дијаграму који следи. На том дијаграму није приказана класа која представља корен дрвета наслеђивања, класа `Object`.



Пример хијарархије класа

Поред једноструког, постоји вишеструко наслеђивање, где класа може имати више директних наткласа, као што је и приказано на дијаграму која следи. У том случају, хијерархија наслеђивања није дрво, већ усмерени ациклички граф. На дијаграму испод, на пример, класа која се односи на авион има две родитељске класе – класу за возило и класу за летелицу.



Пример вишеструког наслеђивања

Вишеструко наслеђивање није подржано у Јави, али јесте на пример у језику C++. Вишеструко наслеђивање, са једне стране, обезбеђује корисне функционалности за програмера, а са друге стране, повећава сложеност у реализацији програмског окружења језика и начина рада. Креатори Јаве су се одлучили да овај програмски језик и окружење не треба да подржавају вишестуко наслеђивање.

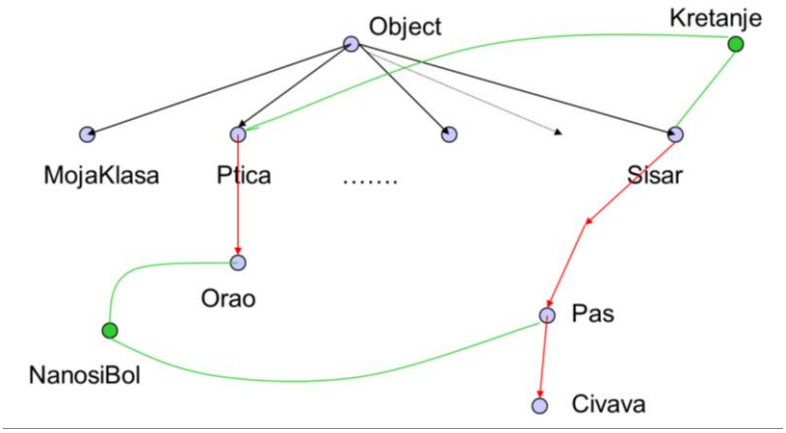
Интерфејси у Јави омогућавају да се у одређеној мери компензује непостојање вишеструког наслеђивања – једна класа може имплементирати више интерфејса, а један интерфејс може проширивати један или више интерфејса.

С обзиром да један интерфејс има могућност проширивања већег броја интерфејса, неопходно је да се ажурира синтакса интерфејса дата у секцији 9.2.1, тако да буде допуштена могућност вишеструког проширивања интерфејса:

```
<тип интерфејса> ::= interface <назив интерфејса><проширивање><тело интерфејса>
<назив интерфејса> ::= {<идентификатор>.}<идентификатор>
<проширивање> ::= extends <листа назива интерфејса>
<листа назива интерфејса> ::= {<назив интерфејса>,}<назив интерфејса>
<назив интерфејса> ::= {<идентификатор>.}<идентификатор>
<тело интерфејса> ::= {{(<опис метода интерфејса>|<дефиниција константе интерфејса>)}}
<опис метода интерфејса> ::= <повратни тип> <назив метода>(<параметри>);
<дефиниција константе интрефејса> ::=  final <декларација и иницијализација променљивих>
```

Дијаграм који следи приказује односе класа и интерфејса за једну хијерархију класа (црвено означава наслеђивање класе, а зелено имплементацију интерфејса). У примеру испод се види да класе `Ptica` и `Sisar` имплементирају интерфејс `Kretanje`, а да класе

`Orao` и `Pas` имплеметирају интерфејс `NanosiBol`. Иако то није експлицитно написано (а и не треба да буде) јасно је да класа `Orao` такође имплементира интерфејс `Kretanje`, јер је она изведена из класе `Ptica`, а та класа директно имплементира интерфејс `Kretanje`. Наравно, у класи `Orao` може бити превазиђена импементација метода интерфејса `Kretanje` која је дефинисана у класи `Ptica` – ако је то неопходно.



Пример наслеђивања класа и имплементације интерфејса

Ако класа имплементира неколико интерфејса, онда се инстанца те класе може посматрати као инстанца сваког од тих интерфејса које дата класа имплементира. Прецизније речено, инстанца дате класе се може посматрати као инстанца ма које наткласе те класе у ланцу наслеђивања и као инстанца ма којег интерфејса који имплементира та класа или нека од наткласа дате класе у ланцу наслеђивања.

Пример 7. Нека нам је на располагању већ развијен интерфејс `Radoznao`. Написати интерфејс `Razuman` са методама `razmotriCinjenice()` и `definisiHipotezu()`, Јава класу `Naucnik` са ниска-атрибутом `ime` која имплементира интерфејсе `Radoznao` и `Razuman`, као и програм који користи имплементиране методе преко променљивих типа интерфејса.□

Програмски кôд са интерфејса `Razuman` се налази у Јава датотеци `Razuman.java`.

```
package rs.math.oop.g09.p07.interfejsVise;

public interface Razuman
{
      void razmotriCinjenice();
      void definisiHipotezu();
}
```

Јасно је да свака класа која имплементира овај интерфејс мора садржати реализацију два горе побројана метода.

Програмски кôд класе `Naucnik` се налази у Јава датотеци `Naucnik.java`.

```
package rs.math.oop.g09.p07.interfejsVise;

public class Naucnik implements Radoznao, Razuman {
   private String ime;

   public Naucnik(String ime) {
```

```
        this.ime = ime;
    }

    public String getIme() {
        return ime;
    }

    @Override
    public void prikaziUpit() {
        System.out.println( "Научник '"+ ime + "' поставља питање");
    }

    @Override
    public String tekstUpita() {
        return "Научник '"+ ime + "' поставља питање";
    }

    @Override
    public void razmotriCinjenice() {
        System.out.println( "Научник '"+ ime + "' разматра чињенице");
    }

    @Override
    public void definisiHipotezu() {
        System.out.println( "Научник '"+ ime + "' дефинише хипотезу");
    }
}
```

Ова класа садржи приватно ниска-поље `ime` и одговарајући јавни метод за приступ `getIme()` .

Класа `Naucnik` имплементира два интерфејса, па мора имати (било директно у себи, било у оквиру неке од својих наткласа) реализације свих метода побројаних у оба интерфејса. Ако то не би био случај, преводилац би генерисао грешку приликом превођења.

Инстанца класе `Naucnik` може, по потреби, да се посматра било као инстанца интерфејса `Radoznao`, или као инстанца интерфејса `Razuman` зависно од тога шта програмеру у датом тренутку више одговара, јер тај објекат јесте `Naucnik` и `Radoznao` и `Razuman` .

Та чињеница је јасно приказана у главном програму. Програмски кôд са улазном тачком програма се налази у Јава датотеци `PitanjaRazmisljanja.java`.

```
package rs.math.oop.g09.p07.interfejsVise;

public class PitanjaRazmisljanja {
    public static void main(String[] arg)
    {
        Naucnik n = new Naucnik("Марковић");

        Radoznao rd = n;
        rd.prikaziUpit();
        System.out.println(rd.tekstUpita());

        Razuman rz = n;
        rz.razmotriCinjenice();
        rz.definisiHipotezu();
```

```
    }
}
```

Следи резултат рада програма.

```
Научник 'Марковић' поставља питање
Научник 'Марковић' поставља питање
Научник 'Марковић' разматра чињенице
Научник 'Марковић' дефинише хипотезу
```

Прве две линије на излазу су добијене када се објекат типа `Naucnik` третира као `Radoznao`, а друге две линије када се исти објекат третира као `Razuman`. С обзиром на чињеницу да класа имплементира интерфејс, није потребно да се врши експлицитна конверзија типа – кастовање.■

9.2.4. Проширивање интерфејса

Као што је и наведено у претходном поглављу, једна класа може имплементирати више интерфејса, али такође један интерфејс може проширити било један интерфејс, било већи број интерфејса.

Проширивање се реализује помоћу кључне речи `extends` – исте која се користи за дефинисање наслеђивања између класа.

У телу интерфејса, који проширује друге интерфејсе, може, али не мора бити додатних метода. Резултујући (проширени) интерфејс је описан унијом свих метода који се појављују у интерфејсима које посматрани интерфејс проширује и метода које сам дефинише. То значи да се вишеструка појављивања истих потписа метода занемарују, односно резултујући (проширени) интерфејс не може имати дуплиране потписе метода. Класа која имплементира проширени интерфејс мора садржати (било она сама, било њене наткласе) реализацију свих метода дефинисаних у проширеном интерфејсу, као и свих метода интерфејса које овај проширени интерфејс проширује.

Пример 8. Нека су нам на располагању већ развијени интерфејси `Radoznao` и `Razuman`, као и претодно развијена класа `Naucnik`. Креирати нови интерфејс `Eksperimentator` који проширује интерфејсе `Radoznao` и `Razuman` и који садржи метод `realizujeEksperimente()`. Креирати Јава класу `Istrazivac` изведену из класе `Naucnik` која имплементира интерфејс `Eksperimentator`, са ниска-атрибутом `probojUOblasti` и са превазиђеним методом за дефинисање хипотезе. Написати Јава програм који позива имплементиране методе преко променљивих типа интерфејса.□

Програмски кôд са интерфејса `Eksperimentator` се налази у Јава датотеци `Eksperimentator.java`.

```
package rs.math.oop.g09.p08.interfejsProsirenje;

public interface Eksperimentator extends Radoznao, Razuman{
   void realizujeEksperimente();
}
```

Класа која буде имплементирала овај интерфејс мора садржати реализацију метода `realizujeEksperimente()`, као и реализацију свих метода у интерфејсима `Radoznao` и `Razuman` и у свим, евентуалним, интерфејсима које ова два интерфејса проширују.

Програмски кôд класе `Istrazivac` се налази у Јава датотеци `Istrazivac.java`.

```
package rs.math.oop.g09.p08.interfejsProsirenje;
```

```
public class Istrazivac extends Naucnik implements Eksperimentator{
   String probojUOblasti;

   public Istrazivac(String ime, String probojUOblasti) {
      super(ime);
      this.probojUOblasti = probojUOblasti;
   }

   @Override
   public void definisiHipotezu() {
      System.out.println("Истраживач '" + getIme()
              +"' је дао хипотезу у научној области " + probojUOblasti);
   }

   @Override
   public void realizujeEksperimente() {
      System.out.println("Истраживач '" + getIme()
              +"' је реализовао експерименте у научној области " + probojUOblasti);
   }
}
```

Програмски кôд са улазном тачком програма се налази у Јава датотеци `Eksperimenti.java`.

```
package rs.math.oop.g09.p08.interfejsProsirenje;

public class Eksperimenti {
   public static void main(String... args) {
      Eksperimentator eksp = new Istrazivac("Петровић", "Молекуларна Биологија");
      eksp.prikaziUpit();
      eksp.razmortiCinjenice();
      eksp.definisiHipotezu();
      eksp.realizujeEksperimente();

      Radoznao rdz = eksp;
      rdz.prikaziUpit();

      Razuman rzm = eksp;
      rzm.razmotriCinjenice();
      rzm.definisiHipotezu();
   }

}
```

Следи резултат рада програма.

```
Научник 'Петровић' поставља питање
Научник 'Петровић' разматра чињенице
Истраживач 'Петровић' је дао хипотезу у научној области Молекуларна Биологија
Истраживач 'Петровић' је реализовао експерименте у научној области Молекуларна Биологија
Научник 'Петровић' поставља питање
Научник 'Петровић' разматра чињенице
Истраживач 'Петровић' је дао хипотезу у научној области Молекуларна Биологија
```

Прве четири линије на излазу су добијене када се објекат типа `Istrazivac` третирао као `Eksperimentator`, пета када се тај исти објекат третира као `Radoznao`, а последње две линије када тај објекат третира као `Razuman`.■

Флексибилност коју даје имитација вишеструког наслеђивања преко интерфејса, допушта да се неки од ранијије постављених задатака реше на елегантији начин, без потребе за наметањем услова који суштински не произилазе из домена проблема.

Пример 9. Креирати Јава програм за рад са геометријским објектима у равни, тако да програм допусти једноставну надоградњу и проширивање. Треба обезебедити проверу конвексности, проверу ограничености, проверу припадности тачке објекту и рачунање обима и површине. Треба реализовати класе за геометријске објекте који представљају тачку, дуж, праву, троугао, четвороугао и круг. На крају треба креирати разноврсне геометријске објекте, приказати их, срачунати њихов укупни обим и укупну површину, а потом за тачку чије се координате учитавају са стандардног улаза одредити који је од креираних геометријских објеката садрже, а који не.□

За разлику од програмског кода у примерима 1-4, на почетку овог поглавља, где је решаван исти проблем преко апстрактних класа, у овом примеру ће тражена функционалност бити реализована преко интерфејса. Ради уштеде простора, сви методи неке класе који се нису променили у међувремену (тј. који су остали исти као у примерима 1-4) неће бити наведени приказани у виду кода, али ће њихова имена бити наведена у пратећем тексту непосредно иза одговарајућег кода.

У претходном решењу овог проблема је, због тога што постоји само једноструко наслеђивање класа у Јави, сва уопштена функционалност била смештана у (апстрактне) методе једне (апстрактне) класе `GeometrijskiObjekat`, и све потклассе те класе су морале имплементирати све методе класе `GeometrijskiObjekat`. То је довело више појава које нису пожељне (у математичком смислу), на пример, креирање метода за одређивање обима и површине тачке, креирање метода за рачунање обима и површине праве и слично.

Када се овај проблем решава преко интерфејса, одговорности (обавезе) које се стављају пред дату класу могу да буду градиране. У том сценарију, на пример, програмер може реализовати методе за ограниченост и за конвексност праве, али нема обавезу да креира методе за рачунање обима или површине за праву.

Дефиниције обавеза који се намећу су дате интерфејсима. У овом случају постоји више група одговорности, па тиме и више интерфејса.

Прва група се односи на проверу конвексности и ограничености. Она је дефинисана интерфејсом `Oblik`. Програмски кôд за интерфејс `Oblik` се налази у Јава датотеци `Oblik.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public interface Oblik {
   boolean jeKonveksan();
   boolean jeOgranicen();
}
```

Друга група функционалности се односи на рачунање обима и површине – дефинисана је интерфејсом `Mera` и налази се у Јава датотеци `Mera.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public interface Mera {
   double obim();
   double povrsina();
}
```

Трећа група функционалности се односи на проверу да ли се дата тачка налази у том геометријском објекту. Она је дефинисана интерфејсом `Sadrzavanje` и налази се у Јава датотеци `Sadrzavanje.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public interface Sadrzavanje {
    boolean sadrzi(Tacka t);
}
```

На крају, уведен је и интерфејс `MeraOblikSadrzavanje`, који обухвата три претходно побројана интерфејса. Програмски кôд за интерфејс `MeraOblikSadrzavanje` се налази у истоименој Јава датотеци.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public interface MeraOblikSadrzavanje extends Mera, Oblik, Sadrzavanje{ }
```

Као што се може видети, интерфејс `MeraOblikSadrzavanje` не садржи ниједан додатни метод.

Одговорности, које треба реализовати у потклasама, више нису груписане на једном месту, па одговарајуће класе могу преузимати одговорност само за потребне групе функционалности, не за све.

Иако у овом дизајну апстрактна класа `GeometrijskiObjekat` не обавезује друге класе шта да реализују, она и даље постоји. Наиме, још увек постоје опште функционалности заједничке за све њене потклase, које се односе на манипулацију ознакама геометријских објеката. Програмски кôд за ту класу се налази у Јава датотеци `GeometrijskiObjekat.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public abstract class GeometrijskiObjekat {
   private String oznaka;

   public GeometrijskiObjekat(String oznaka) { this.oznaka = oznaka; }

   public String uzmiOznaku() { return oznaka; }

   public void postaviOznaku(String oznaka) { this.oznaka = oznaka; }
}
```

Програмски кôд за класу `Tacka` се налази у Јава датотеци `Tacka.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public class Tacka extends GeometrijskiObjekat implements Oblik {

   private double x;
   private double y;   @Override
   public boolean jeKonveksan() { return true; }

   @Override
   public boolean jeOgranicen() { return true; }
}
```

У претходном коду нису наведени (преоптерећени) конструктори, методи за узимање и постављање приватних поља, тј. координата (`uzmiX()`, `postaviX()`, `uzmiY()`,

`postaviY()`), метод `rastojanje()` за одређивање растојања између тачака. Изостављено је превазилажење метода класе `Object` (за добијање ниска-репрезентације `toString()`, за поређење једнакости `equals()` и за одређивање хеш-кода дате тачке `hashCode()`) – они су исти као у примеру 2. Наведена превазилажења ће, због истоветности са онима у примеру 2 (за дуж и праву) и примеру 3 (за круг, троугао и четвороугао) бити изостављена и у наредним класама које реализују те геометријске елементе.

У неколико наредних класа за опис геометријских елемената, биће неопходно да се превазиђу сви методи класе `Object`, као што је овде истакнуто. То ће бити изостављено у наредним класама.

У овом случају, класа `Tacka` реализује само функционалности за проверу конвексности и ограничености. Тачка нема обим ни површину, нити се тачка може садржавати у тачки. Стога ће класа `Tacka` имплеметирати само интерфејс `Oblik`.

Програмски кôд за класу `Duz` се налази у Јава датотеци `Duz.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public class Duz extends GeometrijskiObjekat implements Oblik, Sadrzavanje {
   private Tacka a;
   private Tacka b;

   @Override
   public boolean jeKonveksan() { return true; }

   @Override
   public boolean jeOgranicen() { return true; }

   @Override
   public boolean sadrzi(Tacka t) {
      boolean kolinearne = ((t.uzmiY() - a.uzmiY()) * (b.uzmiX() - a.uzmiX())
                  == (b.uzmiY() - a.uzmiY()) * (t.uzmiX() - a.uzmiX()));
      if( !kolinearne ) return false;
      if(a.uzmiX() < b.uzmiX()) {
         if (t.uzmiX() < a.uzmiX() || t.uzmiX() > b.uzmiX()) return false;
      }
      else {
         if (t.uzmiX() < b.uzmiX() || t.uzmiX() > a.uzmiX()) return false;
      }
      if(a.uzmiY() < b.uzmiY()) {
         if (t.uzmiY() < a.uzmiY() || t.uzmiY() > b.uzmiY()) return false;
      }
      else {
         if (t.uzmiY() < b.uzmiY() || t.uzmiY() > a.uzmiY()) return false;
      }
      return true;
   }
}
```

У приказаном програмском коду нису наведени (преоптерећени) конструктори и метод `duzina()` за одређивање дужине дужи.

С обзиром на чињеницу да имплементира интерфејсе `Oblik` и `Sadrzavanje`, класа `Duz` мора да реализује проверу конвексности и ограничености (јасно је да је дуж конвексна и ограничена), као и проверу да ли садржи дату тачку (своди се на проверу да ли тачка

припада правој коју дефинишу крајеви дужи и да ли су x и y координате те тачке између x и y координата крајева дужи).
Програмски кôд за класу `Prava` се налази у Јава датотеци `Prava.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public class Prava extends GeometrijskiObjekat implements Oblik, Sadrzavanje {

   private double a;
   private double b;
   private double c;

   private double uvrstiKoordinate(Tacka t) {
      return a * t.uzmiX() + b * t.uzmiY() + c;
   }

   @Override
   public boolean jeKonveksan() { return true; }

   @Override
   public boolean jeOgranicen() { return false; }

   @Override
   public boolean sadrzi(Tacka t) { return (uvrstiKoordinate(t) == 0); }
}
```

У програмском коду нису навдени (преоптерећени) конструктори, методи `suSaIsteStranePrave()` и `suSaRaznihStranaPrave`() за проверу да ли су две тачке са истих страна праве или не.
Програмски кôд за класу `Krug` се налази у Јава датотеци `Krug.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public class Krug extends GeometrijskiObjekat implements MeraOblikSadrzavanje {
   private Tacka o;
   private double r;

   @Override
   public double obim() { return 2 * r * PI; }

   @Override
   public double povrsina() { return pow(r, 2) * PI; }

   @Override
   public boolean jeKonveksan() { return true; }

   @Override
   public boolean jeOgranicen() { return true; }

   @Override
   public boolean sadrzi(Tacka t) { return t.rastojanje(o) <= r; }
}
```

Програмски кôд за класу `Trougao` се налази у Јава датотеци `Trougao.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public class Trougao extends GeometrijskiObjekat implements MeraOblikSadrzavanje {
```

```
   private Tacka a;
   private Tacka b;
   private Tacka c;

   @Override
   public double obim() {return a.rastojanje(b) + b.rastojanje(c) + c.rastojanje(a); }

   @Override
   public double povrsina() {
      double ab = a.rastojanje(b);
      double bc = b.rastojanje(c);
      double ca = c.rastojanje(a);
      double s = (ab + bc + ca) / 2;
      return sqrt(s * (s - ab) * (s - bc) * (s - ca));
   }

   @Override
   public boolean jeKonveksan() { return true; }

   @Override
   public boolean jeOgranicen() { return true; }

   @Override
   public boolean sadrzi(Tacka t) {
      Duz ivica = new Duz(a, b);
      if (ivica.sadrzi(t)) return true;
      ivica = new Duz(b, c);
      if (ivica.sadrzi(t)) return true;
      ivica = new Duz(c, a);
      if (ivica.sadrzi(t)) return true;
      Prava p = new Prava(a, b);
      if (p.suSaRaznihStranaPrave(c, t)) return false;
      p = new Prava(b, c);
      if (p.suSaRaznihStranaPrave(a, t)) return false;
      p = new Prava(c, a);
      if (p.suSaRaznihStranaPrave(b, t)) return false;
      return true;
   }
}
```

Програмски кôд за класу `Cetvorougao` се налази у Јава датотеци `Cetvorougao.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

public class Cetvorougao extends GeometrijskiObjekat implements MeraOblikSadrzavanje {
   private Tacka a;
   private Tacka b;
   private Tacka c;
   private Tacka d;

   @Override
   public double obim() {
      return a.rastojanje(b) + b.rastojanje(c) + c.rastojanje(d) + d.rastojanje(a);
   }

   @Override
   public double povrsina() {
```

```
      Prava p = new Prava(a, c);
      if (p.suSaRaznihStranaPrave(b, d)) {
         Trougao t1 = new Trougao(a, c, b);
         Trougao t2 = new Trougao(a, c, d);
         return t1.povrsina() + t2.povrsina();
      } else {
         Trougao t1 = new Trougao(b, d, a);
         Trougao t2 = new Trougao(b, d, c);
         return t1.povrsina() + t2.povrsina();
      }
   }

   @Override
   public boolean jeKonveksan() {
      Prava p = new Prava(a, b);
      if (p.suSaRaznihStranaPrave(c, d)) return false;
      p = new Prava(b, c);
      if (p.suSaRaznihStranaPrave(a, d)) return false;
      p = new Prava(c, d);
      if (p.suSaRaznihStranaPrave(a, b)) return false;
      p = new Prava(d, a);
      if (p.suSaRaznihStranaPrave(b, c)) return false;
      return true;
   }

   @Override
   public boolean jeOgranicen() { return true; }

   @Override
   public boolean sadrzi(Tacka t) {
      Prava p = new Prava(a, c);
      if (p.suSaRaznihStranaPrave(b, d)) {
         Trougao t1 = new Trougao(a, c, b);
         Trougao t2 = new Trougao(a, c, d);
         return t1.sadrzi(t) || t2.sadrzi(t);
      } else {
         Trougao t1 = new Trougao(b, d, a);
         Trougao t2 = new Trougao(b, d, c);
         return t1.sadrzi(t) || t2.sadrzi(t);
      }
   }
}
```

Улазна тачка програма се налази у Јава датотеци `MereObliciSadrzavanja.java`.

```
package rs.math.oop.g09.p09.interfejsGeometrija;

import java.util.Scanner;

public class MereObliciSadrzavanja {
   public static void main(String[] args) {
      Tacka a = new Tacka("A", 14.5, 15);
      Tacka b = new Tacka("B", 10, 11.5);
      Tacka c = new Tacka( "C", 10.45, 22);
      Tacka d = new Tacka( "D", 22.3, 17.5);
      Tacka e = new Tacka("E", 25, 25.5);
      Duz ab = new Duz("дAB", a, b);
```

```
      Prava p = new Prava("пAB", a, d);
      Trougao bcd = new Trougao("тBCD", b, c, d);
      Cetvorougao bcde = new Cetvorougao("чBCDE", b, c, d, e);
      Krug k1 = new Krug("кB18", b, 18);

      double ukupanObim = 0;
      double ukupnaPovrsina = 0;
      System.out.println("\nГеометријски објекти:");
      GeometrijskiObjekat[] svi = {a, b, c, d, e, ab, p, bcd, bcde, k1};
      for (GeometrijskiObjekat go : svi) {
         System.out.printf("| %s", go);
         if (go instanceof Oblik) {
            Oblik obl = (Oblik) go;
            if (obl.jeKonveksan())
               System.out.print(" конвексан ");
            else
               System.out.print(" неконвексан ");
            if (obl.jeOgranicen())
               System.out.println(" ограничен ");
            else
               System.out.println(" неограничен");
         }
         else
            System.out.println(" не подржава облике");
         if( go instanceof Mera){
            Mera m = (Mera) go;
            System.out.print(" обим: " + m.obim());
            System.out.println(" површина: " + m.povrsina() + "|");
            ukupanObim += m.obim();
            ukupnaPovrsina += m.povrsina();
         }
         else
            System.out.println(" не подржава мерење |");
      }
      System.out.println("Укупан обим:     " + ukupanObim);
      System.out.println("Укупна површина: " + ukupnaPovrsina);

      Scanner sc = new Scanner(System.in);
      System.out.print("Unesite koordinate tacke X: ");
      Tacka x = new Tacka("X", sc.nextDouble(), sc.nextDouble());
      sc.close();

      Sadrzavanje[] sviSd = {ab, p, bcd, bcde, k1};
      System.out.println("Објекти који садрже дату тачку " + x + " су: " );
      for (Sadrzavanje sd : sviSd)
         if( sd.sadrzi(x))
            System.out.printf("%s ", sd);
   }
}
```

Један од примера извршавања програма је дат испод.

```
Геометријски објекти:
| A(14.5,15.0) конвексан  ограничен
 не подржава мерење |
| B(10.0,11.5) конвексан  ограничен
 не подржава мерење |
```

```
| C(10.45,22.0) конвексан  ограничен
 не подржава мерење |
| D(22.3,17.5) конвексан  ограничен
 не подржава мерење |
| E(25.0,25.5) конвексан  ограничен
 не подржава мерење |
| дAB:[A(14.5,15.0);B(10.0,11.5)] конвексан  ограничен
 не подржава мерење |
| пAB:[a=2.5;b=-7.800000000000001;c=80.75000000000001] конвексан  неограничен
 не подржава мерење |
| тBCD:[B(10.0,11.5);C(10.45,22.0);D(22.3,17.5)] конвексан  ограничен
 обим: 36.87069776990427 површина: 63.22500000000001|
| чBCDE:[B(10.0,11.5);C(10.45,22.0);D(22.3,17.5);E(25.0,25.5)] неконвексан  ограничен
 обим: 52.14692919616368 површина: 129.075|
| кB18:[B(10.0,11.5);18.0] конвексан  ограничен
 обим: 113.09733552923255 површина: 1017.8760197630929|
Укупан обим:     202.11496249530052
Укупна површина: 1210.1760197630929
Unesite koordinate tacke X: 15 16
Објекти који садрже дату тачку X(15.0,16.0) су:
тBCD:[B(10.0,11.5);C(10.45,22.0);D(22.3,17.5)] кB18:[B(10.0,11.5);18.0]
```

У првом делу тела метода `main()`, у циклусу за приказ података о геометријским објектима, је преко интерфејса обезбеђено да разнородни објекти буду третирани на униформни начин, као примерци одговарајућих интерфејса. Дакле, и за интерфејсе функционише полиморфизам, описан у секцији 8.5.5.■

9.2.5. Параметри типа интерфејса

Често се јавља потреба да параметар неког метода треба да буде таквог типа да га представљају различите класе, међусобно веома удаљене у хијерархији наслеђивања. У таквом случају, једна од могућности је да се параметри декларишу тако да буду типа интерфејса.

Пример 10. Надоградити претходни пример креирањем Јава метода `sadrziSeU()` у класи `Tacka` који за дату тачку проверава да ли се садржи у геометријском објекту прослеђеном као аргумент метода. Креирати одвојену класу са `main()` методом која ће користити метод `sadrziSeU()` да за тачку, чије су координате учитане са стандардног улаза, одреди све геометријске објекте којима та тачка не припада.□

Овде само треба додати метод у класу `Tacka` и направити нову класу са улазном тачком програма. Све остало, интерфејси `Mera`, `Oblik`, `Sadrzavanje` и `MeraOblikSadrzavanje`, апстрактна класа `GeometrijskiObjekat`, као и конкретне класе `Duz`, `Prava`, `Krug`, `Trougao` и `Cetvorougao` остају поптпуно исте као у претходном примеру и зато се неће овде поново наводити. Параметар метода `sadrziSeU()` треба да представља геометријски објекат, али не било какав геометријски објекат, већ само онај објекат за који има смисла да се проверава да ли му тачка припада или не – другим речима, објекат који имплементира интерфејс `Sadrzavanje`. Дакле, аргумент који се прослеђује методу `sadrziSeU()` треба да буде типа `Sadrzavanje`.

Програмски кôд за класу `Tacka` се налази у Јава датотеци `Tacka.java`.

```
package rs.math.oop.g09.p10.interfejsGeometrija;

public class Tacka extends GeometrijskiObjekat implements Oblik {
```

```
   private double x;
   private double y;

   public boolean sadrziSeU(Sadrzavanje objekat){
        return objekat.sadrzi(this);
   }
}
```

Реализација метода `sadrziSeU()` се своди на позив метода `sadrzi()` у оквиру интерфејса `Sadrzavanje`.

Метод `main()` се налази у Јава датотеци `MereObliciSadrzavanja.java`.

```
package rs.math.oopPitanjaZadaci.g09.p10.interfejsGeometrija;

import java.util.Scanner;

public class MereObliciSadrzavanja {
   public static void main(String[] args) {
      Tacka a = new Tacka("A", 14.5, 15);
      Tacka b = new Tacka("B", 10, 11.5);
      Tacka c = new Tacka( "C", 10.45, 22);
      Tacka d = new Tacka( "D", 22.3, 17.5);
      Tacka e = new Tacka("E", 25, 25.5);
      Duz ab = new Duz("дAB", a, b);
      Prava p = new Prava("пAB", a, d);
      Trougao bcd = new Trougao("тBCD", b, c, d);
      Cetvorougao bcde = new Cetvorougao("чBCDE", b, c, d, e);
      Krug k1 = new Krug("кB18", b, 18);

      Scanner sc = new Scanner(System.in);
      System.out.print("Унесите координате тачке X: ");
      Tacka x = new Tacka("X", sc.nextDouble(), sc.nextDouble());
      sc.close();
      Sadrzavanje[] sviSd = {ab, p, bcd, bcde, k1};
      System.out.println("\nОбјекти који не садрже дату тачку " + x + " су: " );
      for (Sadrzavanje sd : sviSd)
         if( !x.sadrziSeU(sd))
            System.out.println(sd);
    }
}
```

Следи један од примера извршавања програма.

```
Унесите координате тачке X: 15 16

Објекти који не садрже дату тачку X(15.0,16.0) су:
дAB:[A(14.5,15.0);B(10.0,11.5)]
пAB:[a=2.5;b=-7.800000000000001;c=80.75000000000001]
чBCDE:[B(10.0,11.5);C(10.45,22.0);D(22.3,17.5);E(25.0,25.5)]
```

Уочава се полиморфно понашање приликом реализације метода `sadrziSeU()` – начин реализације метода зависи од стварне класе објекта у времену извршавања и фундаментално се разликује у свакој новој итерацији.■

Пример 11. Дефинисати интерфејс `StekNiski` који садржи методе карактеристичне за стек структуру података: 1) додавање ниске на врх стека (енг. `push()`); 2) уклањање елемента са врха стека (енг. `pop()`) и 3) враћање тренутне величине стека (броја елемената).

Реализовати две имплементације овог интерфејса, једну која користи (енкапсулира) раније уведену динамичку структуру `SamorastuciNizNiski` (пример 23 из секције 8.9.1), и другу која користи раније уведену динамичку структуру `PovezanaListaNiski` (пример 24 из секције 8.9.2).
Демонстрирати употребу било које од ове две имплементације "сакривањем" имплементације иза интерфејса `StekNiski` – корисник може да одлучи коју ће имплементацију користити. □

```
package rs.math.oop.g09.p11.stekKaoInterfejs;

public interface StekNiski {
    void dodaj(String elem);
    String ukloni();
    int brojElemenata();
}
```

```
package rs.math.oop.g09.p11.stekKaoInterfejs;

public class StekNiskiPrekoNiza implements StekNiski {
    private SamorastuciNizNiski elementi;
    private int vrhSteka;

    {
        elementi = new SamorastuciNizNiski();
        vrhSteka = -1;
    }

    @Override
    public void dodaj(String elem) {
        elementi.postaviNa(++vrhSteka, elem);
    }

    @Override
    public String ukloni() {
        if (vrhSteka == -1) {
                System.err.println("Грешка при уклањању: стек је празан!");
                return "<нема>";
        }
        return elementi.uzmiSa(vrhSteka--);
    }

    @Override
    public int brojElemenata() {
        return (vrhSteka + 1);
    }
}
```

Прва имплементација стека се потпуно ослања на раније уведену имплементацију `SamorastuciNizNiski`, која динамички расте, па није потребно водити рачуна о реалокацији низа. Једина додатна информација је смештена у поље `vrhSteka` – она показује на позицију последњег додатог елемента па је иницијализована на вредност -1. На основу ње се у сваком моменту зна са које позиције је потребно преузети елемент или на коју позицију га је потребно поставити. На пример, након додавања једног елемента у стек, променљива за `vrhSteka` ће имати вредност 0.

```
package rs.math.oop.g09.p11.stekKaoInterfejs;

public class StekNiskiPrekoListe implements StekNiski {
    private PovezanaListaNiski elementi = new PovezanaListaNiski();

    @Override
    public void dodaj(String elem) {
        elementi.dodajNaPocetak(elem);
    }

    @Override
    public String ukloni() {
        String elem = elementi.ukloniSaPocetka();
        if (elem == null) {
               System.err.println("Грешка при уклањању: стек је празан!");
               return "<нема>";
        }
        return elem;
    }

    @Override
    public int brojElemenata() {
        String elem = elementi.uzmiPrvi();
        if (elem == null)
               return 0;
        int n = 0;
        while (elem != null) {
               n++;
               elem = elementi.uzmiSledeci();
        }
        return n;
    }
}
```

Друга имплементација интерфејса, под називом `StekNiskiPrekoListe`, ослоњена је на раније уведену класу `PovezanaListaNiski`. Овде су реализације метода за додавање и уклањање скоро директно засноване на методима `dodajNaPocetak()` и `ukloniSaPocetka()` класе `PovezanaListaNiski`. Рачунање броја елемената је нешто захтевније, јер `PovezanaListaNiski` не одржава ту информацију у виду поља, па је потребно проћи кроз све елементе листе и пребројати их.

```
package rs.math.oop.g09.p11.stekKaoInterfejs;

import java.util.Scanner;

public class CitajOkreni {
    private static void procitaj(StekNiski stek, Scanner sc) {
        System.out.println("Унеси текст (или реч КРАЈ ћирилицом за крај уноса):");
        while (sc.hasNext()) {
               String rec = sc.next();
               if (rec.equals("KPAJ"))
                      break;
               stek.dodaj(rec);
        }
    }
```

```
    private static void prikazi(StekNiski stek) {
        while (stek.brojElemenata() > 0)
               System.out.printf("%s\t", stek.ukloni());
        System.out.println();
    }

    public static void main(String[] args) {
        StekNiski stek = null;
        Scanner sc = new Scanner(System.in);
        System.out.print("Унеси слово - н (за низ) или л (за листу): ");
        char tip = sc.next().toLowerCase().trim().charAt(0);
        switch (tip) {
        case 'н':
        case 'n': {
               stek = new StekNiskiPrekoNiza();
               break;
        }
        case 'л':
        case 'l': {
               stek = new StekNiskiPrekoNiza();
               break;
        }
        default: {
               System.err.println("Валидни уноси су н и л!");
               System.exit(-1);
        }
        }
        procitaj(stek, sc);
        System.out.println("Број унесених речи: " + stek.brojElemenata());
        prikazi(stek);
        sc.close();
    }
}
```

Класа `CitajOkreni` демонстрира употребу претходне две имплементације стека. Корисник најпре бира коју имплементацију жели да користи, након чега се врши унос елемената. На крају се приказују сви елементи стека тако што се понавља следећи поступак док год се стек не испразни: 1) уклони елемент са врха и 2) прикажи га на конзоли.

Следе два примера извршавања.

```
Унеси слово - н (за низ) или л (за листу): н
Унеси текст (или реч КРАЈ ћирилицом за крај уноса):
Ово
је
садржај
стека
КРАЈ
Број унесених речи: 4
стека    садржај    је    Ово

Унеси слово - н (за низ) или л (за листу): л
Унеси текст (или реч КРАЈ ћирилицом за крај уноса):
Ово
је
садржај
```

```
стека
КРАЈ
Број унесених речи: 4
стека    садржај    је    Ово ■
```

9.3. Интерфејси у JDK-у

Као што је истакнуто, интерфејси се имплементирају од стране Јава класа. Могу се имплементрати интерфејси које је програмер осмислио, али и интерфејси које су осмислили креатори Јаве и који су у оквиру JDK испоручени заједно са програмским окружењем Јава. Ови интерфејси су повезани са честим процедурама у програмирању.

9.3.1. Сортирање, интерфејс Comparable

На почетку се разматра сортирање (уређење) низа. Оно се реализује коришћењем статичког метода `Arrays.sort()`.

Пример 12. Написати Јава програм за сортирање низа целих бројева, низа реалних бројева у покретном зарезу и низа ниски.□

Цео програм се налази у једној Јава датотеци `MereObliciSadrzavanja.java`.

```
package rs.math.oop.g09.p12.uredjenjePredefinisani;

import java.util.Arrays;

public class UredjenjePredefinisani {
    private static void prikaziCele(int[] celi) {
        for (int x : celi)
            System.out.printf("%d ", x);
        System.out.println();
    }

    private static void prikaziRealne(double[] realni) {
        for (double x : realni)
            System.out.printf("%f ", x);
        System.out.println();
    }

    private static void prikaziNiske(String[] niske) {
        for (String x : niske)
            System.out.printf("%s ", x);
        System.out.println();
    }

    public static void main(String[] args) {
        int[] celi = {12, 4, -3, 0, 17, 5};
        System.out.println("Пре сортирања");
        prikaziCele(celi);
        Arrays.sort(celi);
        System.out.println("После сортирања");
        prikaziCele(celi);

        double[] realni = {12.5, 4.7, -3.2e1, 0, +1.7e-1, 5};
        System.out.println("Пре сортирања");
```

```
        prikaziRealne(realni);
        Arrays.sort(realni);
        System.out.println("После сортирања");
        prikaziRealne(realni);

        String[] niske = {"12.5", "мики", "-3.2e1", "0", "+1.7e-1", "5", "паја"};
        System.out.println("Пре сортирања");
        prikaziNiske(niske);
        Arrays.sort(niske);
        System.out.println("После сортирања");
        prikaziNiske(niske);
    }
}
```

Прикази низа целих бројева, низа реалних бројева и низа ниски су издвојени у посебне процедуре `prikaziCele()`, `prikaziRealne()` и `prikaziNiske()`, респективно.
Извршавањем датог програма добија се следећи резултат:

```
Пре сортирања
12 4 -3 0 17 5
После сортирања
-3 0 4 5 12 17
Пре сортирања
12,500000 4,700000 -32,000000 0,000000 0,170000 5,000000
После сортирања
-32,000000 0,000000 0,170000 4,700000 5,000000 12,500000
Пре сортирања
12.5 мики -3.2e1 0 +1.7e-1 5 паја
После сортирања
+1.7e-1 -3.2e1 0 12.5 5 мики паја
```

Уочава се да су коришћењем метода `Arrays.sort()` бројеви сортирани према вредности, а ниске према лексикографском уређењу у растућем (прецизније неопадајућем) поретку.■
Обично испоручилац услуге тврди: ако класа имплементира конкретни интерфејс, ја ћу онда пружити услугу. Таква је ситуација са методом `Arrays.sort()` примењеном над нискама у претходном примеру. Метод ће коректно сортирати низ ниски, али под једним условом – елементи у низу морају сами знати како да се упореде, тј. ниске морају методу за сортирање обезбедити информацију који члан низа треба да иде напред, а који позади.
Одговорност за упоређивање, дакле, није на методу `Arrays.sort()`, већ на класи `String`. Дефинисан је уговор који обавезује класу `String` да упореди инстанцу класе `String` са неком другом ниском, а класа `Arrays` ће, у оквиру метода `sort()`, користити тај уговор ради уређења ниски.
Претходни уговор је дефинисан интерфејсом `Comparable`. Интерфејс се налази у пакету `java.lang`, па не треба наводити његов пун назив.
На овом нивоу знања, имајући у виду да још нису обрађени генерички типови (поглавље 13), може се сматрати да интерфејс `Comparable` има следећу структуру:

```
public interface Comparable {
       int compareTo(Object other);
}
```

Класа имплементира интерфејс `Comparable` ако садржи метод `compareTo()`. Наравно, метод `compareTo()` треба да буде у сагласности са методом `equals()` класе `Object`: ако `equals()` врати `true`, онда `compareTo()` врати `0` и обратно.

Пример 13. Написати Јава програм за сортирање низа тачака. Дата тачка треба да буде пре друге тачке ако је ближа координатном почетку, или ако су им растојања до координатног почетка иста, а она има мању y координату тј. мању ординату. □

Да би метод `Arrays.sort()` могао да се користи за сортирање низа тачака, потребно је да класа `Tacka` имплементира интерфејс `Comparable`. То даље значи да у класи `Tacka` мора да постоји метод `compareTo()` који ће вратити одговорајућу целу вредност – резулатат при поређењу објекта `this` и параметра тог метода.

Класа `Tacka` је изведана из апстрактне класе `GeometrijskiObjekat`, чији је кôд дат у наставку.

```
package rs.math.oop.g09.p13.uredjenjeTacaka;

public abstract class GeometrijskiObjekat {
   private String oznaka;

   public GeometrijskiObjekat(String oznaka) { this.oznaka = oznaka; }

   public String uzmiOznaku() { return oznaka; }

   public void postaviOznaku(String oznaka) { this.oznaka = oznaka; }
}
```

Програмски кôд за класу `Tacka` се налази у Јава датотеци `Tacka.java`.

```
package rs.math.oopPitanjaZadaci.g09.p13.uredjenjeTacaka;

public class Tacka extends GeometrijskiObjekat implements Comparable{
   private double x;
   private double y;

   @Override
   public int compareTo(Object obj) {
        if (!(obj instanceof Tacka))
             return -1;
        Tacka t = (Tacka) obj;
        Tacka o = new Tacka(0, 0);
        double razlika = rastojanje(o) - t.rastojanje(o);
        if (razlika < 0) return -1;
        if (razlika > 0) return 1;
        return (int)(y - t.y);
   }
}
```

У претходном коду је наведен само метод `compareTo()`. Остали методи класе `Tacka` (конструктори, методе за очитавање и постављање кооордината, за проверу једнакости са другом тачком, за одређивање хеш-кода и ниска-репрезентације, као и за одређивање растојања до друге тачке) нису овде приказани, јер су потпуно исти као у примеру 2.

У методу `compareTo`() се прво проверава тип параметра, па ако се не ради о тачки, онда се враће негативна вредност, јер тачка треба да буде пре нечег што није тачка. Ако параметар јесте тачка, тада се пореде растојања самог објекта и параметра до координатног почетка – ако та растојања нису иста, враћа се одговарајући резултат. Ако

су обе тачке једнако удаљене од координартног почетка, тада се пореде њихове y координате.

Улазна тачка програма се налази у Јава датотеци `UredjenjeTacaka.java`.

```
package rs.math.oop.g09.p13.uredjenjeTacaka;

import java.util.Arrays;

public class UredjenjeTacaka {
   private static void prikaziTacke(Tacka[] sve) {
      for (Tacka t: sve)
         System.out.printf("%s ", t);
      System.out.println();
   }

   public static void main(String[] args) {
      Tacka a = new Tacka("A", 14.5, 15);
      Tacka b = new Tacka("B", 10, 11.5);
      Tacka c = new Tacka("C", 10.45, 22);
      Tacka d = new Tacka("D", 22.3, 17.5);
      Tacka e = new Tacka("E", 25, 25.5);

      Tacka[] sve = {a, b, c, d, e};
      System.out.println("Пре сортирања");
      prikaziTacke(sve);

      Arrays.sort(sve);
      System.out.println("После сортирања");
      prikaziTacke(sve);
    }
}
```

Уочава се да је функционалност за приказ низа тачака издвојена у посебни метод. Извршавање програма доводи до следећег резултата.

```
Пре сортирања
A(14.5,15.0) B(10.0,11.5) C(10.45,22.0) D(22.3,17.5) E(25.0,25.5)
После сортирања
B(10.0,11.5) A(14.5,15.0) C(10.45,22.0) D(22.3,17.5) E(25.0,25.5)
```

Јасно је да су на почетку сортираног низа тачке ближе координатном почетку. ■

9.3.2. Вишекритеријумско сортирање, интерфејс Comparator

Веома често је потребно да у оквиру истог софтверског система колекције (поглавље 14) и низови елемената буду сортирани према различитим критеријумима. Претходно описани механизам сортирања допушта сортирање према једном критеријуму, где се користи статички метод `Arrays.sort()` са једним параметром (низом елемената датог типа) и где је одлучивање о редоследу два објекта током сортирања поверено имплементацији метода `compareTo()` у оквиру класе која представља тип елемента низа.

Међутим, метод `Arrays.sort()` је преоптерећен, па у JDK постоји и варијанта овог статичког метода са два параметра, где је први параметар низ, а други параметар је објекат који одређује редослед приликом поређења два елемената низа. Тај други параметар је типа интерфејса `Comparator` из пакета `java.util`.

Опет, имајући у виду да још нису обрађени генерички типови (поглавље 13), може се сматрати да интерфејс `Comparator` има следећу структуру:

```
public interface Comparator {
	int compare(Object o1, Object o2);
}
```

Интерфејс `Comparator` садржи само један метод са два параметра. Метод `compare()` треба реализовати тако да враћа негативан цео број ако објекат `o1` треба да буде испред објекта `o2`, нулу ако су `o1` и `o2` једнаки у смислу уређења и позитиван цео број ако `o2` треба да буде испоред `o1`.

Оно поређење које се реализује у класи која представља тип компоненте низа (а не у посебном објекту за поређење) и које бива коришћено од стране метода `Arrays.sort()` са једним параметром се назива подразумевано поређење.

Пример 14. Написати Јава програм за сортирање низа тачака по више критеријума (по локацији од ближих према даљим у односу на координатни почетак, по ознаци, као и по локацији од даљих према ближим). □

Реализација поређења на основу растуће удаљености тачке од координатног почетка (од ближих према даљим) представља подразумеван начин поређења објеката класе `Tacka`. Класа `Tacka` је изведена из апстрактне класе `GeometrijskiObjekat`, а како нема промена код ове две класе у односу на претходни пример, то програмски кôд ове две класе неће бити поново наведен.

Поређење по ознаци је реализовано у посебној класи `TackaOznakaComparator`. Програмски кôд за ову класу се налази у Јава датотеци `TackaOznakaComparator.java`.

```
package rs.math.oop.g09.p14.viseKriterijumaTacke;

import java.util.Comparator;

public class TackaOznakaComparator implements Comparator
{
	@Override
	public int compare( Object o1, Object o2 )
	{
		if( !(o1 instanceof Tacka ) ) return 1;
		if( !(o2 instanceof Tacka ) ) return -1;
		Tacka t1 = (Tacka) o1;
		Tacka t2 = (Tacka) o2;
		return t1.uzmiOznaku().compareTo( t2.uzmiOznaku() );
	}
}
```

Ова класа имплементира интерфејс `Comparator` и садржи реализацију метода `compare()` са одговарајућим потписом. Ту се, у основи, поређење тачака своди на поређење ниски које представљају ознаке тих тачака.

Програмски кôд за класу `TackaPozicijaComparator` се налази у Јава датотеци `TackaPozicijaComparator .java`.

```
package rs.math.oop.g09.p14.viseKriterijumaTacke;

import java.util.Comparator;

public class TackaPozicijaComparator implements Comparator
{
```

```
        @Override
        public int compare( Object o1, Object o2 )
        {
                if (!(o1 instanceof Tacka)) return 1;
                if (!(o2 instanceof Tacka)) return -1;
                Tacka t1 = (Tacka) o1;
                Tacka t2 = (Tacka) o2;
                Tacka o = new Tacka( 0, 0 );
                double d = o.rastojanje( t2 ) - o.rastojanje( t1 );
                if (d > 0) return 1;
                if (d < 0) return -1;
                return (int)(t2.uzmiX() - t1.uzmiX());
        }
}
```

Улазна тачка програма се налази у Јава датотеци `UredjenjeTacaka.java`.

```
package rs.math.oop.g09.p14.viseKriterijumaTacke;

import java.util.Arrays;

public class UredjenjeTacaka {
   private static void prikaziTacke(Tacka[] sve) {
      for (Tacka t: sve)
         System.out.printf("%s ", t);
      System.out.println();
   }

   public static void main(String[] args) {
      Tacka a = new Tacka("A", 14.5, 15);
      Tacka b = new Tacka("B", 10, 11.5);
      Tacka c = new Tacka("C", 10.45, 22);
      Tacka d = new Tacka("D", 22.3, 17.5);
      Tacka e = new Tacka("E", 25, 25.5);

      Tacka[] sve = {a, b, c, d, e};
      System.out.println("Пре сортирања");
      prikaziTacke(sve);

      Arrays.sort(sve);
      System.out.println("После подразумеваног сортирања");
      prikaziTacke(sve);

      Arrays.sort(sve, new TackaOznakaComparator());
      System.out.println("После сортирања по ознаци тачке");
      prikaziTacke(sve);

      Arrays.sort(sve, new TackaPozicijaComparator());
      System.out.println("После сортирања по позицији тачке");
      prikaziTacke(sve);
   }
}
```

У програмском коду није поново навођен метод за приказ низа тачака `prikaziTacke()`, јер је тај метод исти као у претходном примеру. Овде је три пута позван метод за сортирање: први пут је сортирање извршено на подразумевани начин, док су за потребе

сортирања у преостала два коришћене претходно приказане имплементације интерфејса `Comparator`.

Извршавање овог програма доводи до следећег резултата.

```
Пре сортирања
A(14.5,15.0) B(10.0,11.5) C(10.45,22.0) D(22.3,17.5) E(25.0,25.5)
После подразумеваног сортирања
B(10.0,11.5) A(14.5,15.0) C(10.45,22.0) D(22.3,17.5) E(25.0,25.5)
После сортирања по ознаци тачке
A(14.5,15.0) B(10.0,11.5) C(10.45,22.0) D(22.3,17.5) E(25.0,25.5)
После сортирања по позицији тачке
E(25.0,25.5) D(22.3,17.5) C(10.45,22.0) A(14.5,15.0) B(10.0,11.5)■
```

Додатни критеријуми за уређење при сортирању се могу дефинисати и за класе испоручене у JDK. У примеру 11 је јасно показано да је подразумевано уређење за целе бројеве на основу вредности од мањег према већем, а да је подразумевано уређење за ниске лексикографско. Међутим, то не ограничава програмера да сортира низове бројева и ниски и по неким другим критеријумима.

Пример 15. Написати Јава програм за сортирање низа целих бројева на два различита начина: 1) сортирање у неопадајућем поретку по вредности и 2) сортирање тако да прво буду сви парни бројеви сортирани у неопадајућем поретку, а потом непарни бројеви сортирани у неопадајућем поректу. Програм треба и да сортира низ ниски: 1) лексикографски и 2) тако да предност имају оне ниске које садрже више самогласника, а ако ниске имају исти број самогласника онда предност треба да има краћа ниска. Сортирање реализовати помоћу метода `sort()` класе `Arrays`.□

Поред подразумеваних уређења за класе `Integer` и `String`, потребно је обезбедити и два додатна начина поређења, да би се поређење реализовало онако како је дефинисано у примеру. Дакле, потребно је дефинисати две класе које имплементирају интерфејс `Comparator`.

Прва се односи на поређење целих бројева. Програмски кôд за класу `ParNeparComparator` се налази у Јава датотеци `ParNeparComparator.java`.

```
package rs.math.oop.g09.p15.viseKriterijumaPredefinisani;

import java.util.Comparator;

class ParNeparComparator implements Comparator {
    @Override
    public int compare(Object o1, Object o2) {
        if (!(o1 instanceof Integer)) return 1;
        if (!(o2 instanceof Integer)) return -1;
        int i1 = ((Integer) o1).intValue();
        int i2 = ((Integer) o2);
        if (i1 % 2 == 0) {
            if (i2 % 2 == 0) return i1 - i2;
            else return -1;
        } else {
            if (i2 % 2 == 0) return 1;
            else return i1 - i2;
        }
    }
}
```

У програмском коду се јасно уочавају четири могућа исхода за парност-непарност два цела броја и јасно је како се у сваком од та четири случаја добија резултат поређења.

Друга се односи на поређење ниски. Програмски кôд за ову класу се налази у Јава датотеци `BrojSamoglasnikaComparator.java`.

```
package rs.math.oop.g09.p15.viseKriterijumaPredefinisani;

import java.util.Comparator;

public class BrojSamoglasnikaComparator implements Comparator {
    private int brojSamoglasnika(String s) {
       int ret = 0;
       for (char ch : s.toCharArray())
           if ("aeiouAEIOUаеиоуАЕИОУ".indexOf(ch) >= 0)
               ret++;
        return ret;
    }

    @Override
    public int compare(Object o1, Object o2) {
       if (!(o1 instanceof String)) return 1;
       if (!(o2 instanceof String)) return -1;
       int razlika = brojSamoglasnika((String) o2) - brojSamoglasnika((String) o1);
       if (razlika != 0) return razlika;
       return ((String) o1).length() - ((String) o2).length();
    }
}
```

Овде је одређивање броја самогласника у ниски издвојено у помоћну функцију. Прво се упоређују бројеви самогласника, па тек онда (ако има потребе) дужине ниски.

Улазна тачка програма је класа `VisekriterijumskoSortiranje`, која се налази у Јава датотеци `VisekriterijumskoSortiranje.java`.

```
package rs.math.oop.g09.p15.viseKriterijumaPredefinisani;

import java.util.Arrays;
import java.util.Comparator;

public class VisekriterijumskoSortiranje
{
    private static void prikazi(Integer[] niz) {
       for (int x : niz)
           System.out.print(x + " ");
       System.out.println();
    }

    private static void prikazi(String[] niz) {
       for (String x: niz)
           System.out.printf("%s ", x );
       System.out.println();
    }

    public static void main( String[] args )
    {
       Integer[] celiBrojevi = {-3, 24, -2, 1, 2, 0, 3, 4, -40};
       System.out.println("Пре сортирања:");
       prikazi(celiBrojevi);
       Arrays.sort(celiBrojevi);
       System.out.println("После подразуменваног сортирања (по вредности):");
```

```
        prikazi(celiBrojevi);
        Arrays.sort(celiBrojevi, new ParNeparComparator());
        System.out.println("После пар-непар сортирања:");
        prikazi(celiBrojevi);

        String[] niske = { "Паја", "Мики", "Шиља", "Аладин", "Снежана",
                           "Херкулес",   "Ариел", "Мандрак", "Банана-Мен" };
        System.out.printf( "%nПре сортирања:%n");
        prikazi(niske);
        Arrays.sort( niske );
        System.out.println( "После подразумеваног сортирања (лексикографски)" );
        prikazi(niske);
        Comparator c = new BrojSamoglasnikaComparator();
        Arrays.sort( niske, c );
        System.out.println( "После сортирања по броју самогласника" );
        prikazi(niske);
    }
}
```

Преоптерећени приватни метод `prikazi()` служи за приказ низова. И код сортирања бројева и код сортирања ниски, прво сортирање је извршено на подразумевани начин, а у другом је коришћен посебни објекат који дефинише начин поређења. Као што се може видети, приликом неподразумеваног поређења ниски, објекат који дефинише начин поређења се може посматрати као инстанца интерфејса `Comparator`.
Извршавање програма доводи до следећег резултата.

```
Пре сортирања:
-3 24 -2 1 2 0 3 4 -40
После подразумеваног сортирања (по вредности):
-40 -3 -2 0 1 2 3 4 24
После пар-непар сортирања:
-40 -2 0 2 4 24 -3 1 3

Пре сортирања:
Паја Мики Шиља Аладин Снежана Херкулес Ариел Мандрак Банана-Мен
После подразумеваног сортирања (лексикографски)у
Аладин Ариел Банана-Мен Мандрак Мики Паја Снежана Херкулес Шиља
После сортирања по броју самогласника
Банана-Мен Ариел Аладин Снежана Херкулес Мики Паја Шиља Мандрак
```

Ослањањем на класе и интерфејсе из JDK, малим променама метода који реализују неки интерфејс (у овом случају интерфејс `Comparator`), могу се реализовати и неке веома комплексне функционалности. ■

9.3.4. Клонирање објеката, интерфејс Cloneable

Јасно је да постоји потреба креирања копије неког већ направљеног објекта. Један од начина да се то постигне је да се користе копирајући конструктори, описани у секцији 8.6.2. Други начин за креирање копије већ направљеног објекта, заснива се на механизму клонирања.
Класа `Object` садржи метод `clone()`, који се може искористити за прављење копије, тј. клона датог објекта. Метод `clone()`, у класи `Object`, је проглашен заштићеним, тј. означен је кључном речју `protected`, што обично значи да он није потпуно спреман за

коришћење, већ да се у класи, која превазилази тај метод, морају реализовати још неке додатне активности да се би метод могао користити.
Иако је метод за клонирање дефинисан у најопштијој класи, то не значи да свака од класа, коју креира програмер, треба да превазилази метод за клонирање, већ то треба да се ради само у класама за које програмер одлучи да треба да подрже клонирање. Програмер ће сигнализирати да дата класа подржава клонирање својих инстанци тако што ће написати да та класа имплементира интерфејс `Cloneable`. Овај интерфејс је дефинисан на следећи начин:

```
public interface Cloneable { }
```

Као што се види, сам интерфејс не садржи метод за реализацију клонирања, јер је тај метод дефинисан у класи `Object`. Интерфејси који не садрже методе служе за означавање да класе које их имплементирају поседују неко својство, па се још називају и интерфејси означавања или маркерски интерфејси. У овом случају, интерфејс `Cloneable` служи за означавање да она класа која га имплементира поседује могућност клонирања својих инстанци.
Пример 16. Написати Јава програм где се клонирају примерци класе која представља запосленог и испитује понашање оригиналног и клонираног објекта.□
Програмски кôд за класу `Zaposleni` се налази у Јава датотеци `Zaposleni.java`.

```
package rs.math.oop.g09.p16.kloniranje;

public class Zaposleni implements Cloneable {
   private String ime;
   private String prezime;
   private String opisPosla;
   private double plata;

   public Zaposleni(String ime, String prezime, String opisPosla, double plata) {
      this.ime = ime;
      this.prezime = prezime;
      this.opisPosla = opisPosla;
      this.plata = plata;
   }

   public Zaposleni(){ this("","", "", 0); }

   public void postaviPrezime(String prezime) { this.prezime = prezime; }

   public void povecajPlatu(double zaProcenat) {
      double iznosPovisice = plata * zaProcenat / 100;
      this.plata += iznosPovisice;
   }

   @Override
   public String toString() {
 '" + opisPosla
            + "', плата: " + plata + "]";
   }

   @Override
   public Zaposleni clone() throws CloneNotSupportedException {
      Zaposleni klonirani = (Zaposleni) super.clone();
      return klonirani;
```

```
    }
}
```

Класа `Zaposleni` имплементира интерфејс `Cloneable`, што значи да се на инстанце те класе може применити метод `clone()`.

Клонирање запосленог, тј. превазилажење метода `clone()`, се у овој класи свело на позив метода `clone()` класе `Object` уз конверзију новокреираног клона у тип `Zaposleni`.

Уочава се да метод `clone()` у класи `Zaposleni` није заштићен, него је јаван.

Улазна тачка програма се налази у Јава датотеци `Kloniranje.java`.

```
package rs.math.oop.g09.p16.kloniranje;

public class Kloniranje {
   public static void main(String[] args) throws CloneNotSupportedException{
         Zaposleni original = new Zaposleni("Јован", "Петровић",
                                    "приправник", 30_000);
         Zaposleni klon = original.clone();
         System.out.println("После клонирања, пре промена:");
         System.out.println("оригинал = " + original);
         System.out.println("клон =     " + klon);

         original.postaviPrezime("Станишић");
         System.out.println("После постављања презимена оригинала на 'Станишић'");
         System.out.println("оригинал = " + original);
         System.out.println("клон =     " + klon);

         klon.povecajPlatu(10);
         System.out.println("После повећања плате клона за 10%");
         System.out.println("оригинал = " + original);
         System.out.println("клон =     " + klon);
   }
}
```

Извршавање овог програма доводи до следећег резултата.

```
После клонирања, пре промена:
оригинал = [име: Јован Петровић, посао: 'приправник', плата: 30000.0]
клон =     [име: Јован Петровић, посао: 'приправник', плата: 30000.0]
После постављања презимена оригинала на 'Станишић'
оригинал = [име: Јован Станишић, посао: 'приправник', плата: 30000.0]
клон =     [име: Јован Петровић, посао: 'приправник', плата: 30000.0]
После повећања плате клона за 10%
оригинал = [име: Јован Станишић, посао: 'приправник', плата: 30000.0]
клон =     [име: Јован Петровић, посао: 'приправник', плата: 33000.0]
```

Уочава се да се оригинал и клон овде понашају као референцијално независни ентитети, како и треба да буде.■

Реализација метода у класи `Object` доводи до тзв. плитког копирања, које је ефикасно, и ту нема споредног ефекта у случају да су поља објекта, који се клонира, примитивног типа или имутабилна – то је био случај у претходном примеру.

Међутим, у случају када објекат, који се клонира, садржи мутабилна објектна поља, тада плитко клонирање доводи до непријатног споредног ефекта – тзв. „везивања“ оригинала и клона (попут референцијалне зависности о којој је било речи у секцији 8.6.2).

Пример 17. Написати Јава програм за клонирање инстанци класе која представља запосленог и садржи мутабилна поља. Испитати понашање оригиналног и клонираног објекта.□

У овом случају запослени садржи све податке као и у пертходном примеру. Међутим, овде он нема једноставну структуру, већ су његова поља груписана у два ентитета – део који се односи на опште податке о запосленом и део који се односи на податке о раду запосленог.

Општи подаци о запосленом су енкапсулирани у класи `Generalije`. Програмски кôд за класу `Generalije` се налази у Јава датотеци `Generalije.java`.

```
package rs.math.oop.g09.p17.kloniranjeMutabilnaPoljaProblem;

import java.time.LocalDate;

public class Generalije {
   private String ime;
   private String prezime;
   private String nadimak;
   private LocalDate datumRodjenja;

   public Generalije(String ime, String prezime, String nadimak,
                     LocalDate datumRodjenja) {
      this.ime = ime;
      this.prezime = prezime;
      this.nadimak = nadimak;
      this.datumRodjenja = datumRodjenja;
   }

   public Generalije(String ime, String prezime) {
      this(ime, prezime, "", LocalDate.of(1970, 1, 1));
   }

   public String uzmiIme() { return ime; }

   public void postaviIme(String ime) { this.ime = ime; }

   public String uzmiPrezime() { return prezime; }

   public void postaviPrezime(String prezime) { this.prezime = prezime; }

   public String uzmiNadimak() { return nadimak; }

   public void postaviNadimak(String nadimak) { this.nadimak = nadimak; }

   public LocalDate uzmiDatumRodjenja() { return datumRodjenja; }

   public void postaviDatumRodjenja(LocalDate datumRodjenja) {
      this.datumRodjenja = datumRodjenja;
   }
}
```

Овакве класе, које садрже само поља, конструкторе и методе за очитавање и постављање вредности поља се означавају акронимом POJO – Plain Old Java Object (срп. обичан стари јава објекат).

Информације о радном односу су садржане у класи `RadniOdnos`. Програмски кôд за ову класу се налази у Јава датотеци `RadniOdnos.java`.

```
package rs.math.oop.g09.p17.kloniranjeMutabilnaPoljaProblem;
```

```
import java.time.LocalDate;

public class RadniOdnos {
   private String opisPosla;
   private double plata;
   private LocalDate datumZaposlenja;

   public RadniOdnos(String opisPosla, double plata, LocalDate datumZaposlenja) {
      this.opisPosla = opisPosla;
      this.datumZaposlenja = datumZaposlenja;
      this.plata = plata;
   }

   public RadniOdnos(String opisPosla, double plata) {
      this(opisPosla, plata, LocalDate.of(2000, 1, 1));
   }

   public RadniOdnos(String opisPosla) { this(opisPosla, 300); }

   public String uzmiOpisPosla() { return opisPosla; }

   public void postaviOpisPosla(String opisPosla) { this.opisPosla = opisPosla; }

   public double uzmiPlatu() { return plata; }

   public void postaviPlatu(double plata) { this.plata = plata; }

   public LocalDate uzmiDatumZaposlenja() { return datumZaposlenja; }

   public void postaviDatumZaposlenja(LocalDate datumZaposlenja) {
      this.datumZaposlenja = datumZaposlenja;
   }
}
```

Програмски кôд за класу `Zaposleni` се налази у Јава датотеци `Zaposleni.java`.

```
package rs.math.oop.g09.p17.kloniranjeMutabilnaPoljaProblem;

import java.time.LocalDate;

public class Zaposleni implements Cloneable {
   private Generalije generalije;
   private RadniOdnos radniOdnos;

   public Zaposleni(String ime, String prezime, String opisPosla, double plata) {
      generalije = new Generalije(ime, prezime);
      radniOdnos = new RadniOdnos(opisPosla, plata);
   }

   public void postaviPrezime(String prezime){ generalije.postaviPrezime(prezime); }

   public void postaviDatumZaposlenja(int godina, int mesec, int dan) {
      LocalDate noviDatumZaposlenja = LocalDate.of(godina, mesec, dan);
      radniOdnos.postaviDatumZaposlenja(noviDatumZaposlenja);
   }

   public void povecajPlatu(double zaProcenat) {
```

```
        double iznosPovisice = radniOdnos.uzmiPlatu() * zaProcenat / 100;
        radniOdnos.postaviPlatu( radniOdnos.uzmiPlatu() + iznosPovisice);
    }

    @Override
    public String toString() {
        return "[име: " + generalije.uzmiIme() + " " + generalije.uzmiPrezime()
              + ", плата: " + radniOdnos.uzmiPlatu()
              + ", посао: '" + radniOdnos.uzmiOpisPosla()
              + "', запослен од: " + radniOdnos.uzmiDatumZaposlenja() + "]";
    }

    @Override
    public Zaposleni clone() throws CloneNotSupportedException {
        Zaposleni klonirani = (Zaposleni) super.clone();
        return klonirani;
    }
}
```

Исто као у претходном примеру, класа `Zaposleni` имплементира интерфејс `Cloneable`. Само клонирање је реализовано на потпуно исти начин као у претходном примеру. Међутим, у овом случају класа `Zaposleni` садржи мутирајућа поља `generalije` и `radniOdnos`.

Улазна тачка програма се налази у Јава датотеци `Kloniranje.java` и њен кôд је потпуно исти као у претходном примеру па стога неће бити поново навођен. Извршавање овог програма сада доводи до следећег резултата.

```
После клонирања, пре промена:
оригинал = [име: Јован Петровић, плата: 30000.0, посао: 'приправник', запослен од: 2017-
02-14]
клон =     [име: Јован Петровић, плата: 30000.0, посао: 'приправник', запослен од: 2017-
02-14]
После постављања презимена оригинала на 'Станишић'
оригинал = [име: Јован Станишић, плата: 30000.0, посао: 'приправник', запослен од: 2017-
02-14]
клон =     [име: Јован Станишић, плата: 30000.0, посао: 'приправник', запослен од: 2017-
02-14]
После повећања плате клона за 10%
оригинал = [име: Јован Станишић, плата: 33000.0, посао: 'приправник', запослен од: 2017-
02-14]
клон =     [име: Јован Станишић, плата: 33000.0, посао: 'приправник', запослен од: 2017-
02-14]
После постављања датума запослења оригинала на  01.04.2020
оригинал = [име: Јован Станишић, плата: 33000.0, посао: 'приправник', запослен од: 2020-
04-01]
клон =     [име: Јован Станишић, плата: 33000.0, посао: 'приправник', запослен од: 2020-
04-01]
```

Наредба којом се мења неко поље оригинала доводи до тога да одговарајуће поље клона буде постављено на ту исту вредност, иако то није тражено. Надаље, наредба којом се мења неко поље клона доводи до тога да буде ажурирано одговарајуће поље оригинала – што представља проблем.■

Проблем из претходног примера се може превазићи тако што ће се уместо плитког користити дубоко клонирање за сва мутабилна поља класе. Дакле, биће потребно у самој класи, на мало другачији (не тако директан), превазићи метод `clone()`.

Предложено превазилажење метода `clone()`, у ситуацији када оригинал садржи мутабилна поља, има следеће кораке:

1. направи клон сваког од мутаблиних поља'
2. направи клон самог објекта, позивом метода `clone()` из класе `Object`;
3. повежи мутабилна поља клонираног објекта из корака 2. са клоновима мутаблиних поља из корака 1.

Наравно, поступак се спроводи рекурзивно: када мутабилно поље из корака 1. у себи садржи мутабилно поље/поља, тада се клон тог поља прави по овом истом поступку, а не простим позивом `clone()` из класе `Object`.

Пример 18. Написати Јава програм за клонирање инстанци класе `Zaposleni` која садржи мутабилна поља и испитује понашање оригиналног и клонираног објекта.□

У овом случају запослени нема једноставну структуру, већ садржи два мутабилна поља, али је клонирање реализовано на адекватан начин. Наиме, сада ће клонирању запосленог претходити клонирање његових мутабилних поља.

Општи подаци о запосленом су енкапсулирани у класи `Generalije`. Програмски кôд за класу `Generalije` се налази у Јава датотеци `Generalije.java`.

```
package rs.math.oop.g09.p18.kloniranjeMutabilnaPoljaResenje;

import java.time.LocalDate;

public class Generalije  implements Cloneable {
   private String ime;
   private String prezime;
   private String nadimak;
   private LocalDate datumRodjenja;

   public Generalije(String ime, String prezime, String nadimak,
                     LocalDate datumRodjenja) {
      this.ime = ime;
      this.prezime = prezime;
      this.nadimak = nadimak;
      this.datumRodjenja = datumRodjenja;
   }

   public Generalije(String ime, String prezime) {
      this(ime, prezime, "", LocalDate.of(1970, 1, 1));
   }

   public String uzmiIme() { return ime; }

   public void postaviIme(String ime) { this.ime = ime; }

   public String uzmiPrezime() { return prezime; }

   public void postaviPrezime(String prezime) { this.prezime = prezime; }

   public String uzmiNadimak() { return nadimak; }

   public void postaviNadimak(String nadimak) { this.nadimak = nadimak; }

   public LocalDate uzmiDatumRodjenja() { return datumRodjenja; }

   public void postaviDatumRodjenja(LocalDate datumRodjenja) {
```

```
      this.datumRodjenja = datumRodjenja;
   }

   @Override
   public Generalije clone() throws CloneNotSupportedException {
      return (Generalije)super.clone();
   }
}
```

Како ће у оквиру клонирања запосленог бити потребно да се клонирају генералије, то ова класа имплементира интерфејс `Cloneable` и превазилази метод `clone()`. Сва поља ове класе су имутабилна, па је стога клонирање реализовано као плитко клонирање – директно је позван метод `clone()` класе `Object`.

Информације о радном односу су садржане у класи `RadniOdnos`. Програмски кôд за ову класу се налази у Јава датотеци `RadniOdnos.java`.

```
package rs.math.oop.g09.p18.kloniranjeMutabilnaPoljaResenje;

import java.time.LocalDate;

public class RadniOdnos implements Cloneable{
   private String opisPosla;
   private double plata;
   private LocalDate datumZaposlenja;

   public RadniOdnos(String opisPosla, double plata, LocalDate datumZaposlenja) {
      this.opisPosla = opisPosla;
      this.datumZaposlenja = datumZaposlenja;
      this.plata = plata;
   }

   public RadniOdnos(String opisPosla, double plata) {
      this(opisPosla, plata, LocalDate.of(2000, 1, 1));
   }

   public RadniOdnos(String opisPosla) { this(opisPosla, 300); }

   public String uzmiOpisPosla() { return opisPosla; }

   public void postaviOpisPosla(String opisPosla) { this.opisPosla = opisPosla; }

   public double uzmiPlatu() { return plata; }

   public void postaviPlatu(double plata) { this.plata = plata; }

   public LocalDate uzmiDatumZaposlenja() { return datumZaposlenja; }

   public void postaviDatumZaposlenja(LocalDate datumZaposlenja) {
      this.datumZaposlenja = datumZaposlenja;
   }

   @Override
   public RadniOdnos clone() throws CloneNotSupportedException {
      return (RadniOdnos) super.clone();
   }
}
```

И ова класа, из истих разлога као и претходна, подржава клонирање својих инстанци. С обзиром да ова класа садржи само имутабилна поља, код ње је клонирање такође реализовано као плитко клонирање.
Програмски кôд за класу `Zaposleni` се налази у Јава датотеци `Zaposleni.java`.

```
package rs.math.oop.g09.p18.kloniranjeMutabilnaPoljaResenje;

import java.time.LocalDate;

public class Zaposleni implements Cloneable {
   private Generalije generalije;
   private RadniOdnos radniOdnos;

   public Zaposleni(String ime, String prezime, String opisPosla, double plata) {
      generalije = new Generalije(ime, prezime);
      radniOdnos = new RadniOdnos(opisPosla, plata);
   }

   public void postaviPrezime(String prezime){
      generalije.postaviPrezime(prezime);
   }

   public void postaviDatumZaposlenja(int godina, int mesec, int dan) {
      LocalDate noviDatumZaposlenja = LocalDate.of(godina, mesec, dan);
      radniOdnos.postaviDatumZaposlenja(noviDatumZaposlenja);
   }

   public void povecajPlatu(double zaProcenat) {
      double iznosPovisice = radniOdnos.uzmiPlatu() * zaProcenat / 100;
      radniOdnos.postaviPlatu( radniOdnos.uzmiPlatu() + iznosPovisice);
   }

   @Override
   public String toString() {
      return "[име: " + generalije.uzmiIme() + " " + generalije.uzmiPrezime()
            + ", плата: " + radniOdnos.uzmiPlatu()
            + ", посао: '" + radniOdnos.uzmiOpisPosla()
            + "', запослен од: " + radniOdnos.uzmiDatumZaposlenja() + "]";
   }

   @Override
   public Zaposleni clone() throws CloneNotSupportedException {
      Generalije klonZaGeneralije = generalije.clone();
      RadniOdnos klonZaRadniOdnos = radniOdnos.clone();

      Zaposleni klonirano = (Zaposleni) super.clone();

      klonirano.generalije = klonZaGeneralije;
      klonirano.radniOdnos = klonZaRadniOdnos;
      return klonirano;
   }

}
```

Исто као у претходном примеру, класа `Zaposleni` имплементира интерфејс `Cloneable`. Међутим, клонирање је сада реализовано као дубоко клонирање.

И у овом примеру је улазна тачка програма у Јава датотеци `Kloniranje.java`, чији је садржај исти као у претходна два примера, па се неће овде наводити. Извршавање програма на стандардном излазу приказује следеће резултате.

```
После клонирања, пре промена:
оригинал = [име: Јован Петровић, плата: 30000.0, посао: 'приправник', запослен од: 2017-
02-14]
клон =     [име: Јован Петровић, плата: 30000.0, посао: 'приправник', запослен од: 2017-
02-14]
После постављања презимена оригинала на 'Станишић'
оригинал = [име: Јован Станишић, плата: 30000.0, посао: 'приправник', запослен од: 2017-
02-14]
клон =     [име: Јован Петровић, плата: 30000.0, посао: 'приправник', запослен од: 2017-
02-14]
После повећања плате клона за 10%
оригинал = [име: Јован Станишић, плата: 30000.0, посао: 'приправник', запослен од: 2017-
02-14]
клон =     [име: Јован Петровић, плата: 33000.0, посао: 'приправник', запослен од: 2017-
02-14]
После постављања датума запослења оригинала на  01.04.2020
оригинал = [име: Јован Станишић, плата: 30000.0, посао: 'приправник', запослен од: 2020-
04-01]
клон =     [име: Јован Петровић, плата: 33000.0, посао: 'приправник', запослен од: 2017-
02-14]
```

Уочава се да, иако класа садржи мутирајућа поља, оригинал и клон се понашају као референцијално независни ентитети, како и треба да буде.■

9.4. Принципи и препоруке објектно орјентисаног дизајна

По успешном овладавању програмским конструкцијама које се односе на апстрактне класе и интерфејсе, потребно је из другачије перспективе размотрити како се могу ове конструкције оптимално користити у развоју софтвера.

У ту сврху су, током протеклих деценија развоја програмирања, дефинисани разноврсни прицпи и препоруке. Предложени принципи доброг програмирања суштински представљају један начин формулисања шта треба радити а шта не, и они се поклапају са интуитивним осећајем за добар и лош програмски кôд које искуснији програмер развија током своје каријере. Наравно, те принципе не треба посматрати чврсто као математичке аксиоме, већ као мапу која олакшава налажење у путу – начин њихове примене зависи од контекста, тј. од карактеристика проблема који се решава, знања програмера, карактеристика програмерских алата који се користе и слично.

Ради лакшег памћења, обично се принцими доброг програмирања означавају акронимима. На пример, принцип да приликом програмирања не треба дуплирати исте елементе се назива DRY – Don’t Repeat Yourself (срп. немој да се понављаш), а принцип који промовише једноставност се означава акронимом KISS – Keep It Simple, Stupid (срп. нека остане једноставно, блесави).

Овде ће додатна пажња бити посвећена групи принципа доброг објектно оријентисаног програмирања која се означава акронимом SOLID.

9.4.1. SOLID принципи

Група принципа SOLID се обично приписује Роберту Мартину (познатом и под надимком "Ујка Боб"), Иако он није осмислио свих пет SOLID принципа, његов ангажман је утицао на то да буду формулисани у баш овом облику.
Свако од слова у акроним SOLID означава тачно један принцип објектно оријентисаног дизајна:

- S – Single responsibility principle (срп. принцип јединствене одговорности);
- O – Open-closed principle (срп. принцип отворености и затворености);
- L – Liskov substitution principle (срп. принцип заменљивости Лискова);
- I – Interface segregation principle (срп. принцип раздвајања интерфејса);
- D – Dependency inversion principle (срп. принцип инверзије зависности).

Принцип јединствене одговорности

Принцип јединствене одговорности гласи: свака класа треба да има тачно једну одговорност.
Према овом принципу, може постојати само један разлог због којег класа треба да буде промењена. Другим речима, класа треба да представља решење само једног задатка. Када се поштује овај принцип, тестирање је једноставније, што је у складу са агилним методологијама развоја софтвера, јако популарним у последње време.
Мање функционалности у једној класи такође значи да има мање зависности од осталих класа, што доводи до боље организације кода, јер се мање класе, са јасном сврхом, могу лакше пронаћи.
Пример 19. Креирати класу за рад са подацима о запосленима, где ће функционалност која подржава упис информација о запосленом у базу података да буде реализована у оквиру класе која представља запосленог.□
Као што је и захтевано у примеру, сва функционалност која се односи на запосленог је смештена у једну класу. Имајући у виду да рад са базама података у Јави излази из опсега овог уџбеника, функционалност за чување података у бази података није реализована у овом примеру, већ су само приказане одговарајуће поруке на излазу, и наведени коментари.
Програмски кôд за класу `Zaposleni` се налази у Јава датотеци `Zaposleni.java`.

```
package rs.math.oop.g09.p19.losPrincipS;

public class Zaposleni{
   private String ime;
   private String prezime;
   private String opisPosla;
   private double plata;

   public Zaposleni(String ime, String prezime, String opisPosla, double plata) {
      this.ime = ime;
      this.prezime = prezime;
      this.opisPosla = opisPosla;
      this.plata = plata;
   }

   public Zaposleni(){ this("","", "", 0); }

   public String uzmiIme(){ return ime; }
```

```
    public void postaviIme(String ime) { this.ime = ime; }

    public String uzmiPrezime(){ return prezime; }

    public void postaviPrezime(String prezime) { this.prezime = prezime; }

    public String uzmiOpisPosla(){ return opisPosla; }

    public void postaviOpisPosla(String opisPosla) { this.opisPosla = opisPosla; }

    public double uzmiPlatu(){ return plata; }

    public void postaviPlatu(double plata) { this.plata = plata; }

    @Override
    public String toString() {
        return "[име: " + ime + " " + prezime + ", посао: '" + opisPosla
                + "', плата: " + plata + "]";
    }

    public int sacuvajBazaPodataka(){
        System.out.println("Реализује чување информација о примерку класе '"
                + this.getClass().getSimpleName() + "' у бази података.");
        System.out.println("Чувају се следећи подаци: " + this);
        // овде треба да буду наредбе за чување објекта у базу
        return 0;
    }
}
```

Дакле, у оквиру класе `Zaposleni` се налази и POJO функционалност (приступ пољима објекта) и функционалност за чување података инстанце запосленог у бази, што нарушава принцип јединствене одговорности.

Улазна тачка програма се налази у Јава датотеци `LosPrincipS.java`.

```
package rs.math.oop.g09.p19.losPrincipS;

public class LosPrincipS {

    public static void main(String[] args) throws CloneNotSupportedException{
        Zaposleni z = new Zaposleni("Јован", "Петровић", "приправник", 30_000);
        System.out.println("Запослени: " + z);
        z.postaviPlatu(z.uzmiPlatu() * 1.2);
        int uspeh = z.sacuvajBazaPodataka();
        if(uspeh == 0)
            System.out.println("Информације су успешно сачуване у бази података.");
        else
            System.out.println("Информације нису успешно сачуване у бази података.");
    }
}
```

Овде се читање/измена поља објеката и њихово чување у бази података реализују методима класе `Zaposleni`.■

Пример 20. Креирати класе за рад са са подацима о запосленима, где ће функционалност која подржава упис информација о запосленом бити издвојена у посебан део.□

Сада класа `Zaposleni` има само POJO функционалност. Њен програмски кôд се налази у Јава датотеци `Zaposleni.java`.

```
package rs.math.oop.g09.p20.dobarPrincipS;

public class Zaposleni{
   private String ime;
   private String prezime;
   private String opisPosla;
   private double plata;

   public Zaposleni(String ime, String prezime, String opisPosla, double plata) {
      this.ime = ime;
      this.prezime = prezime;
      this.opisPosla = opisPosla;
      this.plata = plata;
   }

   public Zaposleni(){ this("","", "", 0); }

   public String uzmiIme(){ return ime; }

   public void postaviIme(String ime) { this.ime = ime; }

   public String uzmiPrezime(){ return prezime; }

   public void postaviPrezime(String prezime) { this.prezime = prezime; }

   public String uzmiOpisPosla(){ return opisPosla; }

   public void postaviOpisPosla(String opisPosla) { this.opisPosla = opisPosla; }

   public double uzmiPlatu(){ return plata; }

   public void postaviPlatu(double plata) { this.plata = plata; }

   @Override
   public String toString() {
      return "[име: " + ime + " " + prezime + ", посао: '" + opisPosla
              + "', плата: " + plata + "]";
   }
}
```

Интеракција са базом података, која се односи на информације о запосленом, је сада смештена у посебну класу, названу `ZaposleniBazaPodataka`. Наравно, функционалност за чување података у бази није заиста реализована, већ су само приказане поруке и постављени коментари. Програмски кôд ове класе се налази у Јава датотеци `ZaposleniBazaPodataka.java`.

```
package rs.math.oop.g09.p20.dobarPrincipS;

public class ZaposleniBazaPodataka {
    public static int ucitaj(Zaposleni z){
        System.out.println("Реализује учитавање информација о примерку класе '"
                + z.getClass().getSimpleName() + "' у бази података.");
        // овде треба да буду наредбе за читање информација о објекту из базе
        return 0;
```

```
    }

    public static int sacuvaj(Zaposleni z){
        System.out.println("Реализује чување информација о примерку класе '"
                + z.getClass().getSimpleName() + "' у бази података.");
        System.out.println("Чувају се следећи подаци: " + z);
        // овде треба да буду наредбе за упис информација о објекту у базу
        return 0;
    }
}
```

Сада принцип јединствене одговорности није нарушен, јер свака од класа има јединствену одговорност.

Улазна тачка програма се налази у Јава датотеци `DobarPrincipS.java`.

```
package rs.math.oop.g09.p20.dobarPrincipS;

public class DobarPrincipS {

    public static void main(String[] args) throws CloneNotSupportedException{
        Zaposleni z = new Zaposleni("Јован", "Петровић", "припрравник", 30_000);
        System.out.println("Запослени: " + z);
        z.postaviPlatu(z.uzmiPlatu() * 1.2);
        int uspeh = ZaposleniBazaPodataka.sacuvaj(z);
        if(uspeh == 0)
            System.out.println("Информације су успешно сачуване у бази података.");
        else
            System.out.println("Информације нису успешно сачуване у бази података.");
    }
}
```

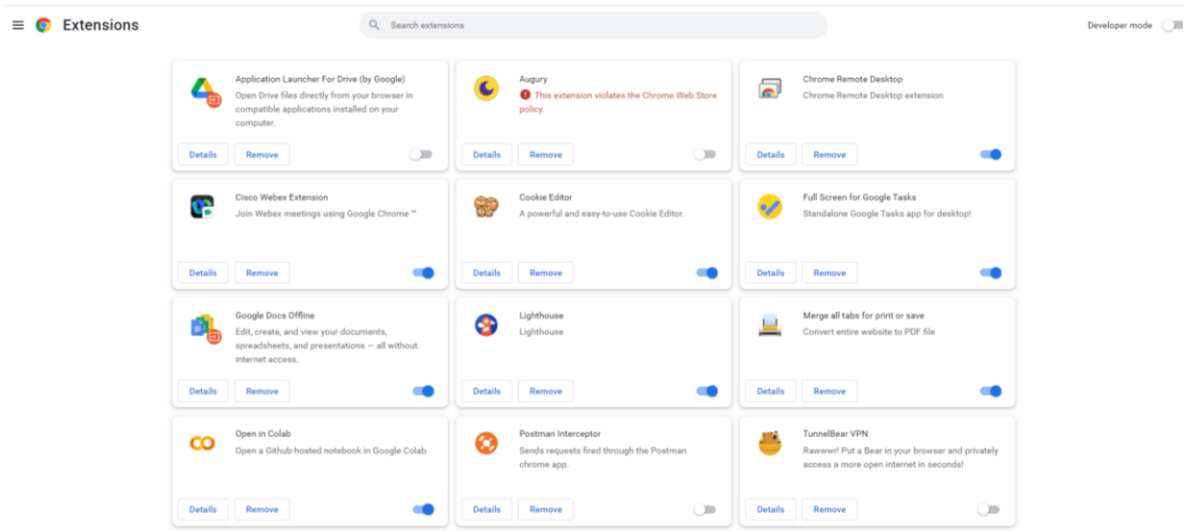
Овде се, за разлику од претходног примера, читање/измена поља објеката реализују методима класе `Zaposleni`, а њихово чување у бази података методима класе `ZaposleniBazaPodataka`.■

Принцип отворености и затворености

Принцип отворености и затворености гласи: софтверске компоненте треба да буду отворене за проширивање, али затворене за модификацију.

По овом принципу, класа треба да буде структуирана тако да реализује своје задатке без претпостављања да ће у будућности бити мењано њено понашање. Истовремено, треба да буде омогућено проширивање функционалности класе, на неки од начина композиције, садржавања, наслеђивања и имплементације који су обрађени у ранијим поглављима.

На пример, модерни веб прегледачи (као што су Google Chrome, Mozilla Firefox, Microsoft Edge и слично) су дизајнирани тако да омогућавају разноврсна проширења (екстензије), која кад се укључе, помажу у раду са одређеним типовима ресурса на вебу. Међутим, њихово укључивање и искључивање не омета основну функцију прегледача.



Страна за подешавање проширења (екстензија) код веб прегледача Google Chrome

Веб прегледачи су, дакле, дизајнирани тако да су затворени за модификацију али омогућавају проширења, па се може рећи да њихов дизајн на овом нивоу поштује принцип отворености и затворености.

Пример 21. Креирати Јава програм за одређивање површине геометријских објеката (кругова и правоугаоника).□

У овом случају се усваја тзв. „брз и прљав“ приступ, где је фокус на брзини развоја и где се не води рачуна о потреби проширивања развијеног програма у будућности. Класа за круг и класа за правоугаоник су развијене као „чисте“ POJO класе. У овом решењу чак није ни препознато да одређивање површине за ова два типа објекта представља фундаментално исти концепт.

Програмски кôд класе `Krug` се налази у Јава датотеци `Krug.java`.

```
package rs.math.oop.g09.p21.losPrincip0;

public class Krug {
   private double poluprecnik;

   public Krug(double poluprecnik) { this.poluprecnik = poluprecnik; }

   public double uzmiPoluprecnik() { return poluprecnik; }
}
```

Кôд за класу `Pravougaonik` се налази у Јава датотеци `Pravougaonik.java`.

```
package rs.math.oopPitanjaZadaci.g09.p21.losPrincip0;

public class Pravougaonik {
   private double sirina;
   private double visina;

   public Pravougaonik(double sirina, double visina) {
      this.sirina = sirina;
      this.visina = visina;
   }

   public double uzmiSirinu() { return sirina; }

   public double uzmiVisinu() { return visina; }
```

```
}
```

Улазна тачка програма се налази у Јава датотеци `LosPrincipO.java`.

```
package rs.math.oop.g09.p21.losPrincipO;

import java.util.Scanner;
import static java.lang.Math.PI;
import static java.lang.System.*;

public class LosPrincipO {

   public static double povrsinaPravougaonika(Pravougaonik p){
      return p.uzmiSirinu() * p.uzmiVisinu();
   }

   public static double povrsinaKruga(Krug k) {
      return PI * k.uzmiPoluprecnik() * k.uzmiPoluprecnik();
   }

   public static void main(String[] argumenti){
      Scanner sc = new Scanner(in);
      out.printf("Унесите полупречник круга: ");
      double r = sc.nextDouble();
      Krug k = new Krug(r);
      out.printf("Површина круга је: %f\n", povrsinaKruga(k));
      out.printf("Унесите ширину и висину правоугаоника: ");
      double a = sc.nextDouble();
      double b = sc.nextDouble();
      Pravougaonik p = new Pravougaonik(a,b);
      out.printf("Површина правоугаоника је: %f\n", povrsinaPravougaonika(p));
      sc.close();
   }
}
```

Одређивање површине круга и правоугаоника је, као што се може видети, реализовано посебним методима у оквиру класе `LosPrincipO`. Овакво решење нарушава принцип отворености и затворености, јер оно није отворено за проширивање. Конкретно, ако треба проширити развијену функционалност тако да буде подржано рачунање површина за још неке геометријске објекте, то би захтевало озбиљне промене у различитим деловима развијеног програмског кода.■

Пример 22. Креирати Јава програм за одређивање површине геометријских објеката (кругова и правоугаоника), водећи рачуна о потреби његовог проширења.□

Када се поведе мало рачуна о квалитету развијеног програмског кода, лако се уочава да активност одређивања површине представља одговорност и за круг и за правоугаоник. Та активност може бити дефинисана кроз Јава интерејс, па је одговорност дате класе, за реализацију те активности, одређена тако што класа имплементира интерфејс.

Програмски кôд интерфејса `Mera` се налази у Јава датотеци `Mera.java`.

```
package rs.math.oop.g09.p22.dobarPrincipO;

public interface Mera {
   double povrsina();
}
```

Сада су обе класе (за круг и за правоугаоник) саме одговорне за рачунање сопствене површине, тј. оне ће имплементирати интерфејс `Mera`.

Програмски кôд класе `Krug` се налази у Јава датотеци `Krug.java`.

```
package rs.math.oop.g09.p22.dobarPrincipO;

import static java.lang.Math.PI;

public class Krug implements Mera {
   private double poluprecnik;

   public Krug(double poluprecnik) { this.poluprecnik = poluprecnik; }

   public double uzmiPoluprecnik() { return poluprecnik; }

   @Override
   public double povrsina() { return PI * poluprecnik * poluprecnik; }
}
```

Кôд за класу `Pravougaonik` се налази у Јава датотеци `Pravougaonik.java`.

```
package rs.math.oop.g09.p22.dobarPrincipO;

public class Pravougaonik implements Mera{
   private double sirina;
   private double visina;

   public Pravougaonik(double sirina, double visina) {
      this.sirina = sirina;
      this.visina = visina;
   }

   public double uzmiSirinu() { return sirina; }

   public double uzmiVisinu() { return visina; }

   @Override
   public double povrsina() { return sirina * visina; }
}
```

Улазна тачка програма се налази у Јава датотеци `DobarPrincipO.java`.

```
package rs.math.oop.g09.p22.dobarPrincipO;

public class DobarPrincipO {
   public static void main(String[] argumenti){
      Mera[] oblici = new Mera[2];
      Scanner sc = new Scanner(in);
      out.printf("Унесите полупречник круга: ");
      double r = sc.nextDouble();
      oblici[0] = new Krug(r);
      out.printf("Унесите ширину и висину правоугаоника: ");
      double a = sc.nextDouble();
      double b = sc.nextDouble();
      oblici[1] = new Pravougaonik(a,b);
      sc.close();
      for(Mera o: oblici){
         out.printf("Површина: %f\n", o.povrsina());
```

```
        }
    }
}
```

У главном методу се и кругови и правоугаоници посматрају на униформан начин, као инстанце интерфејса `Mera`. Јасно је да главни метод не зависи од других класа и да ће коректно радити док год ради са објектима који имплементирају интерфејс `Mera`. Следи пример извршавања програма.

```
Унесите полупречник круга: 10
Унесите ширину и висину правоугаоника: 10 50
Површина: 314.159265
Површина: 500.000000
```

Овакав дизајн поштује принцип отворености и затворености: ако треба проширити развијену функционалност тако да буде подржано рачунање површина за још неке геометријске објекте, довољно је креирати нову класу за тај тип геометријског објекта који имплементира интерфејс `Mera` и метод за рачунање његове површине.■

Принцип заменљивости

Принцип заменљивости је добио име по научници Барбари Лисков. Он гласи: методи који користе референцу за натклasу/интерфејс морају да буду у могућности да успешно користе инстанце потклase/имплементације, а да при том ти методи немају информацију коју тачно подкласу/имплементацију користе.
Прецизније: ако је S изведен из T (или S имплементира T), тада објекти типа T у програму могу бити на сваком месту замењени објектима типа S, без промене пожељних особина програма. Притом, методи који користе T не знају која се од класа изведених из T (или таквих да имплементира T) стварно користи.
Још краће, класа мора реализовати све уговоре својих наткласа (или интерфејса које имплементира) – и на нивоу синтаксе и на нивоу семантике.
Поштовањем овог принципа се спречава злоупотреба наслеђивања.
Пример 23. Развити класе за правоугаоник и за квадрат које обезбеђују рачунање њихових површина.□
Ако се при развоју ових класа и разматрању односа између квадрата и правоугаоника не би узео у обзир принцип замене, могао би се појавити озбиљан проблем.
Нека дизајн класе `Pravougaonik`, који се налази у Јава датотеци `Pravougaonik.java`, буде суштински исти као што је био у претходном примеру.

```
package rs.math.oop.g09.p23.losPrincipL;

public class Pravougaonik {
   private double sirina;
   private double visina;

   public Pravougaonik(double sirina, double visina) {
      this.sirina = sirina;
      this.visina = visina;
   }

   public Pravougaonik() { this(2,2); }

   public double uzmiSirinu() { return sirina; }
```

```
   public void postaviSirinu(double sirina) { this.sirina = sirina; }

   public double uzmiVisinu() { return visina; }

   public void postaviVisinu(double visina) { this.visina = visina; }

   public double povrsina(){ return sirina * visina; }
}
```

Ако би, приликом дизајнирања класе која представља квадрат, програмер резоновао чисто математички, тј. ако би се само ослонио на чињеницу да квадрат јесте правоугаоник, без размишљања о принципу замене, онда би он могао да донесе одлуку да класу за квадрат изведе из класе за правоугаоник.

У том случају, програмски кôд за класу `Kvadrat`, који се налази у Јава датотеци `Kvadrat.java`, могао би да има следећи запис.

```
package rs.math.oop.g09.p23.losPrincipL;

public class Kvadrat extends Pravougaonik {
   public Kvadrat(double ivica) { super(ivica, ivica); }

   public Kvadrat() { this(2); }

   @Override
   public void postaviSirinu(double sirina) {
      super.postaviSirinu(sirina);
      super.postaviVisinu(sirina);
   }

   @Override
   public void postaviVisinu(double visina) {
      super.postaviVisinu(visina);
      super.postaviSirinu(visina);
   }
}
```

Овде се квадрат посматра као правоугаоник којем су ширина и висина једнаке, и превазилазе се методи за постављање висине како би се обезбедило да услов једнаке висине и ширине увек важи. Међутим, овакво обезбеђивање услова једнакости ширине и висине је створило незгодан споредни ефекат који доводи до тога да примерци класе `Kvadrat` не испуњавају све семантичке уговоре који важе за примерке класе `Pravougaonik`, што се јасно види у коду који следи.

```
package rs.math.oop.g09.p23.losPrincipL;

import static java.lang.System.*;

public class LosPrincipL {
   public static void povrsinaProvera(Pravougaonik p) {
      p.postaviSirinu(3);
      p.postaviVisinu(2);
      double povrsina = p.povrsina();
      if(Double.compare(povrsina, 6.0) == 0)
         out.printf(p.getClass().getSimpleName()
           + ": Све OK! Израчуната површина је %f%n", povrsina);
      else
```

```
            err.printf(p.getClass().getSimpleName()
              + ": Проблем! Израчуната површина је %f, треба да буде %f%n",
                  povrsina, 6.0);
    }

    public static void main(String[] argumenti){
        Pravougaonik p = new Pravougaonik();
        povrsinaProvera(p);
        Kvadrat k = new Kvadrat();
        povrsinaProvera(k);
    }
}
```

Провера једног од услова (тј. инваријанте) коју би требало да задовољи сваки правоугаоник је дата методом `povrsinaProvera()`. Ту се проверава да ли је, по постављању ширине и висине правоугаоника, резултат који враћа метод `povrsina()` тачан, тј. да ли се поклапа са производом претходно постављене ширине и висине. Претходно описани метод се у методу `main()` позива ради провере рачунања површина конкретних правоугаоника и квадрата. (Назив класе која одговара инстанци `p` се добија уланчаним позивањем метода `p.getClass().GetSimpleName()`).
Извршавање програма на стандардном излазу приказује следеће резултате.

```
Pravougaonik: Све ОК! Израчуната површина је 6.000000
Kvadrat: Проблем! Израчуната површина је 4.000000, треба да буде 6.000000
```

Класа за квадрат је изведена из класе за правоугаоника, али она очигледно не реализује све уговоре која реализује класа за правоугаоник, па је јасно да постоје ситуације где инстанца потклассе не може заменити инстанцу натклассе – оваквим дизајном је нарушен принцип заменљивости. ■

Принцип раздвајања интерфејса

Принципом раздвајања интерфејса фаворизује се дизајн са већим бројем малих интерфејса у односу на дизајн са малим бројем великих (монолитних) интерфејса. Другим речима, клијенти не треба да буду приморани да имплементирају непотребне методе, тј. методе које неће користити.
Пример 24. Развити интерфејсе и класе за софтверски систем који подржава рад ресторана и води рачуна о прихватању поруџбина (лично, телефонски, он-лајн) и о плаћању (лично, он-лајн).□
Лош дизајн би могао да буде заснован на постојању једног великог (монолитног) интерфејса `Restoran`.
Програмски кôд интерфејса `Restoran` се налази у Јава датотеци `Restoran.java`.

```
package rs.math.oop.g09.p24.losPrincipI;

public interface Restoran{
      public void prihvatiOnLajnPorudzbinu();
      public void prihvatiTelefonskuPorudzbinu();
      public void platiOnLajn();
      public void staniURedZaLicnuPorudzbinu();
      public void platiLicno();
}
```

У овом решењу су све класе које садрже методе за реализацију опслуживања (тј. класа `LicnoDosaoKlijentOpsluzivanje`, класа `TelefonskiKlijentOpsluzivanje` и класа

`OnLajnKlijentOpsluzivanje`) дизајниране тако да имплементирају интерфејс `Restoran`, а класе које представљају клијенте, тј. класе `LicnoDosaoKlijent`, `TelefonskiKlijent` и `OnLajnKlijent` садрже у пољу под називом `hraniSe` референцу на интерфејс `Restoran`, односно на инстанцу неке од класа за опслуживање која одговара начину на који се дати клијент храни.

Један од проблема са овим приступом огледа се у томе што свака од класа за опслуживање мора имплементирати све методе интерфејса `Restoran`.

На пример, овако би изгледао кôд за класу `OnLajnKlijentOpsluzivanje` који се налази у Јава датотеци `OnLajnKlijentOpsluzivanje.java`.

```
package rs.math.oop.g09.p24.losPrincipI;

import static java.lang.System.*;

public class OnLajnKlijentOpsluzivanje implements Restoran {
   @Override
   public void prihvatiOnLajnPorudzbinu() {
      out.println("Реализује се постављање он-лајн поруџбине!");
   }

   @Override
   public void prihvatiTelefonskuPorudzbinu() {
      err.println("Није могуће прихватити телефонску поруџбину за он-лајн клијента!");
   }

   @Override
   public void platiOnLajn() {
      out.println("Реализује се он-лајн плаћање за он-лајн клијента!");
   }

   @Override
   public void staniURedZaLicnuPorudzbinu() {
      err.println("Није могуће стати у ред личних поруџбина за он-лајн клијента!");
   }

   @Override
   public void platiLicno() {
      err.println("Није могуће лично платити за он-лајн клијента!");
   }
}
```

Уочава се да су у овој класи два метода (`prihvatiOnLajnPorudzbinu()` и `platiOnLajn()`) смислено имплементирана, а да преостала три метода имају имплементацију само зато што метод са датим потписом мора да постоји.

Програмски кôд за класу `OnLajnKlijent` се налази у Јава датотеци `OnLajnKlijent.java`

```
package rs.math.oop.g09.p24.losPrincipI;

public class OnLajnKlijent {
   private String ime;
   private String adresa;
   private String email;
   private Restoran hraniSe;
```

```
    public OnLajnKlijent(String ime, String adresa, String email, Restoran hraniSe) {
        this.ime = ime;
        this.adresa = adresa;
        this.email = email;
        this.hraniSe = hraniSe;
    }

    public Restoran getHraniSe() { return hraniSe; }

    public void setHraniSe(Restoran hraniSe) { this.hraniSe = hraniSe; }
}
```

Аналогна је ситуација са другим типовима опслуживања и другим типовима клијената, па они неће бити наведени у тексту.

Овде је потпуно јасно да изабрани дизајн, који није у складу са принципом раздвајања интерфејса, не доводи до елегантног решења. Покушај да се, на пример, дода нови тип плаћања, захтева промену и доградњу великог дела развијеног програма, јер би се морао додати метод у монолитини интерфејс, па онда дорадити скоро цео програмски кôд.■

Ако се поштује принцип раздвајања интерфејса, тада су интерфејси орјентисани према клијенту и клијент не сме да има обавезу да зависи од било којег метода који не користи.

Пример 25. Развити интерфејсе и класе за софтверски систем који подржава рад ресторана и води рачуна о прихватању поруџбина (лично, телефонски, он-лајн) и плаћању (лично, он-лајн), тако да буде уважен принцип раздвајања интерфејса.□

Опет се решава исти проблем као у претходном примеру. Додатном анализом проблема и анализом претходног решења, уочава се да се активност ресторана (тј. оно што у претходном примеру названо опслуживањем) састоји од две компоненте: поруџбине и плаћања. Даље, три операције које се у "старом" интерфејсу за ресторан односе на поруџбину су суштински једна операција. На крају, преостале две операције из "старог" интерфејса за ресторан се могу изједначити.

Стога ће, уместо једног интерфејса са пет метода, сада постојати два интерфејса, `Porudzbina` и `Placanje`, са по једним методом. Програмски кôд интерфејса `Porudzbina` се налази у Јава датотеци `Porudzbina.java`.

```
package rs.math.oop.g09.p25.dobarPrincipI;

public interface Porudzbina {
    public void prihvatiPorudzbinu();
}
```

Програмски кôд интерфејса `Placanje` се налази у Јава датотеци `Placanje.java`.

```
package rs.math.oop.g09.p25.dobarPrincipI;

public interface Placanje {
    public void platiPorudzbinu();
}
```

Пошто су раздвојени интерфејси, раздвојене су и класе које имплементирају интерфејсе, и њихова структура је поједностављена (захтева се да имплементирају један метод, а не пет). Све класе, којима се реализује поручивање (овде су у питању класе `LicnoNapravljenaPorudzbina`, `TelefonskaPorudzbina` и `OnLajnPorudzbina`), имплементирају интерфејс `Porudzbina`, а класе којима се реализује наплата (ради се о класама `LicnoPlacanje`, `TelefonskoPlacanje` и `OnLajnPlacanje`), имплементирају

интерфејс `Placanje`. Сада класе, које представљају клијенте (односно класе `LicnoDosaoKlijent`, `TelefonskiKlijent` и `OnLajбKlijent`), садрже два поља – једно типа `Porudzbina` реферише на дату поруџбину, а друго типа `Placanje` садржи референцу на изабрано плаћање.

Програмски кôд за класу `OnLajnPorudzbina` се налази у Јава датотеци `OnLajnPorudzbina.java`.

```
package rs.math.oop.g09.p25.dobarPrincipI;

public class OnLajnPorudzbina implements Porudzbina {
   @Override
   public void prihvatiPorudzbinu() {
      out.println("Реализује се постављање он-лајн поруџбине!");
   }
}
```

Програмски кôд за класу `OnLajnPlacanje` се налази у Јава датотеци `OnLajnPlacanje.java`.

```
package rs.math.oop.g09.p25.dobarPrincipI;

import static java.lang.System.out;

public class OnLajnPlacanje implements Placanje {

   @Override
   public void platiPorudzbinu() {
      out.println("Реализује се он-лајн плаћање за поруџбину!");
   }
}
```

Са оваквим дизајном ове две класе (а тако је и код осталих класа за све типове поручивања и плаћања) програмер нема обавезу да имплементира методе који неће бити коришћени.

Програмски кôд за класу `OnLajnKlijent` се налази у датотеци `OnLajnKlijent.java`

```
package rs.math.oop.g09.p25.dobarPrincipI;

public class OnLajnKlijent {
   private String ime;
   private String adresa;
   private String email;
   private Porudzbina porudzbina;
   private Placanje placanje;


   public OnLajnKlijent(String ime, String adresa, String email, Porudzbina porudzbina,
                        Placanje placanje) {
      this.ime = ime;
      this.adresa = adresa;
      this.email = email;
      this.porudzbina = porudzbina;
      this.placanje = placanje;
   }

   public Porudzbina getPorudzbina() { return porudzbina; }
```

```
    public void setPorudzbina(Porudzbina porudzbina) { this.porudzbina = porudzbina; }

    public Placanje getPlacanje() { return placanje; }

    public void setPlacanje(Placanje placanje) { this.placanje = placanje; }
}
```

Други типови поручивања, типови плаћања и типови клијената имају аналогну структуру, па они неће бити овде наведени.

Овакав дизајн поштује принцип раздвајања интерфејса и доводи до квалитетног решења, отвореног за надоградњу. Ако би се, на пример, овде додавао нови тип плаћања, морао би да се мења само мали део развијеног програма, а захтеване промене би биле локализоване – не би се „распршиле“ кроз скоро цео програмски кôд, као што је то био случај у претходном примеру.■

Принцип инверзије зависности

Принцип инверзије зависности гласи: треба зависити од апстракција, а не од конкретне реализације.

По овом принципу, модули вишег нивоа никако не треба да зависе од модула нижег нивоа, већ и модули нижег нивоа и модули вишег нивоа треба да зависе од апстракција. При том, апстракције не треба да зависе од детаља, већ детаљи треба да зависе од апстракције.

На пример, у процесу плаћања кредином картицом, само процесирање кредитних картица не треба да зависи од типа кредитне картице.

Мало конкретније исказан принцип у језику Јава (слично је и код других популарних програмских језика): сваки пут када се у оквиру неког метода класе A креира примерак класе B, тада је оформљена зависност између класе A и класе B. Принцип инверзије зависности захтева да, уместо да класа A захтева креирање објекта дате класе, она би требала да захтева референцу на апстракцију објекта (независну од детаља), а други сегмент програмског система који се развија треба да се побрине да за захтевану рефреренцу обезбеди конкретну реализацију тог апстрактног објекта.

Пример 26. Развити класе за клијент-сервер систем где клијент, приликом реализације датог метода, користи метод сервера.□

У овом случају, задатак ће бити решен на уобичајен начин. Постојаће класа за сервер и класа за клијента. Програмски кôд за класу која представља сервис `ServisB` се налази у датотеци `ServisB.java`

```
package rs.math.oop.g09.p26.losPrincipD;

public class ServisB {
   public String getInfo() {
      return "Информације о сервису ServisB";
   }
}
```

Програмски кôд за класу која представља сервис `KlijentA` се налази у датотеци `KlijentA.java`

```
package rs.math.oop.g09.p26.losPrincipD;

import static java.lang.System.out;
```

```
public class KlijentA {
   ServisB servis = new ServisB();

   public void uradiNesto() {
      String info = servis.getInfo();
      out.println("KlijentA - " + info);
   }
}
```

Приликом креирања клијента, коришћењем оператора `new`, директно се креира инстанца сервера и то инстанца класе `ServisB` .
Програмски кôд улазне тачке програма је дат у Јава датотеци `LosPrincipL.java`.

```
package rs.math.oop.g09.p26.losPrincipD;

public class LosPrincipD {
   public static void main(String[] args){
      KlijentA ka = new KlijentA();
      ka.uradiNesto();
   }
}
```

У главном програму се само креира инстанца клијента, тј. класе `KlijentA` и позове метод `uradiNesto()`. Приликом извршавања програма, на стандардном излазу се приказују следећи резултати.

```
KlijentA - Информације о сервису ServisB
```

Јасно је да у овом случају није поштован принцип инверзије зависности.■
Да би се реализовала инверзија зависности, потребно је да постоје апстракције објеката, што у Јави обично буду интерфејси.
Пример 27. Развити интерфејсе и класе за клијент-сервер систем, где клијент приликом реализације датог метода користи метод сервера, водећи рачуна о принципу инверзије зависности.□
Као што је истакнуто, потребно је обезбедити апстракције и у ту сврху ће служити интерфејси. Потребан је интерфејс који представља апстракцију сервиса и интерфејс који представља апстракцију клијента.
Програмски кôд за интерфејс `Servis` се налази у Јава датотеци `Servis.java`.

```
package rs.math.oop.g09.p27.dobarPrincipDKonstruktor;

public interface Servis {
   String getInfo();
}
```

У овом примеру, већи број класа ће имплементирати претходно дефинисан сервис.
Програмски кôд класе `ServisB` која имплементира интерфејс `Servis` се налази у Јава датотеци `ServisB.java`, која има следећи садржај.

```
package rs.math.oop.g09.p27.dobarPrincipDKonstruktor;

public class ServisB implements Servis {
   @Override
   public String getInfo() {
      return "Информације о сервису ServisB";
   }
}
```

Програмски кôд класе `ServisC`, која имплементира интерфејс `Servis`, се налази у Јава датотеци `ServisC.java`.

```
package rs.math.oop.g09.p27.dobarPrincipDKonstruktor;

public class ServisC implements Servis {
   @Override
   public String getInfo() {
      return "Информације о сервису ServisC";
   }
}
```

Програмски кôд класе `ServisD` која имплементира интерфејс `Servis` се налази у Јава датотеци `ServisD.java`.

```
package rs.math.oop.g09.p27.dobarPrincipDKonstruktor;

public class ServisD implements Servis {
   @Override
   public String getInfo() {
      return "Информације о сервису ServisD";
   }
}
```

У овом примеру је дефинисана и апстракција клијента. Програмски кôд за интерфејс `Klijent` се налази у Јава датотеци `Klijent.java`.

```
package rs.math.oop.g09.p27.dobarPrincipDKonstruktor;

public interface Klijent {
   void uradiNesto();
}
```

Уочава се да су интерфејси за сервис и за клијент потпуно независни.
Програмски кôд за класу `KlijentA`, која имплементира интерфејс `Klijent`, се налази у датотеци `KlijentA.java`

```
package rs.math.oop.g09.p27.dobarPrincipDKonstruktor;

import static java.lang.System.out;

public class KlijentA implements Klijent {
   private Servis servis;

   public KlijentA(Servis servis) {
      this.servis = servis;
   }
```

```
    @Override
    public void uradiNesto() {
        String info = servis.getInfo();
        out.println("KlijentA - " + info);
    }
}
```

Уочава се да је поље за сервис типа интерфејса, а не конкретног типа. Даље, постављањем наредбе доделе конкретног типа сервиса у конструктор клијента, омогућено је да се из спољашњости одреди који ће од сервиса на располагању конкретизовати апстракцију. Другим речима, омогућено је да овај клијент ради са различитим конкретизацијама сервиса – у зависности од тога која је конкретизација одабрана приликом креирања клијента.

Овакав начин обезбеђења инверзије зависности се назива инверзија зависности помоћу конструктора.

Програмски кôд улазне тачке програма је дефинисан садржајем Јава датотеке `DobarPrincipDKonstruktor.java`.

```
package rs.math.oop.g09.p27.dobarPrincipDKonstruktor;

public class DobarPrincipDKonstruktor {
    public static void main(String[] args){
        Servis sB = new ServisB();
        Servis sC = new ServisC();
        Servis sD = new ServisD();

        Klijent kA = new KlijentA(sB);
        kA.uradiNesto();

        kA = new KlijentA(sC);
        kA.uradiNesto();

        kA = new KlijentA(sD);
        kA.uradiNesto();
    }
}
```

Јасно је да се у главном програму, приликом креирања клијента, одређује који ће сервер бити придружен новокреираном клијенту, док је у претходном примеру то било фиксирано на нивоу конкретне класе која представља клијента. Овде су извршена три креирања објекта-клијента и при сваком креирању је клијенту придружен други сервис. Још једна важна напомена: све променљиве у главном програму су типа интерфејса – нема ниједне променљиве која је типа класе.

Извршавање програма на стандардном излазу приказује следеће резултате.

```
KlijentA - Информације о сервису ServisB
KlijentA - Информације о сервису ServisC
KlijentA - Информације о сервису ServisD
```

Дакле, у овом случају јесте поштован принцип инверзије зависности и реализована је инверзија зависности преко конструктора. Овај приступ има једну интересантну карактеристику: придруживање конкретног сервиса клијенту се врши приликом креирања клијента – није могуће већ креираном клијенту „у лету“ придружити неки други сервис.■

Пример 28. Развити интерфејсе и класе за клијент-сервер систем, где клијент, приликом реализације датог метода, користи метод сервера, водећи рачуна о принципу инверзије зависности. Омогућити да се током рада клијента може мењати конкретан сервис који тај клијент користи .□

Интерфејси `Servis` и `Klijent` су потпуно исти као у претходном примеру, па се неће овде поново наводити. Програмски кôд класа `ServisB`, `ServisC` и `ServisD` које имплементирају интерфејс `Servis`, потпуно је исти као у претходном примеру, па се неће овде поново наводити.

Програмски кôд за класу `KlijentA` која имплементира интерфејс `Klijent` се налази у датотеци `KlijentA.java`

```
package rs.math.oop.g09.p28.dobarPrincipDMetodPostavi;

import static java.lang.System.out;

public class KlijentA implements Klijent {
   private Servis servis;

   public KlijentA( Servis servis) {
      this.servis = servis;
   }

   public void postaviServis(Servis servis) {
      this.servis = servis;
   }

   @Override
   public void uradiNesto() {
      String info = servis.getInfo();
      out.println("KlijentA - " + info);
   }
}
```

Овде је, у односу на програмски кôд исте класе из претходног примера, само додат метод `postaviServis()`, за постављање референце на сервис који ће клијент да користи. Дакле, овде се референца на сервис може поставити на два начина: приликом креирања примерка класе `KlijentA` и позивом метода `postaviServis()`. Овај други начин се назива инверзија зависности помоћу метода за постављање поља.

Програмски кôд улазне тачке програма је дефинисан садржајем Јава датотеке `DobarPrincipDMetodPostavi.java`.

```
package rs.math.oop.g09.p28.dobarPrincipDMetodPostavi;

public class DobarPrincipDMetodPostavi {
   public static void main(String[] args){
      Servis sB = new ServisB();
      Servis sC = new ServisC();
      Servis sD = new ServisD();

      KlijentA ka = new KlijentA(sB);
      ka.uradiNesto();

      ka.postaviServis(sC);
      ka.uradiNesto();
```

```
        ka.postaviServis(sD);
        ka.uradiNesto();
    }
}
```

Овде је извршено једно креирање објекта-клијента и при том креирању је клијенту придружен један сервис, а онда је два пута позивима метода `postaviServis()` мењан сервис придружен клијенту.
Као што се и може очекивати, извршавањем програма добија се следећи резултат на стандардном излазу.

```
KlijentA - Информације о сервису ServisB
KlijentA - Информације о сервису ServisC
KlijentA - Информације о сервису ServisD
```

И у овом случају је поштован принцип инверзије зависности. Реализована је инверзија зависности преко конструктора и инверзија зависности помоћу метода за постављање поља.■
На крају, треба напоменути да се у реализацијама, које поштују принцип инверзије зависности, најчешће користи предефинисани контејнер за инверзију зависности, који је већ имплементиран у датом развојном окружењу[9].

9.4.2. Објектно оријентисани дизајн – препоруке

Принципи представљају, војничком терминологијом исказано, упутства на стратешком (тј. општијем) нивоу, а препоруке су упутства на тактичком (тј. конкретнијем) нивоу. Као и код принципа, примена преорука зависи од контекста у којем се ради, тј. окружења.
Неке од најважнијих препорука за дизајн класа и интерфејса су побројане у даљем тексту.

Користити наслеђивање искључиво за моделирање односа „јесте“

Понекад програмери претерују у коришћењу наслеђивања. Наслеђивање, као што се досад могло видети, има јако много корисних особина. Међутим, у жељи за ефикасношћу, у жељи да програм буде што концизније написан и да се поново искористи већ развијени кôд, програмери понекад искористе наслеђивање и за врсту везе која не пролази тест односа „јесте“. На тај начин се добије краткорочно убрзање процеса развоја софтвера по цену урушавања стабилности софтверског система који се развија, чиме се креирају дугорочни проблеми.
Пример 29. Креирати класу са информацијама о запосленом и то: име, презиме, опис посла и плата. Креирати класу за раднике запослене преко студентске задруге.□
Програмски кôд за класу `Zaposleni` се налази у Јава датотеци `Zaposleni.java`.

```
package rs.math.oop.g09.p29.losOdnosJeste;

public class Zaposleni {
    private String ime;
    private String prezime;
    private String opisPosla;
```

[9] На пример, код програмског језика Јава би то могао бити контејнер у оквиру радног оквира Spring. Тиме је програмер ослобођен додатног рада на развоју контејнера за инверзију зависности, али зато у свој пројекат мора укључити нове предефинисане библиотеке.

```
   private double plata;

   public Zaposleni(String ime, String prezime, String opisPosla, double plata) {
      this.ime = ime;
      this.prezime = prezime;
      this.opisPosla = opisPosla;
      this.plata = plata;
   }

   public Zaposleni(){ this("","", "", 0); }

   public String uzmiIme(){ return ime; }

   public void postaviIme(String ime) { this.ime = ime; }

   public String uzmiPrezime(){ return prezime; }

   public void postaviPrezime(String prezime) { this.prezime = prezime;  }

   public String uzmiOpisPosla(){ return opisPosla; }

   public void postaviOpisPosla(String opisPosla) { this.opisPosla = opisPosla; }

   public double uzmiPlatu(){ return plata; }

   public void postaviPlatu(double plata) { this.plata = plata; }

   public void povecajPlatu(double zaProcenat) {
      double iznosPovisice = plata * zaProcenat / 100;
      this.plata += iznosPovisice;
   }

   @Override
   public String toString() {
      return "[име: " + ime + " " + prezime + ", посао: '" + opisPosla
            + "', плата: " + plata + "]";
   }
}
```

Подаци о раднику који ради преко студентске задруге обихватају: име, презиме и опис посла, али не обухватају плату, јер се ти радници плаћају по сату.
Које би биле последице одлуке да се класа `RadnikPrekoStudentskeZadruge` направи као потклаcа класе `Zaposleni` којој је додато поље `satnica`? Ако се тако уради, програмски кôд за класу `RadnikPrekoStudentskeZadruge` би имао следећи садржај.

```
package rs.math.oop.g09.p29.losOdnosJeste;

public class RadnikPrekoStudentskeZadruge extends  Zaposleni{
    private double satnica;

    public RadnikPrekoStudentskeZadruge(String ime, String prezime,
                                    String  opisPosla, double satnica) {
        super(ime, prezime, opisPosla, -1);
        this.satnica = satnica;
    }
```

```
    public RadnikPrekoStudentskeZadruge(){ this("","", "", 0); }

    public double uzmiSatnicu(){ return satnica; }

    public void postaviSatnicu(double satnica) { this.satnica = satnica;  }
    public void povecajSatnicu(double zaProcenat) {
        double iznosPovisice = satnica * zaProcenat / 100;
        this.satnica += iznosPovisice;
    }

    @Override
    public String toString() {
        return "[име: " + uzmiIme() + " " + uzmiPrezime()
                + ", посао: '" + uzmiOpisPosla()
                + "', сатница: " + satnica + "]";
    }
}
```

Овако нешто не би била добра идеја: инстанца класе `RadnikPrekoStudentskeZadruge` садржи и поље за плату и поље за сатницу, а то доводи до проблема.
Програмски кôд улазне тачке програма је дефинисан садржајем Јава датотеке `LosOdnosJeste.java`.

```
package rs.math.oop.g09.p29.losOdnosJeste;

public class LosOdnosJeste {
    public static void main(String[] argumenti){
        RadnikPrekoStudentskeZadruge mm = new RadnikPrekoStudentskeZadruge("Марко",
              "Марковић", "sekretar", 1000);
        System.out.println("По креирању радника:");
        System.out.println("Радник: " + mm + " - плата: " + mm.uzmiPlatu());
        mm.povecajSatnicu(10);
        System.out.println("По повећању сатнице за 10%:");
        System.out.println("Радник: " + mm + " - плата: " + mm.uzmiPlatu());
        mm.povecajPlatu(10);
        System.out.println("По повећању плате за 10%:");
        System.out.println("Радник: " + mm + " - плата: " + mm.uzmiPlatu());
    }
}
```

Извршавање програма на стандардном излазу приказује следеће резултате.

```
По креирању радника:
Радник: [име: Марко Марковић, посао: 'sekretar', сатница: 1000.0] - плата: -1.0
По повећању сатнице за 10%:
Радник: [име: Марко Марковић, посао: 'sekretar', сатница: 1100.0] - плата: -1.0
По повећању плате за 10%:
Радник: [име: Марко Марковић, посао: 'sekretar', сатница: 1100.0] - плата: -1.1
```

Овде се може приступити и плати и сатници код радника који ради преко студентске задруге. Закључак: пошто однос између објекта радник преко задруге и запосленог не пролази тест “јесте” (јер радник преко задруге није запослен) не треба формирати однос наслеђивања.■

Заједничке операције и поља сместити у натклase

Ова препорука је интуитивно јасна – функционалност коју реализује више класа поставити што је могуће навише у дрвету наслеђивања или графу имплементација

интерфејса. На тај начин се уклања непотребно понављање и дуплирање програмског кода, одржава се једноставност и повећава се могућност поновне искористивости развијеног кода.
Тако је у примеру 14 из поглавља 8, оформљена класа `Ljubimac` као наткласа за класе `Macka` и `Pas`, а на исти начин је у примерима на почетку овог поглавља оформљена апстрактна класа `GeometrijskiObjekat` као наткласа класа `Tacka`, `Prava`, `Duz`, `Krug`, `Trougao` и `Cetvorougao`. И у претходном примеру би уважавање ове препоруке, тј. креирање наткласе која би садржавала поља и методе из класа `Zaposleni` и `RadnikPrekoStudentskeZadruge` успешно решило проблем који је тамо уочен.

Не користити наслеђивање, сем уколико оно има смисла за све методе класе из које се наслеђује

Ова препорука је обухваћена принципом заменљивости, обрађеним у претходној секцији. Уколико постоји бар један метод наткласе/интерфејса такав да нема смисла када се наткласа/интерфејс замени потклaсом/имплементацијом, онда не треба креирати однос наслеђивања/имплементације.
Тако је, на пример, извођење класе `Kvadrat` из класе `Pravougaonik` у примеру 23 било неадекватно, јер изведена класа `Kvadrat` није испуњавала све уговоре на које се обавезала класа `Pravougaonik`.

Приликом превазилажења метода не мењати очекивано понашање

Ова препорука значи да се принцип заменљивости из претходне секције примењује и на синтаксу и на семантику (тј. значење, понашање).
У примеру 23 је баш то било у питању: изведена класа `Kvadrat` је, приликом постављања ширине, аутоматски постављала и висину, чиме је мењала понашање очекивано од класе `Pravougaonik` да су ширина и висина правоугаоника независне и да позив за постављање ширине правоугаоника нема никакав утицај на висину.

Дати предност композицији и садржавању у односу на наслеђивање

Пракса је показала да претерана употреба наслеђивања доводи до формирања веома дубоких хијерархија наслеђивања, описаних великим дрвоидним структурама. Такве хијерархије наслеђивања нису флексибилне и са протеком времена постају све веће, а самим тим и напор који је потребно уложити за одржавање и надоградњу таквог кода. Стога се препоручује фаворизовање композиције и садржавања у односу на наслеђивање. Приликом одлуке да ли користити наслеђивање за моделирање односа између датих класа, пажљиво размотрити и све претходно наведене препоруке.

Избегавати употребу заштићених поља

Модификатор `protected` не пружа много заштите атрибутима и методама неке класе, из два разлога:

- Може се увек направити потклаcа неке класе и на тај начин приступити заштићеном атрибуту/методу.
- У програмском језику Јава све класе у истом пакету имају приступ `protected`, атрибутима/методама, тако да се класа може сместити у исти пакет и тиме омогућити приступ.

Међутим, `protected` методи могу бити корисни за назначавање да дати метод није спреман за општу употребу и да треба да буде редефинисан у поткласама, као што је урађено у примерима 15-17 секције 9.3.4, где је описан метод `clone()`.

Користити полиморфизам, а не информације о типу

У објектно оријентисаном програмирању се не сматра добром праксом да се угњежденим упитима проверава тип објекта, па се, у зависности од исхода провере, предузима одговарајућа активност,
На пример, кад год се наиђе на програмску конструкцију следећег облика:

```
if (x instanceof Tip1)
       akcija1(x);
else if (x instanceof Tip2)
       akcija2(x);
```

треба размотрити могућност примене полиморфизма. Прво треба проверити да ли `akcija1()` и `akcija2()` описују сродно понашање, па ако је то случај, то понашање названо, на пример, `akcija()`, треба да буде метод заједничке натклase или интерфејса који имплементирају типови `Tip1` и `Tip2`.
Потом једноставно треба позвати `x.akcija()` па ће механизам динамичког активирања, који је инхерентан полиморфизму, покренути одговарајућу акцију.

9.5. Резиме

У овом поглављу је демонстриран потенцијал објектно оријентисаног програмирања за решавање комплексних изазова који се јављају у развоју модерног софтвера.
Кроз увођење концепата апстрактне класе и интерфејса наглашен је значај раздвајања апстракција од њихових реализација.
До „осећаја“ како треба организовати програмски кôд, односно до којег нивоа га треба апстраховати, долази се: 1) изучавањем теорије, 2) анализом добрих примера из праксе, али и 3) на основу сопственог искуства. Прва два аспекта су донекле покривена кроз принципе и препоруке објектно оријентисаног дизајна и кроз дате примере који су били итеративно побољшавани. За трећи, можда и најважнији аспект, потребно је стећи године искуства развоју софтверских решења.

9.6. Питања и задаци

1. Које су сличности, а које разлике између апстрактних класа и интерфејса? Илустровати примерима.
2. Објаснити и примером илустровати наслеђивање апстрактних класа.
3. Истражити и илустровати примерима потенцијалне проблеме вишеструког наслеђивања. Како су креатори Јаве надоместили непостојање вишеструког наслеђивања?
4. Како се наслеђивање, а како имплементација интерфејса могу повезати са принципом полиморфизма?
5. Објаснити и примером илустровати проширивање интерфејса.

6. Примером илустровати ситуацију када је проблем елегантније решити употребом интерфејса, уместо наслеђивањем апстрактне класе. Такође, примером илустровати и обрнуту ситуацију.
7. Шта је омогућено имплементацијом интерфејса `Comparable`, а шта имлементацијом интерфејса `Comparator`? Илустровати примером.
8. Упоредити предности и недостатке употребе конструктора копије за прављење копије објекта са предностима и недостацима употребе механизма клонирања имплементацијом интерфејса `Cloneable`.
9. Објаснити и илустровати примерима групу SOLID принципа објектно оријентисаног дизајна. Истражити још неке принципе објектно оријентисаног дизајна (на пример GRASP). Да ли има преклапања између ових група принципа?
10. Које су најважније препоруке за дизајн класа и интерфејса? Како се препоруке уклапају у претходно разматране принципе објектно оријентисаног дизајна?

10. Угњеждене класе

Јава допушта да се дефинише класа унутар неке друге класе. Таква класа се назива угњеждена класа. Постоје два типа угњеждених класа: статичка угњеждена класа и унутрашња класа (или нестатичка угњеждена класа).

```
class SpoljasnjaKlasa{
    ...
    static class StatickaUgnjezdenaKlasa{
      ...
    }
    ...
    class UnutrasnjaKlasa{
      ...
    }
}
```

Као што се може видети из наведеног кода, разлика је у постојању модификатора `static`. Нестатичка угњеждена класа се заправо зове унутрашња класа с обзиром на њену блиску везу са спољашњом класом — она се може сматрати њеном правом чланицом попут нестатичких метода и нестатичких поља.

Скуп металингвистичких формула које описују класу, а које обухватају угњеждене и унутрашње класе, има следећи облик:

```
<тип класе> ::= [модификатор видљивости> ][final |abstract ]class <назив класе>
                              <проширивање><имплементација><тело класе>
<проширивање> ::= extends <назив класе>
<имплементација> ::= implements <листа назива интерфејса>
<листа назива интерфејса> ::= {<назив интерфејса>,}<назив интерфејса>
<назив интерфејса> ::= {<идентификатор>.}<идентификатор>
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>
                    |<иницијализациони блок>|<конструктор>
                    |<угњеждена унутрашња класа>)}}
<дефиниција поља> ::=  [<модификатор видљивости> ][static ][final ]
                                <декларација и иницијализација променљивих>
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= [<модификатор видљивости> ][static ][final |abstract ]
                                 <повратни тип><назив метода>(<параметри>)
<повратни тип> ::= <тип>|void
<назив метода>::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= [final ]<тип>
<назив параметра>::= <идентификатор>
<тело метода> ::= <блок>
<иницијализациони блок> ::= [static ]<блок>
<конструктор> ::= <име класе>(<параметри>)<блок>
<име класе> ::= <идентификатор>
<угњеждена унутрашња класа> ::= [static ]<тип класе>
```

Уочава се да је овде модификована металингвистичка променљива која описује тело класе.

Постоји неколико разлога за коришћење угњеждених класа.

- То је начин логичког груписања класа које се користе само на једном месту.
- Тиме се побољшава енкапулација (учаурење).

- Ако се класа B смести унутар класе A, из метода класе B се може приступати пољима класе A чак иако су она приватна.
- Угњеждене класе доводе до читљивијег кода који се лакше одржава.
- Угњеждавање малих класа у велику доводи до тога да кôд неке операције буде смештен близу места коришћења.

10.1. Статичке угњеждене класе

Статичка угњеждена класа се може упоредити са статичким пољем или статичким методом. То значи да статичка угњеждена класа не представља праву чланицу инстанци спољашње класе. Веза је „лабавија“ и више је мотивисана могућношћу погодне логичке организације кода, тј. као што пакети омогућавају груписање логички блиских класа тако и концепт статичких угњеждених класа омогућава груписање на још ближем нивоу.
Животни циклус инстанци статичке угњеждене класе није зависан од животног циклуса објеката спољашње класе. Баш као што и животни циклус статичког поља није везан за животни циклус појединачних објеката неке класе.

```
StatickaUgnjezdenaKlasa staticka = new Spoljasnja.StatickaUgnjezdenaKlasa();
```

Као што се види у коду изнад, нигде се не спомиње инстанца спољашње класе. За разлику од статичког поља, које је јединствено и већ унапред меморијски алоцирано, објекти статичке угњеждене класе нису унапред инстанцирани и њихов број није ограничен. То значи да је могуће креирати произвољан број инстанци статичке угњеждене класе и све оне ће “живети” као стандардни независни објекти на хипу.

```
StatickaUgnjezdenaKlasa staticka1 = new Spoljasnja.StatickaUgnjezdenaKlasa();
StatickaUgnjezdenaKlasa staticka2 = new Spoljasnja.StatickaUgnjezdenaKlasa();
```

Статичка угњеждена класа не може приступати нестатичким пољима спољашње класе. Овде се опет крије аналогија са статичким методима — статички методи не могу приступати нестатичким пољима класе. Разлог је то што инстанца спољашње класе уопште не мора да постоји током живота статичке угњеждене класе. А чак и да постоји, опет не би било јасно на коју се инстанцу спољашње класе мисли. Исто важи и за приступ нестатичким методима спољашње класе.

```
StatickaUgnjezdenaKlasa staticka1 = new Spoljasnja.StatickaUgnjezdenaKlasa();
StatickaUgnjezdenaKlasa staticka2 = new Spoljasnja.StatickaUgnjezdenaKlasa();
SpoljasnjaKlasa spoljasnja1= new Spoljasnja();
SpoljasnjaKlasa spoljasnja2= new Spoljasnja();
```

Са друге стране, статичка угњеждена класа има приступ статичким пољима и методима спољашње класе, чак иако су та поља приватна. Следећи пример демонстрира могућности статичке угњеждене класе и њену интеракцију са спољашњом класом.
Пример 1. Написати Јава програм који рачуна плату на месечном нивоу. Плата је дефинисана као умножак броја радних сати и цене сата. Минималан број сати које радник мора да оствари је 160 (норма), а све преко тога се рачуна као прековремени рад. Прековремени сати су реткост и постоји засебан механизам рачунања прековременог додатка. За сваки наредни сат преко норме увећава се цена по сату за 2%. На пример, ако је радник радио 1 прековремени сат добиће 102% цене сата као додатак. Ако је радио два сата, добиће 102%+ 104.04%=206.04% итд. Максимална цена по сату је 2000 РСД, било да је реч о основној цени рада или прековременој. □

Решење овог задатка је реализовано кроз три класе `ObracunPlata`, `PrekovremeniRad` и `ObracunPlataTest`, где је `PrekovremeniRad` приватна статичка угњеждена класа унутар класе `ObracunPlata`. Иако је било могуће `main()` метод убацити у класу `ObracunPlata`, намерно је он записан у одвојеној класи `ObracunPlataTest` како би се демонстрирао ефекат модификатора `private` над угњежденом класом.

```
package rs.math.oop.g10.p01.nestatickaUgnjezdenaObracunPlate;

public class ObracunPlata {
    private static int minimalnoSati=160;
    private static int maksimalnaCenaSata=2000;
    private int cenaSata;

    public ObracunPlata(int cenaSata) {
        if(cenaSata>maksimalnaCenaSata) {
               System.out.println("Редукујемо цену на износ "+maksimalnaCenaSata);
               cenaSata = maksimalnaCenaSata;
        }
        if(cenaSata<=0) {
               System.err.println("Цена сата "+cenaSata+" нема смисла.");
               System.exit(1);
        }
        this.cenaSata=cenaSata;
    }

    public double izracunajPlatu(int brojSati) {
        if(brojSati<minimalnoSati) {
               System.err.println("Није јасна рачуница
                                   кад је број сати испод норме!");
               System.exit(1);
        }
        double plata = minimalnoSati*cenaSata;
        if(brojSati>minimalnoSati) {
               PrekovremeniRad prekovremeni = new PrekovremeniRad(cenaSata,
                             brojSati-minimalnoSati);
               plata+=prekovremeni.izracunajCenuPrekovremenog();
        }
        return plata;
    }

    private static class PrekovremeniRad{
        private static double koeficijentUvecanja=1.02;
        private int cenaSata;
        private int prekovremenoSati;

        private PrekovremeniRad(int cenaSata, int prekovremenoSati) {
               this.cenaSata=cenaSata;
               this.prekovremenoSati=prekovremenoSati;
        }

        private double izracunajCenuPrekovremenog() {
               double koeficijent = koeficijentUvecanja;
               double cenaPrekovremenih = 0;
               for(int i=0; i<prekovremenoSati; i++) {
                      double uvecanaCena = cenaSata*koeficijent;
                      if(uvecanaCena>maksimalnaCenaSata)
```

```
                        uvecanaCena=maksimalnaCenaSata;
                    cenaPrekovremenih+=uvecanaCena;
                    koeficijent*=koeficijentUvecanja;
                }
                return cenaPrekovremenih;
            }
        }
    }
```

Запажамо да се из метода `izracunajCenuPrekovremenog()` може приступити статичком пољу `ObracunPlata.maksimalnaCenaSata`, без обзира што је оно приватно. Такође, класа `ObracunPlata` у својим методима може користити конструктор и метод класе `PrekovremeniRad`, без обзира што су приватни.

```
package rs.math.oop.g10.p01.nestatickaUgnjezdenaObracunPlate;

public class ObracunPlataTest {
    public static void main(String[] args) {
        ObracunPlata op1 = new ObracunPlata(1000);
        ObracunPlata op2 = new ObracunPlata(1800);
        ObracunPlata op3 = new ObracunPlata(800);
        ObracunPlata op4 = new ObracunPlata(2200);
        System.out.println(String.format("%.2f", op1.izracunajPlatu(162)));
        System.out.println(String.format("%.2f", op2.izracunajPlatu(170)));
        System.out.println(String.format("%.2f", op3.izracunajPlatu(160)));
        System.out.println(String.format("%.2f", op4.izracunajPlatu(162)));

        //ObracunPlata.PrekovremeniRad pr=new ObracunPlata.PrekovremeniRad(1000,23);
        //System.out.println(pr.izracunajCenuPrekovremenog());
    }
}
```

Класа `ObracunPlataTest` демонстрира употребу класе `ObracunPlata`, при чему се статичка угњеждена класа не може користити независно због модификатора видљивости. Међутим, ако модификатор видљивости не би био `private`, што би такође имало смисла, прековремени рад би могао да се користи и ван тела припадајуће класе. Унутар класе `ObracunPlata`, за приступ класи `PrekovremeniRad`, није неопходно навести пуну путању до конструктора `ObracunPlata.PrekovremeniRad`. То би требало урадити ако би се класа инстанцирала ван класе `ObracunPlata`.
Следи резултат извршавања програма.

```
Редукујемо цену на износ 2000
162060.40
307554.62
128000.00
324000.00■
```

10.2. Унутрашње класе

Унутрашња класа се придружује инстанци класе која је обухвата па и она има директан приступ до свих поља и метода објекта који је садржи. Како је унутрашња класа

придружена инстанци, то она сама не може садржати статички члан. Објекти унутрашње класе постоје само у оквиру инстанце спољашње класе.
Када се дефинише унутрашња класа, она је члан спољашње класе на исти начин као и остали чланови (поља и методи). Унутрашња класа може имати модификаторе приступа као и остали чланови.

```
Spoljasnja spoljasnja = new Spoljasnja();
UnutrasnjaKlasa unutrasnja1 = spoljasnja.new Unutrasnja();
UnutrasnjaKlasa unutrasnja2 = spoljasnja.new Unutrasnja();
```

Као што се може видети, инстанца унутрашње класе се креира над спољашњом. Синтакса може бити краћа у ситуацијама када се инстанца унутрашње класе креира у нестатичком методу спољашње. Тада се може изоставити кључна реч `this` која упућује на инстанцу спољашње. Јасно је да у случају статичког метода ово није могуће.

```
class Spoljasnja{
    ...
    void f(){
       ...
       Unutrasnja unutrasnja1 = new Unutrasnja();
       Unutrasnja unutrasnja2 = this.new Unutrasnja();
       ...
    }
    ...
}
```

Поставља се питање када је примерено користити унутрашњу класу уместо стандардно дефинисане класе (у засебној датотеци). Оваква ситуација се може препознати онда када постоји јако блиска веза између две класе где једна класа ексклузивно користи услуге друге класе или је користи као свој градивни елемент. Следећи пример демонстрира ту употребу у контексту повезане листе. Главна сврха примера није демонстрација рада са колекцијама — њима ће бити посвећена специјална пажња у поглављу 14.

Пример 2. У секцији 8.9.2, примеру 24, дата је имплементација једноструко повезане листе ниски. Проблем са том имплементацијом је могућност креирања чворова листе и ван тела класе `PovezanaListaNiski`. Онемогућити овакво непожељно понашање применом концепта унутрашње класе.□

```
package rs.math.oop.g10.p02.jednostrukoPovezanaLista;

public class PovezanaListaNiski {

    public class Cvor {
        private String sadrzaj;
        private Cvor sledeci;

        private Cvor(String elem) {
               sadrzaj = elem;
               sledeci = null;
        }

        public String uzmiSadrzaj() {return sadrzaj; }

        public void postaviSadrzaj(String sadrzaj) { this.sadrzaj = sadrzaj; }
```

```
        public Cvor uzmiSledeci() { return sledeci; }

        public void postaviSledeci(Cvor sledeci) { this.sledeci = sledeci; }

        @Override
        public String toString() { return sadrzaj; }
    }

    private Cvor pocetak = null;
    private Cvor kraj = null;
    private Cvor tekuci = null;

    public PovezanaListaNiski() { }

    public PovezanaListaNiski(String elem) {
        if (elem != null)
                tekuci = kraj = pocetak = new Cvor(elem);
    }

    public PovezanaListaNiski(String[] elementi) {
        if (elementi == null)
                return;
        for (int i = 0; i < elementi.length; i++)
                dodajNaKraj(elementi[i]);
        tekuci = pocetak;
    }

    public void dodajNaKraj(String elem) {
        Cvor noviKraj = new Cvor(elem);
        if (pocetak == null)
                pocetak = kraj = noviKraj;
        else {
                kraj.postaviSledeci(noviKraj);
                kraj = noviKraj;
        }
    }

    public String ukloniSaKraja() {
        if (kraj == null)
                return null;
        if (pocetak == kraj) {
                Cvor jedini = kraj;
                tekuci = pocetak = kraj = null;
                return jedini.uzmiSadrzaj();
        }
        Cvor poslednji = kraj;
        Cvor pretposlednji = pocetak;
        while (pretposlednji.uzmiSledeci() != poslednji)
                pretposlednji = pretposlednji.uzmiSledeci();
        pretposlednji.postaviSledeci(null);
        kraj = pretposlednji;
        return poslednji.uzmiSadrzaj();
    }

    public void dodajNaPocetak(String elem) {
        Cvor noviPocetak = new Cvor(elem);
```

```
        if (kraj == null)
                tekuci = pocetak = kraj = noviPocetak;
        else {
                noviPocetak.postaviSledeci(pocetak);
                pocetak = noviPocetak;
                if (tekuci == null)
                        tekuci = pocetak;
        }
    }

    public String ukloniSaPocetka() {
        if (pocetak == null)
                return null;
        if (pocetak == kraj) {
                Cvor jedini = kraj;
                pocetak = kraj = null;
                return jedini.uzmiSadrzaj();
        }
        Cvor prvi = pocetak;
        pocetak = prvi.uzmiSledeci();
        return prvi.uzmiSadrzaj();
    }

    public boolean stigaoDoKraja() {
        if (tekuci == null)
                return true;
        if (tekuci == kraj)
                return true;
        return false;
    }

    public String uzmiPrvi() {
        tekuci = pocetak;
        return (tekuci == null) ? null : tekuci.uzmiSadrzaj();
    }

    public String uzmiSledeci() {
        if (!stigaoDoKraja())
                tekuci = tekuci.uzmiSledeci();
        return (tekuci == null) ? null : tekuci.uzmiSadrzaj();
    }

    public int brojCvorova() {
        int n = 1;
        Cvor tek = pocetak;
        if (tek == null)
                return 0;
        while (tek != kraj) {
                tek = tek.uzmiSledeci();
                n++;
        }
        return n;
    }

    @Override
    public String toString() {
```

```
        StringBuilder sb = new StringBuilder("[");
        Cvor iterator = pocetak;
        while (iterator != null) {
               sb.append(iterator + " ");
               iterator = iterator.sledeci;
        }
        return sb.toString().trim() + "]";
    }
}
```

Употреба унутрашње класе `Cvor` са приватним конструктором је погодна у овом примеру, јер се на тај начин онемогућава инстанцирање објеката те класе ван тела спољашње класе `PovezanaListaNiski`. Овде је класа `Cvor` проглашена за јавну. На тај начин се ван тела класе `PovezanaListaNiski` (чак и из другог пакета) може референсати на инстанцу класе `Cvor`.

```
package rs.math.oop.g10.p02.jednostrukoPovezanaLista;

import java.util.Scanner;

public class CitajPrikazi {
    private static void napuni(Scanner sc, PovezanaListaNiski lista) {
        while (sc.hasNext()) {
               String rec = sc.next();
               if (rec.equals("KPAJ"))
                      break;
               lista.dodajNaKraj(rec);
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        PovezanaListaNiski lista = new PovezanaListaNiski();
        System.out.println("Унеси текст (или реч КРАЈ ћирилицом за крај уноса):");
        napuni(sc, lista);
        sc.close();
        System.out.println("Број речи: " + lista.brojCvorova());
        System.out.println("Листа:\n" + lista);
        // Cvor c = new Cvor("пример");
    }
}
```

Иако је класа `Cvor` јавна, приватност њеног конструктора онемогућава креирање елемената ван тела класе `PovezanaListaNiski`. Из тог разлога је последња наредба `main()` метода коментарисана – у супротном се кôд не би компајлирао.

Следи испис добијен извршењем овог програма.

```
Унеси текст (или реч КРАЈ ћирилицом за крај уноса):
Ово
је
листа
речи
КРАЈ
Број речи: 4
Листа:
[Ово је листа речи] ■
```

Поред унутрашње класе, која се сматра чланицом спољне класе, постоје још две варијанте унутрашњих класа. У питању су локална унутрашња класа и анонимна класа.

10.2.1. Локалне унутрашње класе

Локална унутрашња класа није чланица спољашње класе, већ је чланица метода. Због тога се она дефинише и инстанцира у телу припадајућег метода. На овај начин је ниво енкапсулације додатно увећан у односу на обичну унутрашњу класу.
Дефиниција класе, таква да обухвата локалне унутрашње класе, дата је следећим металингвистичким формулама:

```
<тип класе> ::= [модификатор видљивости> ][final |abstract ]class <назив класе>
                              <проширивање><имплементација><тело класе>
<проширивање> ::= extends <назив класе>
<имплементација> ::= implements <листа назива интерфејса>
<листа назива интерфејса> ::= {<назив интерфејса>,}<назив интерфејса>
<назив интерфејса> ::= {<идентификатор>.}<идентификатор>
<тело класе> ::= {{(<дефиниција поља>|<дефиниција метода>
                    |<иницијализациони блок>|<конструктор>
                    |<угњеждена унутрашња класа>)}}
<дефиниција поља> ::=  [<модификатор видљивости> ][static ][final ]
                               <декларација и иницијализација променљивих>
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= [<модификатор видљивости> ][static ][final |abstract ]
                                <повратни тип><назив метода>(<параметри>)
<повратни тип> ::= <тип>|void
<назив метода>::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= [final ]<тип>
<назив параметра>::= <идентификатор>
<тело метода> ::= { <наредбе тела матода> }
<наредбе блока> ::= {<наредба тела метода>}
<наредба тела метода> ::= <наредба>|<тип класе>
<иницијализациони блок> ::= [static ]<блок>
<конструктор> ::= <име класе>(<параметри>)<блок>
<име класе> ::= <идентификатор>
<угњеждена унутрашња класа> ::= [static ]<тип класе>
```

Уочава се да је сада, оваквом нотацијом, допуштено да класа буде дефинисана унутар метода.
Локалне унутрашње класе су уведене јер је у неким ситуацијама потребно дефинисати класу само за потребе реализације појединачног метода. Притом та класа није од интереса за методе унутар других класа. Локална унутрашња класа може позивати променљиве декларисане у припадајућем методу, под условом да су те променљиве финалне.
Пример 3. Написати Јава класу и метод за проверу да ли број мобилног телефона има валидан запис у српским мобилним мрежама (то не гарантује и постојање броја). Поред тога, број је потребно довести до стандардног формата облика [позивни број]/[број]. Валидан запис подразумева да он почиње неким од доступних позивних бројева мрежа: 060, 061, 068 (A1), 062, 063, 069 (Yettel), 064, 065, 066 (MTC). Након тога се могу појављивати цифре, симболи ‘/’, ‘-’ и празан простор. Претпоставити (поједностављено) да је број валидан ако поред позивног броја има још 6 или 7 цифара. За потребе валидације и довођења у стандардни формат користити локалну унутрашњу класу. □

```
package rs.math.oop.g10.p03.lokalneUnutrasnjeValidacijaBrojaMobilnog;

public class ValidacijaBrojaMobilnog {
    static boolean validirajBrojMobilnog(String mobTekst) {
        int duzinaPozivnog = 3;
        int minBrojCifara = 6;
        int maksBrojCifara =7;
        String[] dozvoljeniPozivni = { "060", "061", "062", "063",
                        "064", "065", "066", "068", "069" };

        class BrojTelefona {
                String mobTekst;
                String mobTekstStd;

                BrojTelefona(String mobTekst) {
                        this.mobTekst = mobTekst;
                        standardizuj();
                }

                void standardizuj() {
                        mobTekstStd = mobTekst.replaceAll("[/|\\-|\s+]", "");
                        if (mobTekstStd.length() >= duzinaPozivnog)
                                mobTekstStd = mobTekstStd.substring(0, duzinaPozivnog)
                                + mobTekstStd.substring(duzinaPozivnog);
                        else
                                mobTekstStd = null;
                        System.out.println(mobTekstStd);
                }

                String pozivni() {
                        return mobTekstStd.substring(0, 3);
                }

                String broj() {
                        return mobTekstStd.substring(3);
                }
        }
        BrojTelefona brojTelefona = new BrojTelefona(mobTekst);
        if (brojTelefona.mobTekstStd == null)
                return false;
        String pozivni = brojTelefona.pozivni();
        boolean dozvoljen = false;
        for (String dp : dozvoljeniPozivni) {
                if (dp.equals(pozivni)) {
                        dozvoljen = true;
                        break;
                }
        }
        String broj = brojTelefona.broj();
        return dozvoljen && broj.length() >= minBrojCifara
                        && broj.length() <= maksBrojCifara;
    }

    public static void main(String[] args) {
        String[] brojevi = { "069/434233","065   /32 -3321-2",
                        "033 1223 555","067 332 - 2221"};
```

```
        for(String b : brojevi) {
                System.out.println(b);
                if(validirajBrojMobilnog(b))
                        System.out.println("Валидан");
                else
                        System.out.println("Није валидан");
                System.out.println("-----------------");
        }
    }
}
```

У реализацији метода `validirajBrojTelefona()` коришћена је локална унутрашња класа `BrojTelefona`. Она није у потпуности извршила посао валидације броја, али је допринела том циљу. Претпоставља се да ова класа нема никакву употребну вредност ван тела поменутог метода. Стога је оправдана њена дефиниција у виду локалне унутрашње класе.

Следи резултат извршавања програма.

```
069/434233
069434233
Валидан
-----------------
065   /32 -3321-2
0653233212
Валидан
-----------------
033 1223 555
0331223555
Није валидан
-----------------
067 332 - 2221
0673322221
Није валидан
-----------------■
```

10.2.2 Анонимне класе

Анонимне класе су попут локалних унутрашњих класа, само што нису именоване. Последица овога је обавеза истовременог дефинисања и инстанцирања анонимне класе – накнадно инстанцирање не би било могуће, јер класа нема назив. Из истог разлога није могуће нити извршити инстанцирање више пута. Анонимне класе, приликом компилације, добијају аутоматски генерисано име које није од интереса програмеру, али јесте компајлеру.

Најчешћа намена анонимне класе је једнократни пренос функционалности у позив метода. Будући да Јава не подржава рад са показивачима па самим тим ни са показивачима на методе (функције), пренос метода се реализује прослеђивањем објекта који у себи садржи дефиницију метода. Овакви објекти се називају функционали. Наредни пример демонстрира пренос метода поређења који се користи приликом сортирања низа објеката, попут функције поређења која се прослеђује `qsort()` функцији у C-у.

Пример 4. Написати Јава програм који сортира низ ниски растуће према суми ASCII вредности ниске. Ако две ниске имају исту суму ASCII вредности, као секундарни критеријум поређења, користити стандардно растуће лексикографско уређење. За

сортирање користити уграђени метод `Arrays.sort()`, уведен у секцији 7.6, и проследити му одговарајући функционал за поређење.□

```
package rs.math.oop.g10.p04.anonimneKlasePoredjenjePriSortiranju;

import java.util.Arrays;
import java.util.Comparator;

public class SortirajNiske {
    static int sumaAscii(String s) {
        int suma = 0;
        for(int i=0; i<s.length(); i++)
               suma+=s.charAt(i);
        return suma;
    }

    public static void main(String[] args) {
        String[] niske = new String[] {"Анонимне класе", "су попут",
                        "локалних унутрашњих класа", "с тим што", "нису именоване.",
                        "Ово за последицу", "има да се", "анонимна класа",
                        "истовремено", "и", "дефинише и", "инстанцира.",
                        "Јасно је да се", "инстанцирање", "не може извршити",
                        "више пута,","јер не постоји назив",
                        "класе па самим тим ни конструктора."};

        Arrays.sort(niske, new Comparator() {
               @Override
               public int compare(Object o1, Object o2) {
                        if(!(o1 instanceof String) || !(o2 instanceof String)) {
                                System.err.println("Сви објекти морају бити ниски.");
                                System.exit(1);
                        }
                        String s1 = (String) o1;
                        String s2 = (String) o2;
                        int razlikaAscii = sumaAscii(s1)-sumaAscii(s2);
                        if(razlikaAscii==0)
                                return s1.compareTo(s2); // стандардно лексикографско
                        else
                                return razlikaAscii;
               }
        });
        for(String niska : niske)
               System.out.println(niska);
    }
}
```

Метод `Arrays.sort()`, поред стандардног начина позивања којем се само проследи низ, омогућава и прослеђивање другог аргумента који је критеријум за поређење (секција 9.3.2). Ово је погодно у ситуацијама када је класа чије објекте сортирамо:

1. већ дефинисана и програмер нема контролу над њеним кодом или
2. већ имплементира интерфејс `Comparable`, чиме је прецизиран примарни начин поређења.

Конкретно, класа `String` испуњава оба ова критеријума па је у решењу задатка оправдана употреба инстанце анонимне класе, тј. критеријума за поређење.

Следи резултат извршавања програма.

```
и
има да се
су попут
с тим што
више пута,
дефинише и
инстанцира.
истовремено
Јасно је да се
инстанцираће
Анонимне класе
анонимна класа
нису именоване.
Ово за последицу
не може извршити
јер не постоји назив
локалних унутрашњих класа
класе па самим тим ни конструктора.■
```

10.3. Резиме

У овом поглављу описане су угњеждене класе – концепт који се не сматра есенцијалним за програмирање. Наиме, могу се писати квалитетни објектно оријентисани програми и без употребе овог концепта. Међутим, на дужи рок, употреба угњеждених класа, посебно унутрашњих и њених подваријанти (локалне и анонимне) може побољшати организацију кода, повећати његову прегледност и унапредити примену принципа отворености и затворености кода (једног од SOLID принципа из секције 9.4.1). Вероватно најкориснија, и нешто чешће примењивана варијанта угњеждене класе, јесте анонимна класа. Она је нашла примену у програмирању графичких апликација, где се користи за реализацију метода који реагују на догађаје (клик миша, притисак на тастатури итд.), али и за пренос функционалности у метод у општем случају – алтернатива показивачима на функције.

10.4. Питања и задаци

1. Шта су угњеждене класе и које су предности употребе угњеждених класа?
2. Објаснити концепт статичке угњеждене класе. Којим елементима спољашње класе може да приступа статички угњеждена класа? Примером илустровати употребу статички угњеждене класе.
3. Које су разлике између статичке угњеждене класе и унутрашње класе? Илустровати примером.
4. Када је корисно дефинисати унутрашњу класу, а када локалну унутрашњу класу? Илустровати примерима.
5. Шта су анонимне класе и када се користе?

11. Изузеци и тврдње

Један од кључних изазова у програмирању је писање програма без грешака или тзв. багова (енг. bugs). Такође је пожељно да се програми не гасе, посебно у ситуацијама када људски животи зависе од извршавања програма (на пример, аутопилоти и контролни системи на железницама) или гашење може да изазове велики финансијски губитак (на пример, банкарски и берзански софтвер). Уколико би за написани програм постојао формални доказ (верификација) да никад не може да погреши, онда би једини ризици по његово успешно извршење били спољни фактори попут нестанка струје, временских непогода и слично. Формална верификација програма је тешка, скупа, а често и немогућа у императивним програмским језицима. Из овог разлога, у Јави се прибегава употреби два начина за контролу грешака у току рада програма:

1. Изузеци — механизам сигнализирања и пропратног реаговања на грешке који је користан у фази употребе програма.
2. Тврдње — механизам обуставе рада програма приликом појаве грешке. Овај механизам има за циљ очување интегритета програма и користан је у фази развоја и тестирања програма.

11.1. Изузеци

Приликом извршавања програма, ако се појави грешка, настаје изузетна ситуација и стога се овај догађај назива изузетак. Пошто они настају када у Јава програмима „ствари крену наопако“, изузетке треба озбиљно размотрити приликом осмишљавања и реализације програма. Две кључне користи од употребе изузетака су:

1. раздвајање кода који обрађује неочекиване ситуације од кода који се извршава када је све очекивано;
2. присиљавање програмера да размотри и реагује на одређене врсте грешака.

Не треба све грешке у програмима третирати као изузетке – довољно је само неуобичајене или катастрофалне. На пример, ако корисник не унесе исправан улазни податак, за то не треба користити изузетке. У том случају је рационалније обавестити корисника о лошем уносу и евентуално му дати шансу да податак поново унесе. Руковање изузецима укључује много додатног процесирања и самим тим успорава извршавања програма.

Унутар Јава програма изузетак се моделује као објекат који у себи носи информацију о насталој непредвиђеној ситуацији. Када се деси непредвиђена ситуација унутар тела неког метода, она ја праћена наредбом за тзв. избацивање (креирање) изузетка. Обично се каже да је метод „избацио“ изузетак. Надаље, изузетак (објекат) може бити:

1. прослеђен (пропагиран) методу-позиваоцу, тј. методу који је позвао актуелни метод;
2. обрађен (или „ухваћен“) унутар неког од метода.

Пример 1. Написати Јава програм који покушава да приступи елементу низа ван дозвољених граница. Ухватити изузетак и обрадити га унутар `main()` метода. □

```
package rs.math.oop.g11.p01.izuzeciIndeksVanGranica;

public class IndeksVanGranica {
    public static void main(String[] args) {
        int a[] = new int[2];
```

```
        System.out.println("Приступам елементу:" + a[3]);
    }
}
```

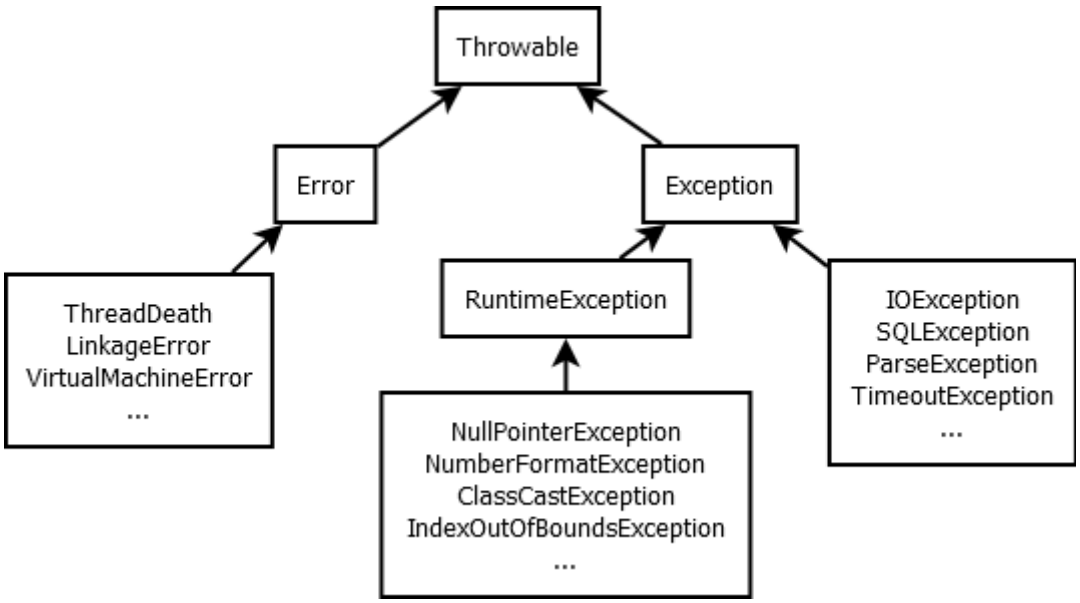
Изузетак објекат је, у овом случају, избачен приликом индексирања, односно током приступа елементу низа са задатим индексом.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of
bounds for length 2
    at
rs.math.oop.g11.p01.izuzeciIndeksVanGranica.IndeksVanGranica.main(IndeksVanGranica.java:7
)
```

Индексирање у Јави се реализује преко метода, тако да је овде реч о пропагацији изузетка из метода индексирања ка `main()` методу. Будући да није било никакве обраде изузетка од стране `main()` метода, програм се завршава исписом претходно наведене поруке на стандардном излазу за грешке `System.err`.■

11.1.1 Класе изузетака

Изузетак је увек објекат неке поткласе класе `Throwable`. То важи и за изузетке које сами дефинишемо, као и за стандардне, већ уграђене изузетке. Две директне поткласе класе `Throwable` су класе `Error` и `Exception` – оне покривају све стандардне изузетке.



`Throwable` класа и њене потклсе

Изузеци класе Error

За изузетке класе `Error` и њених поткласа од програмера се не очекује да предузима неку акцију, тј. не очекује се да их обрађује. Ове врсте грешака се ретко јављају, али у Јави се и оне контролишу, односно, саопштава се када се појаве. Класа `Error` има неколико директних поткласа, а неке значајније су:

- `ThreadDeath` – избацује изузетак када се нит, која се извршава, намерно заустави.
- `LinkageError` – указује на озбиљне проблеме са класама у програму, на пример, некомпатибилност међу класама или покушај креирања објекта непостојеће класе и слично.

- `VirtualMachineError` – садржи четири потклase изузетака који се избацују када се деси катастрофални пад JVM.

Изузеци класа изведених из `LinkageError` и `VirtualMachineError` су резултат катастрофалних догађаја и услова. У таквим ситуацијама програмер само може да констатује поруку о грешци и евентуално да је запамти у датотеци, бази података или слично. На основу поруке треба да покуша да схвати шта је у написаном коду могло да изазове такав проблем како би се проблем отклонио.

Пример 2. Написати Јава програм који демонстрира рад нетачне рекурзивне имплементације метода за рачунање факторијела (нема изласка из рекурзије). □

```
package rs.math.oop.g11.p02.izuzeciPrekoracenjeStekMemorije;

public class FaktorijelBezUslovaIzlaska {
    static int faktorijelBezUslovaIzlaska(int n) {
        return n * faktorijelBezUslovaIzlaska(n - 1);
    }

    public static void main(String[] args) {
        System.out.println(faktorijelBezUslovaIzlaska(10));
    }
}
```

Настаје прекорачење стека `StackOverflowError`, јер рекурзивни метод за факторијел нема услов изласка из рекурзије па се креирање нових стек оквира, за сваки наредни позив метода, наставља, све док се не потроши целокупна доступна стек меморија.

```
Exception in thread "main" java.lang.StackOverflowError
    at
rs.math.oop.g11.p02.izuzeciPrekoracenjeStekMemorije.FaktorijelBezUslovaIzlaska.faktorijel
BezUslovaIzlaska(FaktorijelBezUslovaIzlaska.java:6)
    at
rs.math.oop.g11.p02.izuzeciPrekoracenjeStekMemorije.FaktorijelBezUslovaIzlaska.faktorijel
BezUslovaIzlaska(FaktorijelBezUslovaIzlaska.java:6)
    at
rs.math.oop.g11.p02.izuzeciPrekoracenjeStekMemorije.FaktorijelBezUslovaIzlaska.faktorijel
BezUslovaIzlaska(FaktorijelBezUslovaIzlaska.java:6)
...■
```

Изузеци класе RuntimeException

За скоро све изузетке, представљене потклasама класе `Exception`, може се у програм укључити кôд који ће их обрађивати, осим за објекте класе `RuntimeException`. Преводилац допушта да програмер игнорише ове изузетке, јер су они најчешће последица логичких грешака у написаном програмском коду. Тиме се имплицитно препоручује измена програма. На пример, приступ елементу чији индекс је ван граница низа се сигнализира изузетком класе која је потклаcа `RuntimeException`. Очигледно је да овај изузетак може да се спречи одговарајућим кодом па Јава не стимулише његово обрађивање.

Пример 3. Написати Јава програм у којем се спречава појава референсирања на елемент чији индекс је ван граница индекса низа, тј. спречава се појава изузетка класе `ArrayIndexOutOfBoundsException`.□

```
package rs.math.oop.g11.p03.izuzeciSprecavanjeIndeksaVanGranica;
```

```
import java.util.Scanner;

public class SprecavanjeIndeksaVanGranica {
    public static void main(String[] args) {
        int a[] = new int[] { 1, 2, 3, 4, 5 };
        System.out.println("Унесите индекс елемента којем приступате:");
        Scanner skener = null;
        skener = new Scanner(System.in);
        int i = skener.nextInt();
        if (i < 0 || i >= a.length)
               System.err.println("Индекс ван граница.");
        else
               System.out.println("Број на траженом индексу је " + a[i]);
        skener.close();
    }
}
```

Добија се следећи извештај у случају индекса ван граница.

```
Унесите индекс елемента којем приступате:
10
Индекс ван граница.
```

Док се у супротном исписује елемент на траженој позицији.

```
Унесите индекс елемента којем приступате:
3
Број на траженом индексу је 4■
```

Постоји велики број класа које су директне потклase класе `RuntimeException`, а неке од познатијих су:

- `ArithmeticException` — недозвољена ситуација у примени аритметичких операција, на пример, дељење нулом.
- `IndexOutOfBoundsException` — индекс низа којем се приступа није валидан.
- `NegativeArraySizeException` — алокација низа са негативном димензијом.
- `NullPointerException` — покушај приступа методу или пољу инстанцне променљиве која има специјалну референтну вредност `null`.
- `ClassCastException` — покушај конверзије инстанцне променљиве у недозвољени тип.

Пример 4. Написати Јава програм у којем се демонстрира покушај конверзије `Integer` објекта у `Boolean` објекат.□

```
package rs.math.oop.g11.p04.izuzeciKonverzijaUPogresanObjektniTip;

public class KonverzijaUPogresanObjektniTip {
       public static void main(String[] args) {
              Integer broj = 5;
              Object objekat = broj;
              Boolean logickoSlovo = (Boolean) objekat;
       }
}
```

Резултат извршавања овог програма је информисање корисника о изузетку типа `ClassCastException`.

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot
be cast to class java.lang.Boolean (java.lang.Integer and java.lang.Boolean are in module
```

```
java.base of loader 'bootstrap')
        at
rs.math.g10.p04.izuzeciKonverzijaUPogresanObjektniTip.KonverzijaUPogresanObjektniTip.main
(KonverzijaUPogresanObjektniTip.java:8)
```

Јава компајлер не преиспитује овакве некоректне конверзије тако да се оне могу само детектовати, али не и спречити. Стога је на програмеру да логичке проблеме, попут ових, отклони и тиме спречи појаву оваквог или сличних изузетака. Хватањем оваквих изузетака не постиже се пуно, тј. само се одлаже суочавање са логичким проблемом који постоји у програму.■

Изузеци класе Exception

Већ је истакнуто да програмер не може да обрађује изузетке класе `Error`. Са друге стране, када се деси `RuntimeException`, могућа је поправка, али то треба да буде само привремена мера, а да се прави проблем реши у новој верзији програма. Стога, у оба ова случаја, Јава компајлер не захтева постојање кода за обраду изузетака.
Када су у питању сви остали изузеци класе `Exception` или њених потклаcа, они нису последица лошег програмирања, попут изузетка класе `RuntimeException`, нити су непоправљиве попут изузетака `Error`. Стога ће Јава компајлер, у тим ситуацијама, проверити да ли је примењен један од наредна два механизма за обраду изузетака:

1. хватање изузетка (`try-catch` блок);
2. прослеђивање изузетка (наредба `throws`).

Уколико није урађено ни једно ни друго, програм неће бити преведен.
Постоји велики број директних потклаcа класе `Exception` (поред `RuntimeException`), а неке од њих су:

- `IOException` — општи изузетак у вези са улазно/излазним операцијама. На пример, класа `FileNotFoundException`, као потклаcа ове класе, односи се на ситуацију када тражена датотека не постоји.
- `ParseException` — указује на проблем у парсирању текста.
- `SQLException` — општи проблем при повезивању са базом података или при прављењу SQL упита, обради резултата упита и слично.
- `TimeoutException` — изузетак који се избацује када истекне време за чекање неке блокирајуће операције — најчешће се појављује у вишенитном програмирању.

11.1.2. Руковање изузецима

Претпоставимо да метод `f()` позива други метод `g()` који може избацити изузетак који није `RuntimeException` нити `Error` типа. Нека је изузетак, на пример, класе `IOException`.

```
double f(){
  ...
  g(); // избацује IOException
  ...
}
```

Један начин реаговања на изузетак, у овом случају, може бити додавање специјалне наредбе `throws` у оквиру декларације метода, тј. прослеђивање изузетка методу-

позиваоцу. Та наредба је праћена типом изузетка који може бити избачен унутар метода (у овом случају је то `IOException`).

```
double f() throws IOException { ... }
```

Ако постоји више изузетака који могу бити избачени, онда се формира листа типова раздвојених зарезом. На пример, нека у телу метода `f()` постоји и позив метода `h()` који може избацити `FileNotFoundException` или `SQLException`.

```
double f(){
  ...
  g(); // избацује IOException
  ...
  h(); // избацује FileNotFoundException или SQLException
  ...
}
```

У том случају би декларација метода `f()` могла да буде нека од следећих.

```
double f() throws IOException, FileNotFoundException, SQLException {...}
double f() throws IOException, SQLException {...}
double f() throws Exception {...}
```

Друга декларација је могућа, јер је `FileNotFoundException` поткласа класе `IOException`. Такође, све три класе су потклсе класе `Exception` па је валидна и трећа декларација.

Други начин реаговања је обрада изузетака у оквиру метода од интереса. За те сврхе потребно је укључити три блок-наредбе при чему је трећа опциона:

1. `try` блок – кôд који може да избаци један или више изузетака мора бити унутар овог блока.
2. `catch` блок – обухвата кôд који је намењен руковању изузецима одређеног типа, који могу бити избачени у придруженом `try` блоку.
3. `finally` блок – увек се извршава пре него се метод заврши, без обзира да ли се десио изузетак у оквиру `try` блока или не.

```
double f(){
  ...
  try{
     ...
     g(); // избацује IOException
     ...
  }catch(IOException ex){
     ...
  }
  ...
  try{
     ...
     h(); // избацује FileNotFoundException или SQLException
     ...
  }catch(Exception ex){
    ...
  }
  ...
}
```

Могуће је и комбиновати поменута два начина, тј. неке изузетке само проследити даље, а неке хватати. На пример, претходно описан метод `f()` је могао да обради само `SQLException`, а остале да проследи помоћу `throws` наредбе.

```
double f() throws IOException, FileNotFoundException{
  ...
  g(); // избацује IOException
  ...
  try{
     ...
     h(); // избацује FileNotFoundException или SQLException
     ...
  }catch(SQLException ex){
    ...
  }
  ...
}
```

Уколико се за изузетак, који није типа `RuntimeException` или `Error`, не уради ни једно ни друго, доћи ће до грешке приликом компилације.
У наредним секцијама ће бити детаљније објашњени уведени механизми.

Блок try

Када треба да се ухвати изузетак, кôд метода који избацује изузетак се обухвата `try` блоком. Блок `try` чини кључна реч `try` за којом следи пар витичастих заграда и унутар њих кôд који може избацити изузетак.

```
try {
   // кôд који може избацити један или више изузетака
}
```

Блок `try` је неопходан и када желимо да хватамо изузетке класе `Error` или `RuntimeException`.
У оквиру `try` блока се могу наћи и наредбе које не производе изузетак, што суштински значи да се читав кôд метода може обухватити `try` блоком. Ипак, препорука је да се ово не ради, већ да се `try` блоком обухвата минимална секвенца наредби која производи изузетак или изузетке од интереса.

Блок catch

Кôд за руковање изузетком одређеног типа треба да буде ограђен витичастим заградама у `catch` блоку. Блок `catch` се мора налазити непосредно иза `try` блока који садржи кôд који може избацити изузетак од интереса. Блок `catch` се састоји од кључне речи `catch` праћене једним параметром унутар облих заграда. Смисао тог параметра је идентификација типа изузетка којим блок рукује. Ово прати кôд за руковање изузетком који се налази унутар пара витичастих заграда.

```
try {
  // кôд који може избацити један или више изузетака
}catch(FileNotFoundException e) {
  // кôд за руковање изузетком типа FileNotFoundException
}
// извршавање се наставља овде...
```

Вишеструки catch блок

У претходном примеру, у `catch` блоку се руковало само изузецима типа `FileNotFoundException`. Ако могу бити избачени и други (изузев `Error` и `RuntimeException`), претходни кôд се неће успешно превести.

Параметар за `catch` блок мора бити објекат класе `Throwable` или неке њене поткласе. Ако објекат који се задаје као параметар `catch` блока припада класи која има потклассе, од `catch` блока се очекује да процесира изузетке тог типа, али и свих њених потклаcа. Ако `try` избацује неколико различитих врста изузетака, тада је након `try` блока потребно поставити више `catch` блокова за руковање њима.

```
try {
  // кôд који може избацити један или више изузетака
}catch(FileNotFoundException e) {
  // кôд за руковање изузетком типа FileNotFoundException
}catch(IOException e) {
  // кôд за руковање изузетком типа IOException
}
// извршавање се наставља овде...
```

У претходном примеру је други `catch` блок намењен хватању `IOException` изузетака који припада наткласи класе `FileNotFoundException`. Значи, ако је изузетак класе `FileNotFoundException` или припада некој њеној потклаcи, онда ће први `catch` блок обрадити изузетак. У супротном, проверава се да ли изузетак припада хијерархији класа `IOException`, и након тога ће се обрадити у другом `catch` блоку. Дакле, уђи ће се највише у један `catch` блок. У навођењу `catch` блокова потребно је водити рачуна о хијерархији изузетака и увек прво наводити оне специфичније па потом оне општије.

Блок finally

Природа изузетака је таква да се извршавање `try` блока прекида по избацивању изузетка, без обзира на значај кода који следи наредбу у којој је изузетак избачен. Тиме се ствара могућност да изузетак изазове неконзистентно стање делова програма. На пример, може се догодити да се отвори датотека, и да се, пошто је избачен изузетак, не извршава кôд за затварање те датотеке.

Блок `finally` обезбеђује решење овог или сличних проблема. Наредбе овог блока се извршавају увек, без обзира да ли је у оквиру придруженог `try` блока избачен изузетак. Ако нема `catch` блокова, `finally` блок се смешта непосредно након `try` блока, у супротном се смешта непосредно након свих придружених `catch` блокова. Није смислена употреба `try` блока без придруженог бар једног `catch` или `finally` блока, тако да се програм, у том случају, не може ни превести. Уколико је коришћењем `return` наредбе враћена нека вредност унутар `finally` блока, то поништава `return` наредбу која је евентуално извршена у `try` блоку. Структура комплетне `try-catch-finally` наредбе је следећа.

```
try{
  // кôд који може избацити изузетке...
}catch(ExceptionType1 e) {
  // ...
}catch(ExceptionType2 e) {
  // ...
}finally{
```

```
  // кôд који се увек извршава након try-блока
}
// извршавање се наставља овде...
```

Бекусовом нотацијом исказано наредба `try` има следећу структуру:

```
<наредба try> ::= try <блок>catch(<тип класе> <изузетак>)<блок>
                          {catch(<тип класе> <изузетак>)<блок>}
                          [finally<блок>]
<изузетак> ::= <идентификатор>
```

Пример 5. Написати Јава програм који у петљи прихвата текст са стандардног улаза. У свакој итерацији петље покушати парсирање унетог текста у датум-формат “dd.MM.yyyy”. Корисник треба да уноси текст све док га не унесе у валидном формату — након тога прекинути извршавање петље, односно програма. □

```
package rs.math.oop.g11.p05.izuzeciParsiranjeDatuma;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.Scanner;

public class ParsiranjeDatuma {
    public static void main(String[] args) {
        LocalDate datum = null;
        Scanner skener = null;
        DateTimeFormatter datumFormat = DateTimeFormatter.ofPattern("dd.MM.yyyy");
        Boolean unetValidanFormat = false;
        try {
               skener = new Scanner(System.in);
               while (!unetValidanFormat) {
                   try {
                       System.out.println("Датум dd.MM.yyyy:");
                       String datumString = skener.next();
                       datum = LocalDate.parse(datumString, datumFormat);
                       System.out.println("Валидан датум: " + datum);
                       unetValidanFormat = true;
                    } catch (DateTimeParseException e) {
                       System.out.println("Погрешан формат датума!");
                    }
              }

        } finally {
               skener.close();
        }
    }
}
```

У случају уноса формата датума који није у складу са очекиваним, програм ће детектовати изузетак и потом исписати грешку кориснику. За разлику од претходних примера, овде је приказан сценарио у којем се програм не зауставља, већ се кориснику даје прилика да поново унесе податак у траженом формату. Овде програм користи угњеждени `try-catch` блок унутар петље која се извршава, зависно од тачности индикатора за валидан унос. Спољни `try-finally` блок служи да се отворени `Scanner` објекат увек затвори, без обзира на потенцијалне неухваћене изузетке. Пример једног извршавања претходног кода је дат испод.

```
Датум dd.MM.yyyy:
10-12-2022
Погрешан формат датума!
Датум dd.MM.yyyy:
10.12.2022
Валидан датум: 2022-12-10■
```

11.1.3. Прослеђивање (пропагирање) изузетака

У ситуацијама када се у неком методу појави изузетак, програмер може да одлучи да тај изузетак не хвата, већ да га проследи (пропагира) у метод-позивалац. Мотивација за такву одлуку може бити боље познавање околности под којима је изузетак настао у методу-позиваоцу, па се унутар њега може боље разрешити проблем. Као што је раније објашњено, прослеђивање изузетка у метод-позивалац се реализује помоћу кључне речи `throws`, иза које следи листа изузетака.
Стога је неопходно ажурирати металингвистичке формуле које описују метод, како би се допустило да метод избацује изузетак, То се може постићи на следећи начин:

```
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= [<модификатор видљивости> ][static ][final |abstract ]
                                <повратни тип><назив метода>(<параметри>)<избацује>
<повратни тип> ::= <тип>|void
<назив метода>::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= [final ]<тип>
<назив параметра>::= <идентификатор>
<избацује> ::= throws <типови изузетака>
<типови изузетака> ::= {<тип изузетка>,}<тип изузетка>
<тип изузетка> ::= <назив класе>
<тело метода> ::= { <наредбе тела матода> }
<наредбе блока> ::= {<наредба тела метода>}
<наредба тела метода> ::= <наредба>|<тип класе>
```

Пример 6. Написати Јава програм за решавање проблема из примера 5, при чему парсирање треба издвојити у засебан статички метод. (Напомена: с обзиром да овај метод неће имати контролу над главном петљом, у којој се уноси текст од стране корисника, пропагација изузетка у `main()` метод има више смисла него реаговање на изузетак унутар метода за парсирање.) □

```
package rs.math.oop.g11.p06.izuzeciPropagiranje;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.Scanner;

public class ParsiranjeDatumaPropagiranjeIzuzetka {
    final static DateTimeFormatter datumFormat =
                DateTimeFormatter.ofPattern("dd.MM.yyyy");

    static LocalDate parsirajDatum(String datumString) throws DateTimeParseException{
        LocalDate datum = LocalDate.parse(datumString, datumFormat);
        return datum;
    }
```

```
    public static void main(String[] args) {
        LocalDate datum = null;
        Scanner skener = null;

        Boolean unetValidanFormat = false;
        try {
               skener = new Scanner(System.in);
               while (!unetValidanFormat) {
                      try {
                             System.out.println("Datum dd.MM.yyyy:");
                             String datumString = skener.next();
                             datum = parsirajDatum(datumString);
                             System.out.println("Валидан датум: " + datum);
                             unetValidanFormat = true;
                      } catch (DateTimeParseException e) {
                             System.out.println("Погрешан формат датума!");
                      }
               }
        } finally {
               skener.close();
        }
    }
}
```

Резултат извршавања је исти као раније.■

11.1.4. Избацивање изузетака

До сада је било речи о хватању изузетака и њиховом прослеђивању. Да би се било шта од овога десило, изузетак се најпре мора избацити. Методи унутар стандардних Јава класа, који избацују изузетке, увек у својим имплементацијама имају наредбу `throw` (не `throws`) која изазива избацивање изузетка. На пример, метод за дељење, скраћено записан као оператор "/", може у својој реализацији имати проверу да ли је делилац нула. Ако јесте, применом наредбе `throw` се креира објекат класе `ArithmeticException` са одговарајућом поруком да је у питању дељење нулом. Слично, програмери могу креирати сопствене изузетке ако процене да је то потребно.
Бекусовом нотацијом исказано наредба `throw` има следећу структуру:

```
<наредба throw> ::= throw <изузетак>
<изузетак> ::= <објекат>
```

Пример 7. Написати Јава програм који демонстрира сабирање одговарајућих елемената два дводимензионална низа. (Елементи су одговарајући ако имају исте индексе.). Уколико се сабирање односи на два низа који немају идентичне димензије, потребно је избацити нови тип изузетка. □

```
package rs.math.oop.g11.p07.izuzeciKorisnickiDefinisan;

public class Niz2DIzuzetak extends Exception {
    public Niz2DIzuzetak(String poruka) {
        super(poruka);
    }
}
```

Дефинисана је нови класа изузетка `Niz2DIzuzetak` тако што је изведена из класе `Exception`. Алтернативно, програмер је могао да искористи неку постојећу класу изузетка па чак и најопштију `Exception`. Међутим, зарад суптилнијег реаговања на изузетке и употребе вишеструких `catch` блокова, смисленије је користити специјализовану хијерархију изузетака (посебно у оквиру већих софтверских пројеката).

```
package rs.math.oop.g11.p07.izuzeciKorisnickiDefinisan;

public class Nizovid2D {
    static double[][] saberi(double[][] a, double[][] b) throws Niz2DIzuzetak {
        if (a.length != b.length)
               throw new Niz2DIzuzetak("Лоше спољне димензије.");
        double[][] c = new double[a.length][];
        for (int i = 0; i < a.length; i++) {
               if (a[i].length != b[i].length)
                      throw new Niz2DIzuzetak("Лоше унутрашње димензије.");
               c[i] = new double[a[i].length];
               for (int j = 0; j < c[i].length; j++)
                      c[i][j] = a[i][j] + b[i][j];
        }
        return c;
    }

    public static void main(String[] args) {
        double[][] m1 = new double[][] { { 1, 2, 3 }, { 4, 5, 6 } };
        double[][] m2 = new double[][] { { 1, 2 }, { 3, 4 }, { 5, 6 } };
        double[][] m3 = new double[][] { { 1, 1, 1 }, { 1, 1, 1 } };
        try {
               double[][] m4 = saberi(m1, m3);
               System.out.println("Прво сабирање успело.");
               double[][] m5 = saberi(m1, m2);
               System.out.println("Друго сабирање успело.");
        } catch (Niz2DIzuzetak e) {
               System.err.println(e);
        }
    }
}
```

Смисао класе `Nizovi2D` је сличан смислу класе `Arrays`, с тим што се уместо са једнодимензионалним низовима, овде ради са дводимензионалним. Претпоставимо да је у оквиру класе `Nizovi2D` реализован још велики број операција над дводимензионалним низовима – тада би вероватно имало смисла дефинисати засебну хијерархију изузетака, који су сви изведени из `Niz2DIzuzetak`.

Следи пример извршавања наведеног програма.

```
Прво сабирање успело.
rs.math.g10.p07.izuzeciKorisnickiDefinisan.Niz2DIzuzetak: Лоше спољне димензије.■
```

11.1.5. Препоруке за рад са изузецима

У претходном тексту су већ наведене неке препоруке за употребу изузетака. Неке од најбитнијих су:

- Блок `finally`, са или без постојећих `catch` блокова, треба користити за затварање отворених ресурса, на пример, `Scanner` објекта.

- Изузетке из потклаcа класе `RuntimeException` као и класе `Error` не треба обрађивати суштински, већ их само треба документовати у фази употребе програма (продукцији). Прве из разлога што указују на грешку у програмирању коју је боље спречити у новој верзији софтвера, а друге, јер њиховим хватањем није могуће решити проблем. На пример, не можемо утицати на извршавање програма ако се појави изузетак класе `StackOverflowError`.
- Преферирати што специфичније изузетке који боље описују проблематичну ситуацију — ово се спроводи, не само кроз креирање нових класа за изузетке, већ и кроз адекватно описивање проблема путем поруке која се прослеђује кроз конструктор.
- Унутар вишеструких `catch` блокова најпре хватати специфичне изузетке па потом општије с обзиром да се улази у највише један `catch` блок. Обрнути редослед би довео до потпуног игнорисања специфичнијих изузетака.
- Не треба хватати најопштији могући изузетак из класе `Throwable` — овим ће се маскирати било какви проблеми у раду апликације. Програм ће наставити да се извршава, али са потенцијално озбиљним проблемима који су занемарени (попут преоптерећења меморије).
- У фази програмирања не игнорисати изузетке, посебно изведене из класе `RuntimeException`. Они често указују на проблеме који ће у фази употребе програма постати још израженији.
- Не претеривати са употребом изузетака будући да њихово прослеђивање и обрада може изазвати одређено успорење рада програма.

11.2. Тврдње

Тврдња је оператор који омогућава проверу испуњености неког услова у програму. На пример, може се проверавати ненегативност индекса елемента у низу, да ли је површина круга позитивна, да ли је брзина кретања честице у симулацији мања од брзине светлости и слично. Уколико услов није испуњен, програм аутоматски избацује грешку и прекида се. Овај ригорозни приступ проверавању испуњености услова помаже програмеру, током програмирања, у разрешавању грешака и повећавању степена поверења у то да програм заиста ради оно што се очекује.

11.2.1. Наредба assert

Тврдње се реализују помоћу оператора `assert` и класе `java.lang.AssertionError`. Тврдња започиње кључном речи `assert` након чега следи логички израз и потом опциона порука.

```
<наредба assert> ::= asser <логички израз>[ <порука>]
<порука> ::= <литерал-ниска>
```

Ако је логички израз тачан, ништа се не дешава и извршавање се наставља од наредне наредбе. Са друге стране, ако је логички израз нетачан, креира се објекат класе `AssertionError`. Програм проверава, у току извршавања, тврдње само ако су оне активиране, што се постиже задавањем аргумента `-ea` (енг. enable assertions) виртуелној машини приликом покретања програма. На овај начин се једноставно, без измене програмског кода, може прелазити из режима употребе у режим програмирања и отклањања грешака.

Пример 8. Написати Јава програм који помоћу тврдње проверава да ли је број ненегативан. □

```
package rs.math.oop.g11.p08.tvrdnjeNenegativanBroj;

public class NenegativanBrojTvrdnja {
          public static void main(String[] args)
          {
             int x = -1;
             assert x >= 0;
          }
}
```

Испис доста подсећа на онај који се добија приликом избацивања изузетка који није обрађен, тј. ухваћен у програму.

```
Exception in thread "main" java.lang.AssertionError
       at
rs.math.g10.p08.tvrdnjeNenegativanBroj.NenegativanBrojTvrdnja.main(NenegativanBrojTvrdnj
a.java:8)■
```

Пример 9. Написати Јава програм који ради исто што и програм у примеру 8 са том разликом што помоћу тврдње треба да се прикаже и опис грешке. □

```
package rs.math.oop.g11.p09.tvrdnjeSaPorukomNenegativanBroj;

public class NenegativanBrojTvrdnjaSaPorukom {
       public static void main(String[] args)
       {
           int x = -1;
           assert x >= 0 : "x < 0";
       }
}
```

Овде смо употребили други облик оператора `assert` који укључује и додатну поруку. Покретањем програма добија се следећи испис.

```
Exception in thread "main" java.lang.AssertionError: x < 0
       at
rs.math.g10.p09.tvrdnjeNenegativanBrojSaPorukom.NenegativanBrojTvrdnjaSaPorukom.main(Nene
gativanBrojTvrdnjaSaPorukom.java:8)■
```

Честа употреба тврдњи је у проверавању такозваних предуслова или постуслова, односно стања пре или после извршавања неког метода.

Пример 10. Написати Јава програм, који помоћу тврдње, проверава да ли метод за сортирање низа заиста сортира низ (постуслов). □

```
package rs.math.oop.g11.p10.tvrdnjeSortiranjePostuslov;

public class SortiranjeSaPostuslovom {
       public static void main(String[] args) {
              int[] niz = { 20, 91, -6, 16, 0, 7, 51, 42, 3, 1 };
              sortiraj(niz);
              assert jeSortiran(niz) : "низ није сортиран";
              for (int e : niz)
                     System.out.printf("%d ", e);
              System.out.println();
       }
```

```
	private static boolean jeSortiran(int[] x) {
		for (int i = 0; i < x.length - 1; i++)
			if (x[i] > x[i + 1])
				return false;
		return true;
	}

	private static void sortiraj(int[] x) {
		int j, a;
		for (int i = 1; i < x.length; i++) {
			a = x[i];
			j = i;
			while (j > 0 && x[j - 1] > a) {
				x[j] = x[j - 1];
				j--;
			}
			x[j] = a;
		}
	}
}
```

Будући да је сортирање уметањем, у овом примеру, исправно реализовано, на излазу неће бити пријављена грешка, већ ће бити исписан низ сортираних бројева.

```
-6 0 1 3 7 16 20 42 51 91
```

Ако би позив метода за сортирање био коментарисан, добио би се следећи испис.

```
Exception in thread "main" java.lang.AssertionError: низ није сортиран
	at
rs.math.oop.g11.p10.tvrdnjeSortiranjePostuslov.SortiranjeSaPostuslovom.main(SortiranjeSaP
ostuslovom.java:9) ■
```

11.2.2. Препоруке за рад са тврдњама

Битно је уочити разлику између намене тврдњи и намене изузетака. Тврдње се користе како би програм, у току извршавања, указао на потпуно неприхватљиве околности. Стога је потребно да програмер поправи програм и избегне такве околности. Овим се не гарантује да програм ради оно што се очекује, али добром „покривеношћу“ тврдњама смањује се шанса да се деси нека неприхватљива околност у фази употребе. Са друге стране, изузеци се користе када је реч о грешкама које су зависне и од других фактора, а не само од програмерске логике. Примери оваквих грешака су недозвољена/неочекивана стања система датотека, неадекватни уноси од стране корисника и слично.

У фази употребе програма (продукционом окружењу) тврдње се обично онеспособљавају, односно, програм се извршава без задавања параметра -ea виртуелној машини.

11.3. Резиме

У овом поглављу објашњени су концепти изузетка и тврдње. Систем изузетака је настао из потребе за транспарентнијим и суптилнијим начином реаговања на проблематичне сценарије у току извршавања програма. Помоћу конструкције `try-catch-finally`

програмеру је омогућено да ухвати и обради грешку. Алтернативно, може се одложити обрада прослеђивањем изузетка методу-позиваоцу употребом кључне речи `throws`. Изузеци, као објекти, могу у себи садржати разноврсне корисне информације које указују на узрок проблема, а применом наслеђивања је могуће правити јасну диференцијацију њихове намене. Са друге стране, тврдње омогућавају најригорознији приступ провери коректности програма. Посебно су погодне код императивних језика и њихов циљ је да помогну програмеру у фази развоја програма, а не у фази његове употребе (продукционом окружењу).

11.4. Питања и задаци

1. Шта су изузеци и које су предности употребе изузетака?
2. Илустровати примером када је у програму потребно проблем обрадити употребом изузетака, а када је довољно исписати само информативну поруку кориснику.
3. Објаснити хијерархију изузетака у програмском језику Јава.
4. Објаснити и илустровати примером ситуацију када долази до изузетка типа `Error`, изузетка типа `RuntimeException`, а када до изузетка типа `Exception`.
5. Који су могући начини руковања изузецима? Илустровати примером.
6. Објаснити и илустровати примером ситуацију када се користи вишеструки `catch` блок. О чему посебно треба водити рачуна при употреби вишеструког `catch` блока?
7. Када је корисно корситити `finally` блок? Илустровати примером.
8. Написати Јава програм којим се са стандардног улаза уносе ниске све док се не унесе празна ниска. За сваку унесену ниску на екрану исписати подниску састављену од карактера који почињу на позицији 3 и дужине су 5, у којем су сва мала слова претворена у одговарајућа велика слова.
9. Шта се подразумева под прослеђивањем изузетка, а шта под избацивањем изузетка?
10. Решити проблем дат у примеру 8, с тим што се издвајање подниске и претварање слова одвија у посебној функцији чији су аргументи позиција индекса од којег почиње подниска и дужина подниске.
11. Када се користи кључна реч `throw`, а када кључна реч `throws`?
12. Истражити класу `ArithmeticException`, који методи се налазе у тој класи? Примером илустровати употребу неких метода класе `ArithmeticException`.
13. Написати Јава програм који демонстрира множење два дводимензионална низа целих бројева. Уколико димензије низова који се множе не одговарају правилима за множење, потребно је избацити нови тип изузетка.
14. Које су најважније препоруке за коришћење изузетака?
15. Шта су тврдње, како се реализују и када се користе?
16. Упоредити и примером илустровати ситуације када се за контролу грешака у току рада програма користе изузеци, а када тврдње.

12. Набројиви (енумерисани) тип

Набројиви тип (енг. enumerated type, enum) је објектни тип. Вредности које може узети променљива набројивог типа су познате већ у фази компилације програма. Постоје многобројне ситуације у којима су вредности података познате пре компајлирања и тада је погодно користити набројиви тип. На пример, постоји 7 дана у недељи и 12 месеци у години, па нема смисла креирање још неког редног броја дана или месеца у току извршавања програма.

Набројиви тип је подржан почев од верзије Јава 5. Међутим, потреба за набројивим типом је постојала и пре додавања подршке у Јави. До тада се ова потреба компензовала креирањем целобројних статичких поља. На пример, за дане у недељи и месеце у години користиле су се декларације попут следећих.

```
public static final int PONEDELJAK = 0;
public static final int UTORAK = 1;
...
public static final int JANUAR = 0;
public static final int FEBRUAR = 1;
...
```

Овај приступ не пружа заштиту од мешања различитих типова. На пример, следеће поређење је синтаксно коректно (и програм се преводи), али је његова семантика упитна.

```
int dan = JANUAR;
if(dan==PONEDELJAK){
  //...
}
```

Дакле, променљивој која представља дан у недељи се додељује вредност месеца у години. Набројиви типови спречавају овакве ситуације. Бекусовом нотацијом исказано, опис набројивог типа у најпростијем случају може бити дат на следећи начин:

```
<дефиниција набројивог типа> ::= enum <назив набројивог типа>{<дефиниције константи>}
<назив набројивог типа>  ::= <идентификатор>
<дефиниције константи> ::= {<дефиниција константи>,}<дефиниција константи>
<дефиниција константи> ::= <назив константе>[=<целобројни литерал>]
<назив константе>  ::= <идентификатор>
```

Декларација и иницјализација променљиве набројивог типа може бити представљена следећим металингвистичким формулама:

```
<декларација и иницијализација проемнљиве набројивог типа> ::=
  <назив набројивог типа> <назив променљиве>[=<назив набројивог типа>.<назив константе>]
```

Пример 1. Написати Јава програм који демонстрира дефинисање и употребу набројивог типа за дане у недељи. □

```
package rs.math.oop.g12.p01.enumiDaniUNedelji;

public enum DanUNedelji{
       PONEDELJAK, UTORAK, SREDA, CETVRTAK, PETAK, SUBOTA, NEDELJA;
}
```

Приликом дефинисања набројивог типа, наводе се и све његове могуће вредности. Током рада програма оне се не могу изменити, али не може се додати нова вредност или обрисати постојећа. Као што се може видети из претходно наведеног кода,

променљива набројивог типа показује на објекат који у себи садржи подразумеване атрибуте: име и редни број. Редни бројеви почињу од 0 и повећавају се за 1 код сваке наредне набројиве константе — на пример, `CETVRTAK` има додељен редни број 3.

```
package rs.math.oop.g12.p01.enumiDaniUNedelji;

public class TestirajDane {
    public static void main(String[] args) {
        DanUNedelji d1 = DanUNedelji.CETVRTAK;
        DanUNedelji d2 = DanUNedelji.UTORAK;
        DanUNedelji d3 = DanUNedelji.CETVRTAK;
        //DanUNedelji d4 = new DanUNedelji(); // није могуће
        System.out.println(d1.name());
        System.out.println(d1.ordinal());
        System.out.println(d1==d2);
        System.out.println(d1==d3);
    }
}
```

Као што се види из кода, креирање нових инстанци није дозвољено (наредба конструктора је коментарисана). Инстанцним променљивама је могуће доделити искључиво већ постојећу вредност набројивог типа.

Методима `name()` и `ordinal()` дохватају се придружени назив вредности набројивог типа и придружени редни број.

Поређење променљивих набројивог типа се мало разликује од осталих објектних типова. Наиме, ако се `d1` и `d3` упореде помоћу оператора `==` резултат је `true`, док би код других објектних типова за овај резултат требало употребити методе `equals()`. Међутим, ово је очекивани резултат, јер су `d1` и `d2` инстанцне променљиве које показују на исти објекат `CETVRTAK`. Стога не постоји потреба за методом `equals()`.

```
CETVRTAK
3
false
true
```

■

12.1. Набројиви типови и наредба switch

Набројиви типови се, поред употребе у оквиру `if-else` наредбе, могу користити и у оквиру наредбе `switch`.

Пример 2. Написати Јава програм за испис броја дана у одабраном месецу одабране године. Одабир месеца и године се остварује при покретању програма, навођењем као аргумената командне линије. Месец се описује називом записаним словима латиничног писма, док се година задаје као број. □

```
package rs.math.oop.g12.p02.enumiBrojDanaUMesecu;

public enum MesecKalendarski {
       JANUAR, FEBRUAR, MART, APRIL, MAJ, JUN,
       JUL, AVGUST, SEPTEMBAR, OKTOBAR, NOVEMBAR, DECEMBAR;
}
```

```
package rs.math.oop.g12.p02.enumiBrojDanaUMesecu;
```

```
public class TestirajMesece{
	public static void main(String[] args) {
		if (args.length != 2) {
			System.err.println("Употреба: java <назив месеца> <година>");
			System.exit(-1);
		}
		try {
			MesecKalendarski mesec =
					MesecKalendarski.valueOf(args[0].toUpperCase());
			int godina = Integer.valueOf(args[1]);
			int brojDana = 0;
			switch (mesec) {
			case APRIL:case JUN:case SEPTEMBAR:case NOVEMBAR:
				brojDana = 30;
				break;
			case JANUAR:case MART: case MAJ: case JUL:
				case AVGUST:case OKTOBAR: case DECEMBAR:
					brojDana=31;
					break;
			case FEBRUAR:
				if ((godina % 4 == 0 && godina % 100 != 0)
						|| (godina % 400 == 0))
					brojDana = 29;
				else
					brojDana = 28;
				break;
			}
			System.out.printf("Број дана у месецу %s године %d је %d.",
					mesec, godina, brojDana);
		} catch (IllegalArgumentException ex) {
			System.err.printf("Грешка при парсирању месеца са називом %s.",
					args[1]);
			System.err.println(ex.getMessage());
		}
	}
}
```

(Скоро идентичан задатак, само без употребе набројивих типова, решен је у оквиру примера 5 из секције 5.5.3. Тамо је појашњено и разматрање у вези броја дана у фебруару, у зависности од тога да ли је година проста или преступна.)

Након што се учита кориснички унос са називом месеца, на основу тог уноса се приступа одговарајућој набројивој вредности помоћу статичког метода `MesecKalendarski.valueOf()`. У случају да је дат текст који не одговара називу неке од доступних набројивих вредности, избацује се изузетак типа `IllegalArgumentException`. Ако је све у реду, програм наставља даље и помоћу наредбе `switch` одређује се број дана за сваки месец. Резултат рада програма за унете аргументе „februar 2000“ је:

```
Број дана у месецу FEBRUAR године 2000 је 29.
```

Ако би унос био неодговарајући, тј. ако би било унето нешто што не постоји у списку назива набројивих вредности, дошло би до изузетка.

```
Грешка при парсирању месеца са називом 2002.No enum constant
rs.math.oop.g11.p02.enumiBrojDanaUMesecu.MesecKalendarski.TEST■
```

12.2. Обогаћивање набројивих типова, конструктори и методи

Набројиви типови у Јави могу носити и додатне информације, поред раније споменутих. Такође је могуће у набројивом типу дефинисати нови метод или превазићи неки постојећи. Исто важи и за конструкторе.
У том случају, синтакса за енумерисани тип постаје осетно компликованија:

```
<дефиниција набројивог типа> ::= enum <назив набројивог типа>
   {<дефиниције константи>[<проширење>]}
<назив набројивог типа>  ::= <идентификатор>
<дефиниције константи> ::= {<дефиниција константи>,}<дефиниција константи>
<дефиниција константи> ::= <назив константе>[=<целобројни литерал>]
<назив константе>  ::= <идентификатор>
<проширење> ::= {{(<дефиниција поља>|<дефиниција метода>|<конструктор>)}}
<дефиниција поља> ::=  [<модификатор видљивости> ][static ][final ]
                              <декларација и иницијализација променљивих>
<дефиниција метода> ::= <заглавље метода><тело метода>
<заглавље метода> ::= [<модификатор видљивости> ][static ][final |abstract ]
                               <повратни тип><назив метода>(<параметри>)
<повратни тип> ::= <тип>|void
<назив метода>::= <идентификатор>
<параметри>::= [<параметар>]{,<параметар>}
<параметар> ::= <тип параметра> <назив параметра>
<тип параметра> ::= [final ]<тип>
<назив параметра>::= <идентификатор>
<тело метода> ::= { <наредбе тела матода> }
<наредбе блока> ::= {<наредба тела метода>}
<наредба тела метода> ::= <наредба>|<тип класе>
<иницијализациони блок> ::= [static ]<блок>
<конструктор> ::= <име класе>(<параметри>)<блок>
<име класе> ::= <идентификатор>
```

Пример 3. Написати Јава програм који ради са набројивим типом за представљање планета Сунчевог система. Поред назива и редног броја планете, потребно је предвидети и додатне информације о маси и пречнику. На стандардни излаз потом исписати информације за сваку планету. Додатно, дефинисати метод који рачуна тежину тела на површини сваке планете. Као аргумент командне линије се задаје маса тела.□

```
package rs.math.oop.g12.p03.enumiPlaneteSuncevogSistema;

public enum PlanetaSuncevogSistema {
    MERKUR (3.303e+23, 2.4397e6),
    VENERA   (4.869e+24, 6.0518e6),
    ZEMLJA   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URAN  (8.686e+25, 2.5559e7),
    NEPTUN (1.024e+26, 2.4746e7);

    private final double masa;   // kg
    private final double precnik; // m
```

```
    PlanetaSuncevogSistema(double masa, double precnik) {
        this.masa = masa;
        this.precnik = precnik;
    }

    double masa() { return masa; }
    double precnik() { return precnik; }

    // универзална гравитациона константа (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double gravitacijaNaPovrsini() {
        return G * masa / (precnik * precnik);
    }

    double tezinaTelaNaPovrsini(double masaTela) {
        return masaTela * gravitacijaNaPovrsini();
    }

    @Override
    public String toString() {
       return String.format("%d %s %g %g",ordinal(), name(), masa, precnik);
    }
}
```

Конструктор, који је дефинисан унутар набројивог типа, видљив је само унутар датотеке у којој је набројиви тип дефинисан. Овај конструктор није могуће позвати експлицитно са неког другог места, осим тамо где се наводи списак набројивих вредности. Набројивом типу `PlanetaSuncevogSistema` додељени су и нови методи попут `gravitacijaNaPovrsini()` и `tezinaTelaNaPovrsini()`, али је и превазиђен постојећи метод `toString()`.

```
package rs.math.oop.g12.p03.enumiPlaneteSuncevogSistema;

public class TestirajPlanete{
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Употреба: java <маса тела (kg)>");
            System.exit(-1);
        }
        double masaTela = Double.parseDouble(args[0]);
        // метода values() враћа списак свих вредности за дати набројиви тип
        System.out.println(
              "Карактеристике планета: редни број, име, маса и пречник:");
        for (PlanetaSuncevogSistema p : PlanetaSuncevogSistema.values())
              System.out.println(p);
        System.out.printf(System.lineSeparator()
              +"Тежина тела масе %.2f на различитим планетама су%n",masaTela);
        for (PlanetaSuncevogSistema p : PlanetaSuncevogSistema.values())
           System.out.printf("%s %.2f N%n",
                             p.name(), p.tezinaTelaNaPovrsini(masaTela));
    }
}
```

За унос од 88 килограма, програм формира следећи испис.

```
Карактеристике планета: редни број, име, маса и пречник:
0 MERKUR 3.30300e+23 2.43970e+06
1 VENERA 4.86900e+24 6.05180e+06
2 ZEMLJA 5.97600e+24 6.37814e+06
3 MARS 6.42100e+23 3.39720e+06
4 JUPITER 1.90000e+27 7.14920e+07
5 SATURN 5.68800e+26 6.02680e+07
6 URAN 8.68600e+25 2.55590e+07
7 NEPTUN 1.02400e+26 2.47460e+07

Тежина тела масе 88.00 на различитим планетама су
MERKUR 325.87 N
VENERA 780.68 N
ZEMLJA 862.63 N
MARS 326.71 N
JUPITER 2182.94 N
SATURN 919.58 N
URAN 780.79 N
NEPTUN 981.96 N■
```

У неким ситуацијама, корисно је набројиви тип обогатити и методом који ће бити реализован различито за различите вредности, а не исто као за досад уведене методе попут `masa()`, `precnik()`, `tezinaTelaNaPovrsini()` и `gravitacijaNaPovrsini()`. Ово је могуће декларисањем апстрактног метода у оквиру набројивог типа и његовом реализацијом приликом навођења сваке од вредности набројивог типа. Следећи пример демонстрира овај концепт.

Пример 4. Написати Јава програм који демонстрира употребу набројивог типа за представљање основних аритметичких операција: сабирање, одузимање, множење и дељење.□

```
package rs.math.oop.g12.p04.enumiAritmetickeOperacije;

public enum AritmetickaOperacija {
	PLUS("+") {
		public double izracunaj(double x, double y) {
			return x + y;
		}
	},
	MINUS("-") {
		public double izracunaj(double x, double y) {
			return x - y;
		}
	},
	PUTA("*") {
		public double izracunaj(double x, double y) {
			return x * y;
		}
	},
	PODELJENO("/") {
		public double izracunaj(double x, double y) {
			return x / y;
		}
	};

	private final String oznaka;
```

```
        AritmetickaOperacija(String oznaka) {
                this.oznaka = oznaka;
        }

        @Override
        public String toString() {
                return oznaka;
        }

        public abstract double izracunaj(double x, double y);

        public static void main(String[] args) {
                double x = Double.parseDouble(args[0]);
                double y = Double.parseDouble(args[1]);
                for (AritmetickaOperacija op : AritmetickaOperacija.values())
                        System.out.printf("%f %s %f = %f%n", x, op, y,
                           op.izracunaj(x, y));
        }
}
```

Дакле, могуће је декларисати апстрактни метод унутар набројивог типа, али га је и обавезно одмах реализовати приликом навођења вредности набројивог типа – ово доста подсећа на концепт анонимних класа, обрађених у секцији 10.2.2.
(У овом примеру је и метод `main()` уписан у датотеку у којој је дефинисан и набројиви тип.)
Следи резултат извршавања овог примера за аргументне командне линије 23 и 10:

```
23.000000 + 10.000000 = 33.000000
23.000000 - 10.000000 = 13.000000
23.000000 * 10.000000 = 230.000000
23.000000 / 10.000000 = 2.300000■
```

12.3. Реализација набројивог типа помоћу класе

Набројиви типови су реализовани као класе, али је због честе употребе њихова реализација сакривена од програмера. У наредном примеру ће бити делимично демонстрирана њихова реализација уз употребу класа (потпуна реализација захтева коришћење рефлексије, која није обрађена у овој књизи).
Пример 5. Написати Јава програм у којем се реализује набројиви тип `DanUNedelji` употребом класе. □

```
package rs.math.oop.g12.p05.enumiDaniUNedeljiKlasa;

public final class DanUNedeljiKlasa {
    public static final DanUNedeljiKlasa PONEDELJAK = new DanUNedeljiKlasa("PONEDELJAK");
    public static final DanUNedeljiKlasa UTORAK = new DanUNedeljiKlasa("UTORAK");
    public static final DanUNedeljiKlasa SREDA = new DanUNedeljiKlasa("SREDA");
    public static final DanUNedeljiKlasa CETVRTAK = new DanUNedeljiKlasa("CETVRTAK");
    public static final DanUNedeljiKlasa PETAK = new DanUNedeljiKlasa("PETAK");
    public static final DanUNedeljiKlasa SUBOTA = new DanUNedeljiKlasa("SUBOTA");
    public static final DanUNedeljiKlasa NEDELJA = new DanUNedeljiKlasa("NEDELJA");

    private final String naziv;
    private final int redniBroj;
```

```
    private static int brojInstanci = 0;

    private DanUNedeljiKlasa(String naziv) {
        this.naziv = naziv;
        this.redniBroj = brojInstanci;
        brojInstanci++;
    }

    public int ordinal() {
        return redniBroj;
    }

    public String name() {
        return naziv;
    }

    public static DanUNedeljiKlasa[] values() {
        return new DanUNedeljiKlasa[] { PONEDELJAK, UTORAK, SREDA, CETVRTAK, PETAK,
SUBOTA, NEDELJA };
    }

    @Override
    public String toString() {
        return naziv;
    }

    public static void main(String[] args) {
        DanUNedeljiKlasa d1 = DanUNedeljiKlasa.CETVRTAK;
        DanUNedeljiKlasa d2 = DanUNedeljiKlasa.UTORAK;
        DanUNedeljiKlasa d3 = DanUNedeljiKlasa.CETVRTAK;
        System.out.println(d1.name());
        System.out.println(d1.ordinal());
        System.out.println(d1 == d2);
        System.out.println(d1 == d3);
        for (DanUNedeljiKlasa d : DanUNedeljiKlasa.values())
               System.out.println(d);
    }
}
```

Понашање овако реализованог набројивог типа је слично оном представљеном у примеру 1:

- додељивање вредности променљивама је исто;
- поређење употребом оператора `==` такође;
- редни број се аутоматски додељује.

Реализовани су методи `values()` и `toString()` за дохватање свих набројивих вредности и за текстуални приказ сваке појединачне вредности.

Следи резултат извршавања програма.

```
CETVRTAK
3
false
true
PONEDELJAK
UTORAK
SREDA
CETVRTAK
```

```
PETAK
SUBOTA
NEDELJA■
```

12.4. Резиме

У овом поглављу уведен је набројив тип података и описане су типичне ситуације у којима се он користи. Постоје још неки концепти, попут набројивих скупова, комбиновања вишеструких набројивих типова итд. који су повезани са набројивим типом. Да би се препознала ситуација у којој треба употребити набројиви тип, потребно је програмерско искуство. Основна смерница је да га треба користити када је скуп вредности познат већ у фази компилације и не очекује се проширивање скупа могућих вредности.

12.5. Питања и задаци

1. Упоредити руковање набројивим типовима пре и после увођења верзије Јава 5.
2. Написати Јава програм који демонстрира дефинисање и употребу набројивог типа за месеце у години.
3. Примером илустровати употребу метода `valueOf()`?
4. Написати Јава програм који демонстрира употребу набројивог типа за представљање основних логичких операција: негација, конјункција и дисјункција.
5. Написати Јава програм који реализује набројиви тип `MesecUGodini` употребом класе.
6. Истражити да ли постоје још неки уграђени методи у оквиру набројивог типа (`enum`) и примером илустровати употребу.

13. Генерички тип

Генерички тип омогућава употребу промењљиве на месту где се користи назив типа података (класе или интерфејса). Као што логичка променљива узима вредност из скупа `{true, false}` тако генерички тип узима вредност из скупа класа или интерфејса.
Ако имамо математичку функцију f(3) која за аргумент 3 даје неки резултат и креирамо функцију f(p) која за различите вредности параметра p (укључујући и 3) даје резултат, креирали смо оштију (параметризовану или генеричку) функцију. Управо на овај начин, коришћењем параметара за различите типове података, уводе се генерички типови почев од верзије Јава 5. Због присуства параметара понекад се називају и параметризовани.
Један од главних мотива за увођење генеричких типова је развој генеричких алгоритама који се могу примењивати на различите типове података. Као добар пример за генеричку функцију може да послужи C-функција `qsort()` помоћу које се реализује уопштен алгоритам за сортирање. Ова функција као аругменте прихвата низ који се сортира, број његових елемената, величину елемента и показивач на функцију поређења. Последња два аргумента омогућавају прилагођавање уопштеног алгоритма за сортирање конкретном типу података.

13.1. Сирови тип

У досадашњем тексту користили су се искључиво сирови типови. То су на пример: `Integer`, `Float`, `LocalDateTime`, `String` итд. Њихова употреба је у већини ситуација довољна, посебно за писање програма који имају конкретну намену. Са друге стране, ако је потребно писати програмске библиотеке са услужним методима (методима које ће користити други програми или библиотеке), тј. програме опште намене, сирови типови могу бити неадекватни. Програми опште намене писани су у Јави и пре појаве генеричких типова, коришћењем класе `Object`, али су били подложни грешкама, као што се види из следећег примера.
Пример 1. Написати Јава програм за рад са класом која представља кутију. У кутију је могуће убацити произвољни објекат. Дефинисати методе за убацивање и извлачење објекта из кутије и тестирати њихов рад. □

```
package rs.math.oop.g13.p01.genericiKutijaSirova;

public class KutijaSirova {
	private Object vrednost;

	void postaviVrednost(Object vrednost) {
		this.vrednost=vrednost;
	}

	Object uzmiVrednost() {
		return this.vrednost;
	}

	public static void main(String[] args) {
		KutijaSirova kutija1 = new KutijaSirova();
		kutija1.postaviVrednost("Текст");
		String tekst1 = (String) kutija1.uzmiVrednost();
```

```
                System.out.println(tekst1);
                Integer broj1 = (Integer) kutija1.uzmiVrednost(); // изузетак
        }
}
```

У кутију се може убацити објекат било које класе. Објекат се помоћу метода `postaviVrednost()` имплицитно конвертује у најопштији сирови тип `Object`. Будући да се на овај начин губи тачна информација о типу поља `vrednost` (осим ако је убачен баш објекат класе `Object`), метод за узимање вредности `uzmiVrednost()` враћа такође објекат типа `Object`. Дакле, убацивање објекта је једноставно, међутим, проблем настаје приликом узимања вредности из кутије. Наиме, програмер је дужан да зна прави тип објекта у кутији и да изврши експлицитну конверзију у складу са тим. Из примера се види да компајлер дозвољева експлицитну конверзију садржаја кутије у нетачан објектни тип `Integer`. Проблем са конверзијом у погрешан тип манифестоваће се тек у фази извршавања и биће избачен изузетак класе `ClassCastException`.

```
Текст
Exception in thread "main" java.lang.ClassCastException: class java.lang.String cannot be
cast to class java.lang.Integer (java.lang.String and java.lang.Integer are in module
java.base of loader 'bootstrap')
        at
rs.math.oop.g12.p01.genericiKutijaSirova.KutijaSirova.main(KutijaSirova.java:26)■
```

13.2. Појам, дефинисање и предности генеричког типа

Генерички типови се понашају слично као формални параметри при дефинисању метода. Програмирање коришћењем генеричких типова има следеће предности:

1. строжија контрола типова приликом превођења Јава програма;
2. елиминација експлицитне конверзије типова (кастовања);
3. могућност имплементације генеричких алгоритама.

Генеричка класа се дефинише на следећи начин:

```
class imeKlase<T1,T2,...,Tn>{ ... }
```

Непосредно иза имена класе следи део у стреличастим заградама са параметрима `T1`, `T2`, ..., и `Tn` који представљају типове. По конвенцији, параметри су означени једним великим словом. Најчешће ознаке параметара су:

- E – елеменат (енг. element)
- K – кључ (енг. key)
- N – број (енг. number)
- T – тип (енг. type)
- V – вредност (енг. value)
- S,U,W итд. – други, трећи, четврти итд.

Пример 2. Написати Јава програм за рад са класом која представља кутију, попут оне у примеру 1. За разлику од примера 1, класу је потребно реализовати као генеричку.□

```
package rs.math.oop.g13.p02.genericiKutijaGenericka;

public class KutijaGenericka<T> {
        private T vrednost;

        public void postaviVrednost(T vrednost) {
```

```
                this.vrednost=vrednost;
        }

        public T uzmiVrednost() {
                return this.vrednost;
        }

        public static void main(String[] args) {
                KutijaGenericka<String> kutija1 = new KutijaGenericka<String>();
                kutija1.postaviVrednost("Текст");
                String tekst1 = kutija1.uzmiVrednost();
                System.out.println(tekst1);
                // kutija1.postaviVrednost(45);
        }
}
```

Класа генеричка кутија је параметризована коришћењем параметра `T`. Параметар има опсег важења у целом телу класе. У наведеном примеру постоје 3 локације у којима се користе параметри:

1. код декларације типа променљиве `vrednost`;
2. код типа аргумента метода `postaviVrednost();`
3. код повратне вредности метода `uzmiVrednost()`.

За разлику од класе `KutijaSirova`, објекти класе `KutijaGenericka`, преко конструктора, подешавају тип података за поље `vrednost`. Ово функционише тако што се у генеричкој класи `KutijaGenericka<T>`, `T` замењује конкретном вредношћу (на пример `String`). На тај начин и цела генеричка класа добија своју конкретизацију (на пример `KutijaGenericka<String>`). Овим се елиминише потреба за експлицитном конверзијом податка који се узима из кутије, али се, у исто време, спречава програмера да за вредност у кутији постави нешто што није `String`.

```
Текст■
```

Пример 3. Написати Јава програм за рад са генеричком класом која представља генерички уређени пар вредности произвољних типова. Поред конструктора, који прихвата обе вредности, потребно је реализовати и методе за дохватање првог и другог члана уређеног пара. Такође је потребно редефинисати метод `toString()`.□

```
package rs.math.oop.g13.p03.genericiUredjeniPar;

public class UredjeniPar<T, S>{
        private T vrednost1;
        private S vrednost2;

        public UredjeniPar(T vrednost1, S vrednost2) {
                this.vrednost1 = vrednost1;
                this.vrednost2 = vrednost2;
        }

        public T getVrednost1() {
                return vrednost1;
        }

        public S getVrednost2() {
                return vrednost2;
        }
```

```
        @Override
        public String toString() {
                return "("+vrednost1+", "+vrednost2+")";
        }

        public static void main(String[] args) {
                UredjeniPar<Integer, Integer> par1 =
                                new UredjeniPar<Integer, Integer>(10, 20);
                UredjeniPar<Integer, String> par2 = new UredjeniPar<>(30,
                                                                "Пример текст");
                // UredjeniPar<Integer, Integer> par3 = new UredjeniPar<>(30, 14.0);
                System.out.println(par1);
                System.out.println(par2);
        }
}
```

Навођењем два различита параметра `T` и `S` омогућено је креирање уређених парова у којима први и други члан не морају имати исти тип. Приликом креирања другог уређеног пара `par2`, примећује се да у позиву конструктора изостају подешавања вредности параметара. Јава компајлер ово допушта јер је, на основу декларације објектне променљиве, јасно да то морају бити цео број и текст. Креирање уређеног пара `par3` није дозвољено јер се тип другог аргумента, прослеђеног конструктору, не поклапа са декларисаним типом.

```
(10, 20)
(30, Пример текст)■
```

13.2.1. Генерички интерфејси и њихова имплементација

Интерфејси се понашају као класе па је могуће креирати генеричке интерфејсе, а такође, помоћу њих декларисати генеричке методе.

```
public interface Interfejs <T>{
    void f(T arg);
    T g();
}
```

Приликом њихове имплементације од стране класа генеричност се може задржати или изгубити (прецизирањем класе при дефинисању).

```
class ObicnaKlasa implements Interfejs<String>{ ... }
class GenerickaKlasa<T> implements Interfejs<T>{ ... }
```

Пример 4. У примеру 11 из секције 9.2.5 дефинисан је интерфејс `StekNiski`, након чега су реализоване његове две различите имплементације: `StekNiskiPrekoNiza` и `StekNiskiPrekoListe`. Прилагодити те дефиниције тако да се може оперисати са произвољним типовима података, а не само са нискама. Након тога, демонстрирати употребу обе имплементације додавањем елемената на стек и накнадним сабирањем свих елемената стека.□

```
package rs.math.oop.g13.p04.generickiInterfejsStek;

public interface Stek<T> {
   void dodaj(T elem);
   T ukloni();
```

```
    int brojElemenata();
}
```

Најпре је дефинисан генерички стек интерфејс са методима карактеристичним за стек структуру: додавање на врх стека, уклањање са врха и враћање броја елемената.

```
package rs.math.oop.g13.p04.generickiInterfejsStek;

import java.util.Arrays;

public class SamorastuciNiz<T> {
    private T[] elementi = (T[]) new Object[8];
    private int brojElemenata = 0;

    private void obezbediKapacitet(int noviKapacitet) {
        if (noviKapacitet <= trenutniKapacitet())
                return;
        T[] pomocni = elementi;
        elementi = Arrays.copyOf(pomocni, noviKapacitet);
    }

    public void postaviNa(int indeks, T vrednost) {
        if (indeks >= trenutniKapacitet()) {
                int noviKapacitet = Integer.max(indeks, 2 * elementi.length);
                obezbediKapacitet(noviKapacitet);
        }
        elementi[indeks] = vrednost;
        if (indeks + 1 > brojElemenata)
                brojElemenata = indeks + 1;
    }

    public T uzmiSa(int indeks) {
        if (indeks < 0) {
                System.err.println("Грешка: индекс је негативан!");
                return null;
        }
        if (indeks >= elementi.length) {
                System.err.println("Грешка: индекс је већи од величине низа!");
                return null;
        }
        return elementi[indeks];
    }

    public int brojElemenata() {
        return brojElemenata;
    }

    public int trenutniKapacitet() {
        return elementi.length;
    }
}
```

Класа `SamorastuciNiz<T>` је сада генеричка, за разлику од претходних примера у којима се односила на тип `String`. Изузев ове разлике, све остало је исто као и раније.

```
package rs.math.oop.g13.p04.generickiInterfejsStek;

public class StekPrekoNiza<T> implements Stek<T> {
```

```
    private SamorastuciNiz<T> elementi;
    private int vrhSteka;

    {
        elementi = new SamorastuciNiz();
        vrhSteka = -1;
    }

    @Override
    public void dodaj(T elem) {
        elementi.postaviNa(++vrhSteka, elem);
    }

    @Override
    public T ukloni() {
        if (vrhSteka == -1) {
                System.err.println("Грешка при уклањању: стек је празан!");
                return null;
        }
        return elementi.uzmiSa(vrhSteka--);
    }

    @Override
    public int brojElemenata() {
        return (vrhSteka + 1);
    }
}
```

Слично је и са класом `StekPrekoNiza<T>`. Ова класа је у свим аспектима иста као и раније дефинисана класа `StekNiskiPrekoNiza`, једино је сада присутна генеричност.

```
package rs.math.oop.g13.p04.generickiInterfejsStek;

public class PovezanaLista<T> {

    public class Cvor<E> {
        private E sadrzaj;
        private Cvor<E> sledeci;

        private Cvor(E elem) {
                sadrzaj = elem;
                sledeci = null;
        }

        public E uzmiSadrzaj() {
                return sadrzaj;
        }

        public void postaviSadrzaj(E sadrzaj) {
                this.sadrzaj = sadrzaj;
        }

        public Cvor<E> uzmiSledeci() {
                return sledeci;
        }

        public void postaviSledeci(Cvor<E> sledeci) {
```

```
                this.sledeci = sledeci;
        }

        @Override
        public String toString() {
                return "Листа: " + sadrzaj;
        }

    }

    private Cvor<T> pocetak = null;
    private Cvor<T> kraj = null;
    private Cvor<T> tekuci = null;

    public PovezanaLista() { }

    public PovezanaLista(T elem) {
        if (elem != null)
                tekuci = kraj = pocetak = new Cvor<T>(elem);
    }

    public PovezanaLista(T[] elementi) {
        if (elementi == null)
                return;
        for (int i = 0; i < elementi.length; i++)
                dodajNaKraj(elementi[i]);
        tekuci = pocetak;
    }

    public void dodajNaKraj(T elem) {
        Cvor<T> noviKraj = new Cvor(elem);
        if (pocetak == null)
                pocetak = kraj = noviKraj;
        else {
                kraj.postaviSledeci(noviKraj);
                kraj = noviKraj;
        }
    }

    public T ukloniSaKraja() {
        if (kraj == null)
                return null;
        if (pocetak == kraj) {
                Cvor<T> jedini = kraj;
                pocetak = kraj = null;
                return jedini.uzmiSadrzaj();
        }
        Cvor<T> poslednji = kraj;
        Cvor<T> pretposlednji = pocetak;
        while (pretposlednji.uzmiSledeci() != poslednji)
                pretposlednji = pretposlednji.uzmiSledeci();
        pretposlednji.postaviSledeci(null);
        kraj = pretposlednji;
        return poslednji.uzmiSadrzaj();
    }
```

```
    public void dodajNaPocetak(T elem) {
        Cvor<T> noviPocetak = new Cvor(elem);
        if (kraj == null)
                pocetak = kraj = noviPocetak;
        else {
                noviPocetak.postaviSledeci(pocetak);
                pocetak = noviPocetak;
        }
    }

    public T ukloniSaPocetka() {
        if (pocetak == null)
                return null;
        if (pocetak == kraj) {
                Cvor<T> jedini = kraj;
                pocetak = kraj = null;
                return jedini.uzmiSadrzaj();
        }
        Cvor<T> prvi = pocetak;
        pocetak = prvi.uzmiSledeci();
        return prvi.uzmiSadrzaj();
    }

    public boolean stigaoDoKraja() {
        return (tekuci == kraj) || (tekuci == null);
    }

    public T uzmiPrvi() {
        tekuci = pocetak;
        return (tekuci == null) ? null : tekuci.uzmiSadrzaj();
    }

    public T uzmiSledeci() {
        if (!stigaoDoKraja())
                tekuci = tekuci.uzmiSledeci();
        return (tekuci == null) ? null : tekuci.uzmiSadrzaj();
    }

    public int brojCvorova() {
        int n = 1;
        Cvor tek = pocetak;
        if (tek == null)
                return 0;
        while (tek != kraj) {
                tek = tek.uzmiSledeci();
                n++;
        }
        return n;
    }
}
```

Класа `PovezanaLista<T>` је сада генеричка, за разлику од претходних примера у којима се односила на тип `String`. Изузев ове разлике, све остало је исто као и раније.

```
package rs.math.oop.g13.p04.generickiInterfejsStek;

public class StekPrekoListe<T> implements Stek<T> {
```

```
    private PovezanaLista<T> elementi = new PovezanaLista();

    @Override
    public void dodaj(T elem) {
        elementi.dodajNaPocetak(elem);
    }

    @Override
    public T ukloni() {
        T elem = elementi.ukloniSaPocetka();
        if (elem == null) {
               System.err.println("Грешка при уклањању: стек је празан!");
               return null;
        }
        return elem;
    }

    @Override
    public int brojElemenata() {
        return elementi.brojCvorova();
    }
}
```

Класа је у свим аспектима иста као и раније дефинисана класа `StekNiskiPrekoListi`, једино је сада присутна генеричност.

```
package rs.math.oop.g13.p04.generickiInterfejsStek;

import java.util.Scanner;

public class CitanjeSumiranje {
    private static Stek<Double> kreiraj(Scanner sc) {
        Stek stek = null;
        System.out.print("Унеси л (за листу) или н (за низ): ");
        char ulaz = sc.nextLine().trim().toLowerCase().charAt(0);
        switch (ulaz) {
        case 'л':
        case 'l':
               stek = new StekPrekoListe();
               break;
        case 'н':
        case 'n':
               stek = new StekPrekoNiza();
        }
        return stek;
    }

    private static void napuni(Scanner sc, Stek<Double> stek) {
        while (sc.hasNext()) {
               String rec = sc.next();
               if (rec.equals("KPAJ"))
                      break;
               try {
                      double d = Double.parseDouble(rec);
                      stek.dodaj(d);
               } catch (NumberFormatException exp) {
                      System.err.println("Некоректан формат: " + rec);
```

```
                }
            }
        }

        private static double prikaziSaberiIsprazni(Stek<Double> stek) {
            double rezultat = 0;
            while (stek.brojElemenata() > 0) {
                    double d = stek.ukloni();
                    rezultat += d;
                    System.out.printf("%10.6f", d);
            }
            System.out.println();
            return rezultat;
        }

        public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            Stek<Double> stek = kreiraj(sc);
            if (stek == null)
                    return;
            System.out.println("Унеси бројеве (или реч КРАЈ ћирилицом за крај уноса):");
            napuni(sc, stek);
            sc.close();
            System.out.println("Број прочитаних бројева: " + stek.brojElemenata());
            Double suma = prikaziSaberiIsprazni(stek);
            System.out.printf("Сума прочитаних бројева: %10.6f", suma);
        }
}
```

У наставку су приказана два сценарија. У првом се користи низовна имплементација стека (унет је карактер ‘н’), док се у другом користи стек реализован преко повезане листе (унет је карактер ‘л’). Након тога се, у оба случаја, уноси секвенца реалних бројева са конзоле. Бројеви се редом додају на врх стека, након чега се врши њихово уклањање, приказивање на конзоли и додавање суми бројева. Потом се на конзоли исписује добијена сума.

```
Унеси л (за листу) или н (за низ): н
Унеси бројеве (или реч КРАЈ ћирилицом за крај уноса):
1
-24
35.6
89.11
КРАЈ
Број прочитаних бројева: 4
 89.110000 35.600000-24.000000  1.000000
Сума прочитаних бројева: 101.710000

Унеси л (за листу) или н (за низ): л
Унеси бројеве (или реч КРАЈ ћирилицом за крај уноса):
1
-24
35.6
89.11
КРАЈ
Број прочитаних бројева: 4
 89.110000 35.600000-24.000000  1.000000
```

```
Сума прочитаних бројева: 101.710000■
```

13.3. Самостални генерички метод

До сада уведени генерички методи „генеричност“ су стицали на основу припадности генеричкој класи или генеричком интерфејсу. Међутим, опсег генеричког параметра не мора обухватати читаву класу или интерфејс. У Јави је могућа и локалнија (самосталнија) употреба генеричких параметара, тј. употреба само у оквиру појединачних метода. Да би се разликовала ова два типа генеричких метода, они код којих генерички параметар није потекао из класе или интерфејса називају се самостални генерички методи.

Пример 5. Написати Јава програм који реализује статички самостални генерички метод за линеарну претрагу генеричког низа и тестирати његове могућности.□

```
package rs.math.oop.g13.p05.genericiPretragaNiza;

public class PretragaNiza {
	public static <T> int pretrazi(T[] niz, T element) {
		for(int i=0; i<niz.length; i++)
			if(niz[i].equals(element))
				return i;
		return -1;
	}

	public static void main(String[] args) {
		Integer[] nizCelih = new Integer[] {2,43,22,11,243,253,64};
		int element1 = 34;
		int element2 = 243;
		System.out.printf("Позиција елемента %d је %d.%n", element1,
				pretrazi(nizCelih, element1));
		System.out.printf("Позиција елемента %d је %d.%n", element2,
				pretrazi(nizCelih, element2));
	}
}
```

Овде је параметар `T` написан непосредно испред повратног типа метода и он нема утицај на остале чланице класе `PretragaNiza`. Алгоритам линеарне претраге, за разлику од алгоритма бинарне претраге, не захтева да низ буде уређен. Једини захтев је да класа која ће конкретизовати параметар `T` буде упоредива на једнакост. Ово својство је сигурно подржано код свих класа у Јави, јер је метод `equals()` дефинисан у класи `Object`.

Следи испис добијен извршавањем програма.

```
Позиција елемента 34 је -1.
Позиција елемента 243 је 4.■
```

13.4. Ограничења за типове

У неким ситуацијама је потребно да генерички параметар испуњава додатне критеријуме како би могао да се користи за реализацију генеричког алгоритма. То значи да параметар не може увек представљати произвољну класу као што је то био случај у досадашњим примерима. На пример, за потребе реализације алгоритма бинарне

претраге, сортирања, тражења минимума, максимума итд. типови морају бити упоредиви. Следећа нотација ограничава параметарски тип `T` са „горње стране“.

```
<T extends TipKojiOgranicava>
```

То значи да тип који ће конкретизовати параметар `T` мора бити или једнак типу `TipKojiOgranicava` или испод њега у хијерархији типова. Ово има различиту интерпретацију у зависности од тога да ли су `T` и `TipKojiOgranicava` класе или интерфејси.

- У случају да су оба класе, онда `T` може бити `TipKojiOgranicava` или њена поткласа.
- Ако је `T` класа, а `TipKojiOgranicava` интерфејс, онда `T` мора имплементирати интерфејс `TipKojiOgranicava`.
- Ако су оба параметра интерфејси, онда су интерфејси исти или интерфејс `T` проширује интерфејс `TipKojiOgranicava`.

Иако би боље решење било да је за ограничавање уведена нова кључна реч (главни кандидат је била реч `sub`), дизајнери Јаве су одлучили да је једноставније да искористе већ постојећу кључну реч `extends`.

Тип може истовремено бити ограничен највише једном класом и произвољним бројем интерфејса. У том случају се ограничења набрајају на следећи начин.

```
<T extends TipKojiOgranicava1 & TipKojiOgranicava2 & ...>
```

Тип `T`, у случају вишеструког ограничавања, мора бити “испод” свих наведених ограничавајућих типова, тј. реч је о конјункцији ограничења.

Пример 6. Написати Јава програм који реализује статички самостални генерички метод за тражење минималног елемента у низу. Након тога, применити метод на низ објеката типа `String` и низ објеката раније уведеног генеричког типа уређени пар. Потребно је „дорадити“ класу за уређени пар тако да имплементира генерички интерфејс `Comparable`. Поређење се врши на основу прве координате, а потом на основу друге ако су прве координате једнаке. □

```
package rs.math.oop.g13.p06.genericiMinimalniElementNiza;

public class UredjeniParUporediv<S extends Comparable<S>, T extends Comparable<T>>
        implements Comparable<UredjeniParUporediv<S, T>>{
        private T vrednost1;
        private S vrednost2;

        public UredjeniParUporediv(T vrednost1, S vrednost2) {
                this.vrednost1 = vrednost1;
                this.vrednost2 = vrednost2;
        }

        public T getVrednost1() {
                return vrednost1;
        }

        public S getVrednost2() {
                return vrednost2;
        }

        @Override
        public String toString() {
```

```
                return "("+vrednost1+", "+vrednost2+")";
        }

        @Override
        public int compareTo(UredjeniParUporediv<S, T> o) {
                int uredjenjeS = this.vrednost1.compareTo(o.vrednost1);
                if(uredjenjeS!=0)
                        return uredjenjeS;
                return this.vrednost2.compareTo(o.vrednost2);
        }
}
```

У прилагођеној верзији генеричког уређеног пара постоје две новине:

1. генерички уређени пар имплементира интерфејс за упоређивање са другим генеричким паровима `Comparable<UredjeniParUporediv<S, T>>`;
2. сваки од типова `S` и `T` је такође упоредив тип.

Ставка 2. ограничава одозго генеричке параметре `S` и `T` генеричким интерфејсом `Comparable<S>` односно `Comparable<T>`.

```
package rs.math.oop.g13.p06.genericiMinimalniElementNiza;

public class MinimalniElementNiza {
    public static <T extends Comparable<T>> T nadjiMinimum(T[] niz) throws Exception {
        if (niz.length == 0)
                throw new Exception("Низ је празан - минимум нема смисла.");
        T minimum = niz[0];
        for (T element : niz)
                if (element.compareTo(minimum) < 0)
                        minimum = element;
        return minimum;
    }

    public static void main(String[] args) {
        String[] stringovi = new String[] { "Паја", "Ана", "Мика",
                                            "Марија", "Пера" };
        UredjeniParUporediv<Integer, Integer>[] parovi =
                (UredjeniParUporediv<Integer, Integer>[]) new UredjeniParUporediv[] {
                        new UredjeniParUporediv<Integer, Integer>(46, 21),
                        new UredjeniParUporediv<Integer, Integer>(10, 21),
                        new UredjeniParUporediv<Integer, Integer>(10, 19),
                        new UredjeniParUporediv<Integer, Integer>(15, 21),
                        };
        // UredjeniParUporediv<Color, Integer> par; // не компајлира се
        try {
                String minString = nadjiMinimum(stringovi);
                System.out.println(minString);
                UredjeniParUporediv<Integer, Integer> minPar = nadjiMinimum(parovi);
                System.out.println(minPar);
        } catch (Exception e) {
                System.err.println(e.getMessage());
        }
    }
}
```

Генерички метод за тражење минималног елемента такође користи ограничавање параметра типа одозго интерфејсом `Comparable<T>` будући да је за тражење минимума

потребно искористити упоређивање елемената низа. За демонстрацију рада метода искоришћен је низ ниски, а након тога и низ уређених парова. Није могуће инстанцирати низ генеричких типова `new UredjeniParUporediv<Integer, Integer>[...]` — слично као што је раније показано да није могуће позвати конструктор `new T[...]`. Због тога је употребљен конструктор за креирање сировог низа уређених парова `new UredjeniParUporediv[...]`, а након тога је извршена експлицитна конверзија у `UredjeniParUporediv<Integer, Integer>[]`.
Приметити да је декларисање уређеног пара у којем је прва координата класе `Color`, а друга класе `Integer`, коментарисано – компајлер не дозвољава овакву декларацију, јер класа `Color` није упоредива на уређење.
Испис који производи овај програм је следећи.

```
Ана
(10, 19)■
```

13.5. Генерици и виртуелна машина

Кад год се дефинише генерички тип, у позадини (на нивоу виртуелне машине) аутоматски се обезбеђује одговарајући сирови тип за њега. Име сировог типа је исто као име генеричког типа, с тим што су уколоњени параметри који представљају типове. Променљиве које представљају типове су замењене типовима који их ограничавају одозго или `Object` (ако за те променљиве није било ограничења). На пример, за раније уведену класу `KutijaGenericka<T>` где `T`, између осталог, параметризује тип поља `vrednost`, у позадини се креира сирова класа слична класи `KutijaSirova` у којој је параметар типа поља `vrednost` замењен класом `Object`. Додатно, компајлер убацује експлицитну конверзију ка одговарајућем типу `T` на свим местима где је то потребно. Другим речима, компајлер преводи генерички кôд у кôд са сировим типовима на сличан начин на који би то урадио програмер који користи само сирове типове. Када се користи бар једно ограничење за параметар `T`, уместо замене параметра типа `T` класом `Object`, врши се замена првим наведеним ограничавајућим типом. За преостала ограничења, ако их има више од једног, у коду се на одговарајућим местима спроводе експлицитне конверзије како би се добио жељени тип.
Пример 7. Написати Јава програм који класу `UredjeniParUporediv<...>` реализује употребом искључиво сирових типова. Слично као у примеру 6, реализовати статички метод за тражење минималног елемента низа и применити га на низ уређених парова. □

```
package rs.math.oop.g13.p07.genericiUredjeniParUporedivSirov;

public class UredjeniParUporedivSirov implements Comparable{
		private Comparable vrednost1;
		private Comparable vrednost2;

		public UredjeniParUporedivSirov(Comparable vrednost1,
				Comparable vrednost2) {
			this.vrednost1 = vrednost1;
			this.vrednost2 = vrednost2;
		}

		public Comparable getVrednost1() {
```

```
                return vrednost1;
        }

        public Comparable getVrednost2() {
                return vrednost2;
        }

        @Override
        public String toString() {
                return "("+vrednost1+", "+vrednost2+")";
        }

        @Override
        public int compareTo(Object o) {
                UredjeniParUporedivSirov par = (UredjeniParUporedivSirov) o;
                int uredjenjeS = this.vrednost1.compareTo(par.vrednost1);
                if(uredjenjeS!=0)
                        return uredjenjeS;
                return this.vrednost2.compareTo(par.vrednost2);
        }
}
```

Како је задатак да се генерички типови потпуно замене сировим, уз задржавање функционалности, изведене су следеће замене типова.
Уместо досадашњег потписа класе

```
public class UredjeniParUporediv<S extends Comparable<S>, T extends Comparable<T>>
        implements Comparable<UredjeniParUporediv<S, T>>
```

нови потпис је:

```
public class UredjeniParUporedivSirov implements Comparable
```

уз додатну експлицитну конверзију унутар метода `compareTo()`:

```
UredjeniParUporedivSirov par = (UredjeniParUporedivSirov) o;
```

и замене свих појављивања типова `S` и `T` интерфејсом `Comparable`.

```
package rs.math.oop.g13.p07.genericiUredjeniParUporedivSirov;

public class MinimalniElementNizaSirov {

        public static Comparable nadjiMinimum(Comparable[] niz) throws Exception{
                if(niz.length==0)
                        throw new Exception("Низ је празан - минимум нема смисла.");
                Comparable minimum = niz[0];
                for(Comparable element: niz)
                        if(element.compareTo(minimum)<0)
                                minimum = element;
                return minimum;
        }

        public static void main(String[] args) {
                UredjeniParUporedivSirov[] parovi =new UredjeniParUporedivSirov[]
                        {
                                new UredjeniParUporedivSirov(46, 21),
                                new UredjeniParUporedivSirov(10, 21),
                                new UredjeniParUporedivSirov(10, 19),
                                new UredjeniParUporedivSirov(15, 21),
```

```
                        };
                try {
                        UredjeniParUporedivSirov minPar =
                                        (UredjeniParUporedivSirov) nadjiMinimum(parovi);
                        System.out.println(minPar);
                } catch (Exception e) {
                        System.err.println(e.getMessage());
                }
        }
}
```

Сличне измене је претрпео и кôд за тражење минималног елемента низа. Резултат извршавања, сада сировог метода за тражење минималног елемента у низу уређених парова, је исти као и раније.

```
(10, 19)■
```

13.6. Генерици и наслеђивање

У каквој су релацији две генеричка објекта уколико су њихови параметри у релацији наслеђивања? На пример, ако су параметри типови `Integer` и `Object`, и познато је да је `Integer` потклаcа класе `Object`, да ли је онда и `KutijaGenericka<Integer>` потклаcа класе `KutijaGenericka<Object>`?

Одговор је одречан. Наиме, објекти ових класа нису ни у каквој релацији те се инстанцна променљива типа `KutijaGenericka<Integer>` не може уопштити помоћу објектне променљиве типа `KutijaGenericka<Object>`. За разлику од овога, инстанцна променљива типа `Integer` се може имплицитно конвертовати у објектну променљиву типа `Object`.

Са друге стране, релација наслеђивања се може успоставити између две генеричке класе. Ову могућност демонстрира наредни пример.

Пример 8. Написати Јава програм који реализује генеричку поткласу претходно уведене класе `KutijaGenericka<T>`. Потклаcа треба да омогући прослеђивање боје кутије, третирајући је као аргумент конструктора. Након тога, тестирати имплицитну конверзију између инстанцних променљивих из смера потклаcе ка наткласи.□

```
package rs.math.oop.g13.p08.genericiKutijaGenerickaObojena;

import java.awt.Color;

public class KutijaGenerickaObojena<T> extends KutijaGenericka<T>{
        private Color boja;

        public KutijaGenerickaObojena(Color boja){
                super();
                this.boja=boja;
        }

        public Color getBoja() {
                return boja;
        }

        public static void main(String[] args) {
                KutijaGenerickaObojena<String> kutijaObojena =
```

```
                        new KutijaGenerickaObojena<String>(Color.red);
                kutijaObojena.postaviVrednost("Текст");
                KutijaGenericka<String> kutija = kutijaObojena;
                System.out.println(kutija.uzmiVrednost());
        }
}
```

Као што се може видети, `KutijaGenerickaObojena<T>` се може уопштити имплицитном конверзијом у `KutijaGenericka<T>`. Наравно, могућа је и експлицитна конверзија у супротном смеру.
Пример се компајлира и производи следећи испис.

```
Текст■
```

13.7. Резиме

Генерички типови су моћан механизам који омогућава писање генеричких алгоритама и растерећивање програмера од писања истог или сличног кода већи број пута. Јава омогућава да се генеричност дефинише на глобалном или локалном нивоу, односно на нивоу целих класа и интерфејса или на нивоу појединачних метода. Применом механизма ограничавања одозго, програмер може додатно прецизирати које предуслове тип мора да испуњава како би се класа/интефејс/метод могао њиме параметризовати. Иако делује да је систем генеричких типова врло чврсто спрегнут са Јава језиком и Јава компајлером, веза је „лабавија“. Генерички типови представљају синтаксну олакшицу програмеру (слично као и набројиви типови). Они се у фази препроцесирања трансформишу у сирове (негенеричке) типове, након чега се врши компилација. Генерички типови су корисни, јер омогућавају бољу контролу типова и избегавање (не и потпуну елиминацију) експлицитних конверзија, које су један од чешћих разлога појаве грешака у фази извршавања програма.

13.8. Питања и задаци

1. Које су предности употребе генеричких типова?
2. Написати Јава програм у којем се дефинише генерички интерфејс за ред структуру података. Након тога дефинисати класу која имплементира генерички ред интерфејс помоћу низа и тестира његове могућности.
3. Шта је самостални генерички метод и када се користи? Илустровати примером.
4. Шта се подразумева под ограничењем параметарског типа `T` са „горње стране“? Које интерпретације ограничења са „горње стране“ постоје?
5. Написати Јава програм који реализује статички самостални генерички метод за тражење максималног елемента у низу. Након тога применити метод на низ објеката неке од уграђених класа програмског језика Јава и на низ објеката кориснички дефинисане генеричке класе по избору.
6. Шта су сирови типови? Примером илустровати употребу сирових типова.
7. У каквој су релацији два генеричка објекта уколико су њихови параметри у релацији наслеђивања? Илустровати примером.
8. Како се концепт генеричких типова уклапа у концепт наслеђивања? Односно, да ли генеричка класа може да наследи неку другу генеричку класу, да ли може да

наследи сирову класу? Да ли сирова класа може да наследи генеричку класу? Ако је могуће, илустровати примером.

9. Да ли је могуће превазићи наслеђени генерички метод из базне класе генеричким методом у изведеној класи? Ако је могуће, илустровати примером.

14. Колекције и речници

Колекције и речници припадају тзв. апстрактним типовима података (АТП). Под АТП подразумевамо групу елемената од којих сваки чува некакав податак (или метаподатак) и повезан је (на логичком нивоу – не мора бити повезан референцом, већ имплицитно кроз контекст) са једним или више других елемената.
АТП има уграђене унутрашње операције које могу бити извршене на њему или помоћу њега. Корисник АТП не мора имати никакве информације о унутрашњој репрезентацији података и операција. АТП се описује преко једне или више класа.
Разумевање апстрактних типова података битно за развој било које врсте софтвера.
Треба знати: пројектовати, користитити и имплементирати апстрактне типове података у Јави.
Колекција представља набројиву групу објеката над којом је могуће вршити операције писања, читања, измене или сумаризације (агрегирања). На пример, може бити: колекција слова у неком језику, бројева телефона, врста животиња, боја и слично. Колекција може имати додатне карактеристике попут уређености, непостојања дупликата итд.
Речници (каталози, мапе) су рачунарски концепт за представљање математичке функције над набројивим доменом. У рачунарској нотацији се уместо термина домен и кодомен користе термини скуп кључева и колекција вредности, респективно. На пример, пресликавање скупа јединствених матичних бројева у колекцију имена и презимена особа је речник. Речник такође представља пресликавање скупа регистарских ознака (бројева таблица) у колекцију одговарајућих аутомобила.
Због блискости и зависности између концепата колекције и речника, ове две теме се често истовремено изучавају. У литератури неки аутори чак и саме речнике сврставају у колекције, јер се речник може посматрати као уређени пар две колекције.
Јава је на почетку испоручивана са свега неколико класа за рад са колекцијама података попут `Vector`, `Stack`, `Hashtable`, `BitSet` и интерфејсом `Enumeration`. У даљем развоју Јаве је требало помирити супротстављене захтеве:

- библиотека за колекције и речнике треба да буде мала и да се лако може користити;
- да не буде сложена као што је STL код C++, али да омогући рад са генеричким алгоритмима на начин сличан оном који се користи у STL-у;
- било је потребно да се старе, већ испоручене класе, природно уклопе у нови оквир.

Јава колекције и речници представљају темељно испланирану Јава библиотеку која је заснована на следећа два хронолошки уређена концепта:

1. интерфејси — општи поглед на могућности колекција и речника, и установљавање разлика између њих;
2. имплементације (и алгоритми) — конкретизације одговарајућих интерфејса уз употребу разноврсних (сакривених) структура података попут низова, повезаних листи и њима придружених метода.

У даљем тексту ће се Јава колекције и речници изучавати систематично с обзиром на наведену хронологију. У великом броју наредних примера и појашњења ће се користити концепт генеричких типова, уведених у поглављу 13.

14.1. Интерфејс и имплементација

Библиотека за Јава колекције и речнике стриктно раздваја интерфејсе од имплементација. Начин раздвајања ће бити детаљније приказан на колекцији ред (енг. queue), која је корисна када елементе треба обрађивати по редоследу приспећа (енг. first in, first out — FIFO).
(Напомена: постоје и редови који не поштују FIFO принцип, и о њима ће бити речи у секцији 14.4.3. Уобичајено је, међутим, да се термин ред односи баш на FIFO ред.)
Пример 1. Дефинисати генерички интерфејс за ред који чине методи за додавање елемента на крај реда, узимање са почетка и добијање информације о броју елемената реда. Након тога, два пута имплементирати уведени интерфејс за генеричке типове података: употребом кружног низа и повезане листе.□

```
package rs.math.oop.g14.p01.kolekcijeRed;

interface Red<T>{
	void dodaj(T element);
	T ukloni();
	int brojElemenata();
}
```

(Напомена: интерфејс `Red<T>` је већ дефинисан, у нешто сложенијој форми, под називом `Queue<T>`, у оквиру стандардне библиотеке за Јава колекције и речнике.)

```
package rs.math.oop.g14.p01.kolekcijeRed;

public class RedPrekoKruznogNiza<T> implements Red<T> {
	private T[] redNiz;
	private int kapacitet;
	private int brojElemenata;
	private int pocetakIndeks;
	private int naredniIndeks;

	public RedPrekoKruznogNiza() {
		kapacitet = 5;
		redNiz = alocirajRedNiz();
	}

	private T[] alocirajRedNiz() {
		System.out.println("Алоцирам низ капацитета: " + kapacitet);
		return (T[]) new Object[kapacitet];
	}

	@Override
	public void dodaj(T element) {
		System.out.println("Додајем " + element);
		if (brojElemenata==kapacitet) {
			kapacitet *= 2;
			T[] noviRedNiz = alocirajRedNiz();
			for (int ip = 0; ip < brojElemenata; ip++) {
				int i = (ip + pocetakIndeks) % brojElemenata;
				noviRedNiz[ip] = redNiz[i];
			}
			redNiz = noviRedNiz;
			pocetakIndeks=0;
```

```
                        naredniIndeks=brojElemenata;
                }
                redNiz[naredniIndeks] = element;
                naredniIndeks = (naredniIndeks + 1) % kapacitet;
                brojElemenata++;
        }

        @Override
        public T ukloni() {
                if (brojElemenata == 0) {
                        System.out.println("Ред је празан па нема смисла уклањање.");
                        return null;
                }
                T element = redNiz[pocetakIndeks];
                redNiz[pocetakIndeks] = null;
                pocetakIndeks = (pocetakIndeks + 1) % kapacitet;
                brojElemenata--;
                System.out.println("Уклањам: "+element);
                return element;
        }

        @Override
        public int brojElemenata() {
                return brojElemenata;
        }

        public static void main(String[] args) {
                Red<Integer> red = new RedPrekoKruznogNiza<>();
                red.dodaj(34);
                red.dodaj(23);
                red.dodaj(11);
                System.out.println("Број елемената: " + red.brojElemenata());
                red.ukloni();
                System.out.println("Број елемената: " + red.brojElemenata());
                red.dodaj(112);
                System.out.println("Број елемената: " + red.brojElemenata());
                red.dodaj(-134);
                System.out.println("Број елемената: " + red.brojElemenata());
                red.dodaj(111);
                System.out.println("Број елемената: " + red.brojElemenata());
                red.dodaj(345);
                System.out.println("Број елемената: " + red.brojElemenata());
                red.ukloni();
                red.ukloni();
                red.ukloni();
                red.ukloni();
                red.ukloni();
                red.ukloni();
                red.ukloni();
                System.out.println("Број елемената: " + red.brojElemenata());
        }
}
```

Имплементација реда преко кружног низа подсећа на раније уведену имплементацију стека преко низа из секције 13.2.1. Међутим, овде није искоришћена раније уведена

структура саморастућег генеричког низа, јер је манипулација кружним низом и његовим проширивањем захтевала мало специфичније понашање.
За опис реда је потребно имати имати два индекса: за почетак реда и за прво наредно слободно место (прво место након краја реда). Слично као код реализације стека, и овде је потребно водити рачуна о капацитету унутрашњег низа. У случају да је капацитет исцрпљен, спроводи се формирање дупло већег низа и преписивање свих елемената. Такође, у методима за додавање и уклањање елемента мора да се води рачуна о кружном понашању индекса за почетак реда и за први наредни — могуће је да, након неког броја уклањања и додавања елемената, индекс за почетак реда буде већи од индекса за наредни.
Предност саморастућег кружног низа у односу на саморастући стандардни низ је боља искоришћеност меморије при употреби операција над придруженим редом. Наиме, код стандардног низа се може након великог броја уклањања елемената формирати значајна количина неискористиве меморије са предње стране низа – то није могуће код кружног низа.
Следи испис који производи програм.

```
Алоцирам низ капацитета: 5
Додајем 34
Додајем 23
Додајем 11
Број елемената: 3
Уклањам: 34
Број елемената: 2
Додајем 112
Број елемената: 3
Додајем -134
Број елемената: 4
Додајем 111
Број елемената: 5
Додајем 345
Алоцирам низ капацитета: 10
Број елемената: 6
Уклањам: 23
Уклањам: 11
Уклањам: 112
Уклањам: -134
Уклањам: 111
Уклањам: 345
Ред је празан па нема смисла уклањање.
Број елемената: 0
```

```java
package rs.math.oop.g14.p01.kolekcijeRed;

public class RedPrekoPovezaneListe<T> implements Red<T>{
    private PovezanaLista<T> elementi = new PovezanaLista();

    @Override
    public void dodaj(T elem) {
        System.out.println("Додајем " + elem);
        elementi.dodajNaKraj(elem);
    }
```

```
    @Override
    public T ukloni() {
        T elem = elementi.ukloniSaPocetka();
        if (elem == null) {
                System.err.println("Грешка при уклањању: ред је празан!");
                return null;
        }
        System.out.println("Уклањам: "+elem);
        return elem;
    }

    @Override
    public int brojElemenata() {
        return elementi.brojCvorova();
    }

    public static void main(String[] args) {
        Red<Integer> red = new RedPrekoPovezaneListe<>();
        red.dodaj(34);
        red.dodaj(23);
        red.dodaj(11);
        System.out.println("Број елемената: " + red.brojElemenata());
        red.ukloni();
        System.out.println("Број елемената: " + red.brojElemenata());
        red.dodaj(112);
        System.out.println("Број елемената: " + red.brojElemenata());
        red.dodaj(-134);
        System.out.println("Број елемената: " + red.brojElemenata());
        red.dodaj(111);
        System.out.println("Број елемената: " + red.brojElemenata());
        red.dodaj(345);
        System.out.println("Број елемената: " + red.brojElemenata());
        red.ukloni();
        red.ukloni();
        red.ukloni();
        red.ukloni();
        red.ukloni();
        red.ukloni();
        red.ukloni();
        System.out.println("Број елемената: " + red.brojElemenata());
    }
}
```

Реализација реда преко повезане листе је потпуно ослоњена на раније уведену класу `PovezanaLista<T>` (пример 4 из секције 13.2.1). Додавање у ред позива метод `dodajNaKraj()`, док уклањање из реда подразумева позив метода `ukloniSaPocetka()`. Треба приметити да је у обе имплементације инстанцна променљива која реферише на новокреиране редове сакривена иза инстанцне променљиве интерфејса `Red<Integer>`.

Ово је корисно, јер остале класе, интерфејси и методи, који користе ред, не морају експлицитно наводити какав ред очекују.
Испис произведен радом овог програма је исти као и у претходној имплементацији реда.■

14.2. Колекције и итератори

Најважније могућности у раду са колекцијама су:

- додавање елемента;
- уклањање елемента;
- испитивање да ли колекција садржи елемент;
- испитивање величине колекције и
- набрајање елемената.

За потребе набрајања елемената, дизајнери колекцијске библиотеке су морали да осмисле општи и проширив начин. Приметити да набројивост не казује ништа о уређењу, тј. не прецизира редослед елемената у оквиру колекције.

14.2.1. Интерфејс Collection

Корени интерфејс колекцијске библиотеке је генерички интерфејс `Collection`.

```
public interface Collection<T>{
   boolean add(T element);
   Iterator<T> iterator();
   boolean contains(Object o);
   boolean remove(Object o);
   int size();
   ...
}
```

Горе наведени методи су најзначајнији, али постоје још неки методи који ће бити описани касније.
Метод `add()` додаје елемент у колекцију . Овај метод враће `true` ако је додавање елемента променило колекцију (што значи да је додавање успело), а враћа `false` ако је колекција остала непромењена (додавање није успело). На пример, ако се додаје у скуп елемент који се већ у њему налази, тада колекција остаје непромењена и метод `add()` враћа `false`.
Метод `iterator()` враћа објекат који имплементира интерфејс `Iterator`. Тако добијени објекат итератор се даље користи за набрајање свих елемената који се налазе у колекцији.
Метод `contains()` испитује да ли се прослеђени објекат налази у колекцији. За испитивање једнакости користи се метод `equals()` који је дефинисан за произвољни генерички тип `T` (генерички типови се могу заменити искључиво објектним типом, а не и примитивним).
Слично, метод `remove()` на основу метода `equals()` лоцира елемент и након тога га уклања или у супротном враћа `false`.
Метод `size()` враћа број елемената колекције.

14.2.2. Интерфејси Iterable и Iterator

Концепт употребе итератора, односно набројивости, није карактеристичан само за колекције. Стога је та могућност издвојена у интерфејс под називом `Iterable`.
Интерфејс `Iterable` обавезује класу која га имплементира (или интерфејсе који га проширују) да буде набројива. Најзначајнији метод у интерфејсу `Iterable` је `iterator()`.

```
public interface Iterable<T>{
  Iterator<T> iterator();
  ...
}
```

Овај метод враћа генерички објекат типа `Iterator`. Генерички интерфејс `Collection` проширује интерфејс `Iterable`, па између осталих метода поседује и метод `iterator()`.
`Iterator` је такође интерфејс. Он садржи четири метода, при чему су, за даље разумевање рада итератора, значајна прва три.

```
public interface Iterator<T>{
   T next();
   boolean hasNext();
   void remove();
   void forEachRemaining(Consumer<? super T> action);
}
```

Позивом метода `next()` посећује се наредни елемент колекције, при чему се првим позивом метода `next()` итератор позиционира на први елемент колекције. Ако се стигне до краја колекције и тада позове метод `next()`, биће избачен изузетак `NoSuchElementException`. Стога је потребно, пре сваког позива метода `next()`, позвати метод `hasNext()`. Метод `hasNext()` враће `true` ако објекат итератор није стигао до краја колекције. За набрајање свих елемената колекције потребно је креирати итератор објекат и потом над њим понављати позивање метода `next()` све док метод `hasNext()` враћа `true`. Почев од Јава 5.0, итератор се елегантније може записати помоћу тзв. колекцијске наредбе `for`.
Пример 2. Демонстрирати употребу итератора над произвољном генеричком колекцијом. Урадити исто и помоћу колекцијске наредбе `for`. □

```
package rs.math.oop.g14.p02.iteratorLista;

import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;

public class IteratorPovezanaLista {
	public static void main(String[] args) {
		Collection<String> kolekcija = new LinkedList<String>();
		kolekcija.add("Марко");
		kolekcija.add("Марија");
		kolekcija.add("Ивана");
		kolekcija.add("Дарко");
		Iterator<String> iterator = kolekcija.iterator();
		while(iterator.hasNext()) {
			String element = iterator.next();
			System.out.println(element);
		}
		for(String element : kolekcija)
```

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

}
```

Након покретања програма биће исписани сви елементи унутар колекције. Будући да је реч о листи, код које важи принцип уређења, редослед исписа ће одговарати редоследу уноса у листу.
Може се лако проверити да набрајања путем итератора и путем колекцијске наредбе `for` производе идентичан испис.

```
Марко
Марија
Ивана
Дарко
Марко
Марија
Ивана
Дарко■
```

14.2.3. Операције над колекцијом коришћењем итератора

Позиција итератора је у сваком моменту "између" елемената колекције.

```
                Елемент(0)   Елемент(1)   Елемент(2)   ...  Елемент(n-1)
позиције      ^            ^            ^              ^               ^
```

За n елемената постоји n+1 позиција итератора, те се може рећи да итератор никад није позициониран на елементу, већ између елемената (осим у случају када је пре почетног или након последњег). При позиву метода `next()`, итератор прескаче следећи елемент и враћа референцу на елемент који је управо прескочио. Метод `remove()` интерфејса `Iterator` уклања елемент враћен последњим позивом метода `next()`. Ово је корисно у ситуацијама када елемент из колекције уклањамо на основу његове вредности. Тада се елемент „прескочи", његова вредност се помоћу метода `equals()` упореди са елементом који се уклања, и онда се елемент уклони уколико је `equals()` вратио `true`. У случају да се елемент уклања на основу позиције у колекцији (има смисла код уређених колекција), принцип је сличан. Међутим, том случају се не користи метод `equals()`, већ се броји до одређеног броја позива метода `next()`, који одговара траженој позицији. На пример, уклањање првог елемента колекције би могло да се изведе на следећи начин.

```
Iterator String it = c.iterator();
it.next(); // прескочи се први елемент
it.remove(); // сада се уклони
```

Уколико се позове `remove()` пре `next()`, долази до избацивања изузетка `IllegalStateException`. Ако треба да се уклоне два суседна елемента, не може се `remove()` позвати два пута узастопно.

```
it.remove();
it.remove(); // грешка!
```

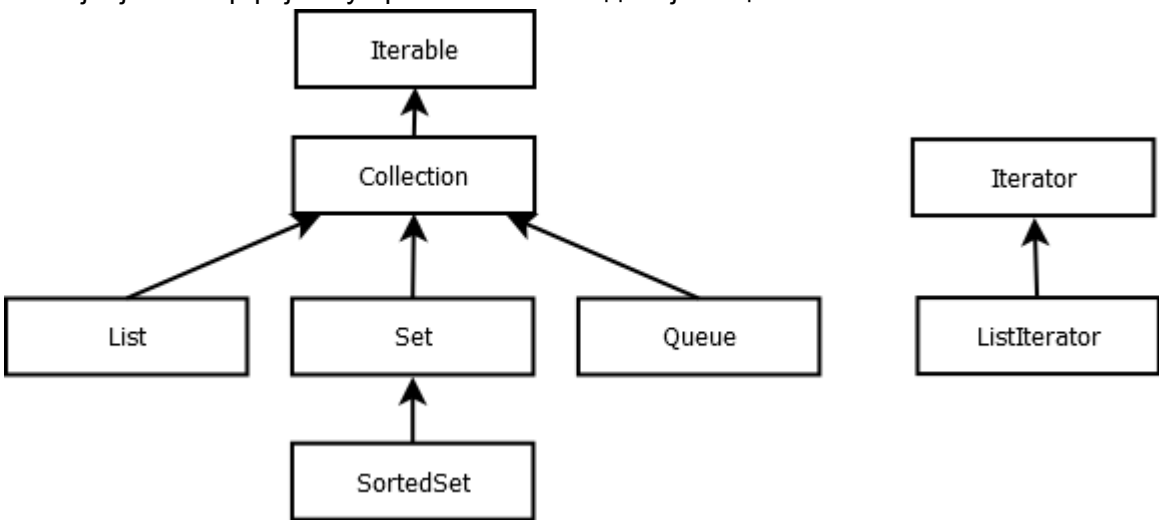
Уместо тога, прво се мора позвати метод `next()` како би итератор прешао преко наредног елемента који треба уклонити.

```
it.remove();
```

```
it.next();
it.remove();
```

14.3. Колекцијски интерфејси

Јава садржи хијерархију интерфејса за рад са колекцијама и речницима. Неки од значајнијих интерфејса су приказани на следећој слици.



Хијерархија неких колекцијских интерфејса у Јави

У Јави постоји велики број класа које имплементирају наведене интерфејсе. Стога их је могуће наследити и прилагодити, али је могуће и написати нове класе без наслеђивања постојећих уз имплементацију свих прописаних метода. Будући да је о интерфејсима `Collection`, `Iterable` и `Iterator` већ било речи, у даљем тексту фокус ће бити на колекцијским интерфејсима: `List`, `Set`, `Queue`.

14.3.1. Листа, интерфејс List

Листа је уређена колекција, позната и под називом секвенца. За разлику од општијег интерфејса `Collection`, код интерфејса `List` корисник има контролу над позицијом сваког елемента. Та контрола се огледа у различитом потпису метода за додавање и приступање елементима.

```
public interface List<T>{
   boolean add(T element);
   void add(int index, T element);
   T get(int index);
   ListIterator<T> listIterator();
   ...
}
```

Као што се види из скраћеног списка потписа метода, стандардни метод за додавање је наслеђен из над-интерфејса `Collection`. Његова улога је додавање елемента на неку подразумевану локацију, на пример, крај листе. Такође, за разлику од интерфејса `Collection`, `List` интерфејс обезбеђује и приступ специфичном типу итератора `ListIterator`. Овај итератор наслеђује интерфејс `Iterator` и при том у класи која га имплементира мора да буде познато, не само које елементе садржи, већ и које су њихове позиције.

Списак метода `ListIterator` интерфејса је нешто већи, јер укључује померање итератора у оба смера.

```
public interface ListIterator<T> extends Iterator<T>{
   void add(T element); // додавање испред позиције итератора
   boolean hasPrevious(); // true ако постоји још неки елемент до краја
   int nextIndex();
   // враћа индекс елемента који ће бити враћен наредним next() позивом
   T previous(); // враћа претходни елемент и помера итератор уназад
   int previousIndex(); // попут nextIndex() методе
   void set(T element);
   // мења последњи елемент добијен next() или previous() са element
}
```

Методи `List` интерфејса за приступање елементу или додавање елемента на задатој позицији (`get()` и `add()`) засновани су на интерфејсу `ListIterator`. Његовом употребом је омогућено позиционирање на циљну позицију, након чега следи читање односно додавање елемента.

14.3.2. Ред, интерфејси Queue и Dequeue

Редови омогућавају ефикасно додавање елемената на крај и уклањање елемената са почетка. Додавање елемената у средину није подржано. Овде је реч о редовима који функционишу по принципу FIFO. Принцип FIFO се често користи у оперативним системима, рачунарским мрежама, и у развоју софтвера. Ово је један од разлога због којих се ред сврстава у основне колекцијске интерфејсе.
Преглед свих метода интерфејса `Queue` је дат испод.

```
public interface Queue<T>{
   boolean add(T element);
   boolean offer(T element);
   T remove();
   T poll();
   T element();
   T peek();
}
```

Методи `add()` и `offer()` додају елемент на крај и потом враћају `true` ако је било места у реду, тј. елемент је успешно додат. Разлика између ова два метода је у начину на који реагују ако нема довољно места у реду за додавање новог елемента, `add()` тада избацује изузетак `IllegalStateException`, док `offfer()` само враћа `false`.
Слична дуалност постоји и између метода `remove()` и `poll()`. Оба уклањају елемент са почетка реда и враћају га као повратну вредност у успешном сценарију. У случају да је ред празан, `remove()` ће избацити изузетак `NoSuchElementException`, док ће `poll()` вратити `null`.
На крају, методи `element()` и `peek()` враћају елемент с почетка реда, а да га при том не уклањају из реда. Као и код претходних метода, `element()` метод експлицитно наглашава да је ред празан избацивањем изузетка `NoSuchElementException`, док `peek()` враћа `null`.
Ред са два краја, (енг. deque), омогућава ефикасно додавање и уклањање елемената и са почетка и са краја. Интерфејс `Dequeue` има више метода него `Queue` будући да неки методи из `Queue` сада имају и суфикс `First` и `Last`. На пример, постоје методи `addFirst()` и `addLast()`. Због тога је `Dequeue` могуће третирати и као LIFO колекцију

(енг. last in first out), односно стек — елементи се могу додавати на почетак (крај) и скидати са почетка (краја).

14.3.3. Скуп, интерфејс Set

Јава скупови, попут оних у математици, не смеју да садрже дупликате. То значи да методи наслеђени из интерфејса `Collection` овде прописују специфичније понашање. Методи `add()` и `contains()`, између осталих, морају подржати забрану постојања дупликата. На пример, покушај убацивања (метод `add()`) елемента у скуп који га већ садржи (познато на основу реализације метода `equals()`) резултира непромењеним скупом и повратном вредношћу `false`. У наставку је приказан списак неких релевантнијих метода интерфејса `Set`.

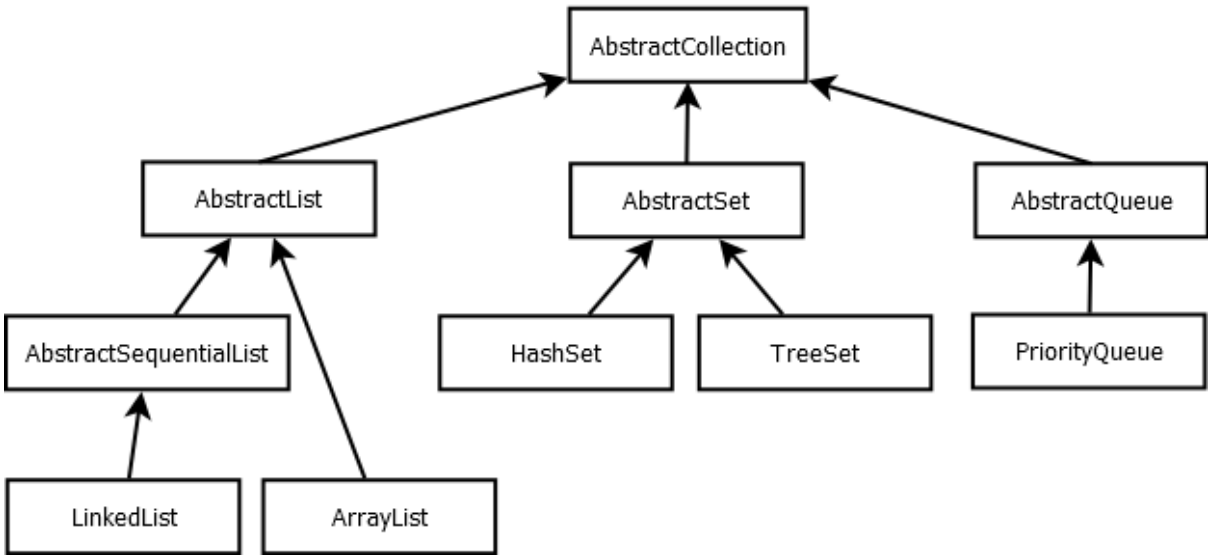
```
public interface Set<T>{
   boolean add(T element); // додавање, ако елемент већ није унутра
   boolean addAll(Collection<? extends T>c); // попут уније, с не мора бити скуп
   void clear(); // формира празан скуп
   boolean equals(); // поређење скупова, небитан редослед
   boolean remove(Object o); // уклањање, ако equals() за неки елемент врати true
   boolean removeAll(Collection<?> c);
   // попут разлике скпуова, с не мора бити скуп
   boolean retainAll(Collection<?> c);
   // попут пресека скупова, с не мора бити скуп
}
```

Иако математички гледано за скуп није од значаја уређење, програмеру може бити корисно да уреди елементе по неком критеријуму, на пример, времену убацивања, вредности или слично. Из тог разлога постоји и додатни интерфејс под називом `SortedSet` који омогућава уређивање елемената скупа. Уређивање је обезбеђено имплементацијом интерфејса `Comparable` или алтернативно експлицитним навођењем начина поређења кроз интерфејс `Comparator` (погледати секцију 9.3). О конкретним примерима употребе уређених скупова ће бити речи када се буде разматрала једна од његових стандардних имплементација под називом `TreeSet`.

14.4. Колекцијске класе

На следећој слици је приказана хијерахија класа у оквиру библиотеке за рад са колекцијама. Може се приметити да су конкретне колекцијске класе изведене из

апстрактних колекцијских класа. Више детаља о апстрактним колекцијским класама ће бити дато у секцији 14.7.



Хијерархија неких колекцијских класа у Јави

У наставку ће свака од класа бити детаљније размотрена. Пре тога следи кратки опис сваке од њих.

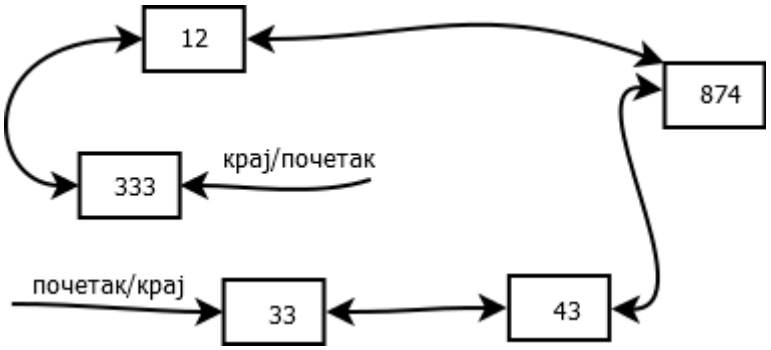
- `LinkedList` — повезана листа са брзим уметањем и брисањем на било којој позицији.
- `ArrayList` — индексирана листа са брзим случајним приступом (енг. random access), смањује се и расте динамички.
- `HashSet` — скуп са брзом провером припадности.
- `TreeSet` — скуп са уређеним елементима.
- `PriorityQueue` — ред са уклањањем елемента који има највећи/најмањи приоритет (није FIFO).

14.4.1. Листе

Овде ће бити речи о конкретним библиотечким имплементацијама `List` интерфејса. У питању су две суштински различите имплементације под називом `LinkedList` и `ArrayList`.

Двоструко повезана листа, класа LinkedList

Логичка меморијска организација двоструко повезане листе `LinkedList` је илустрована следећом сликом.



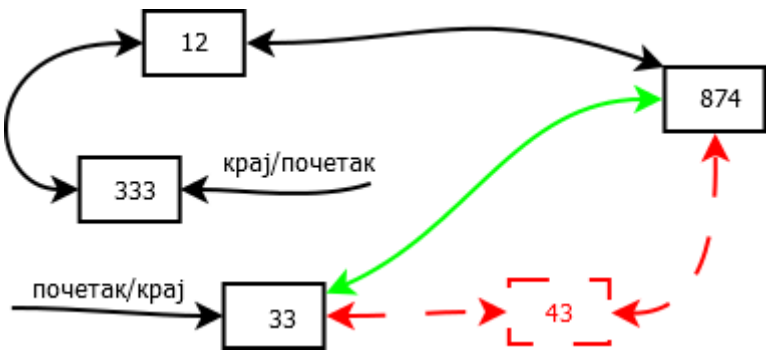
(Напомена: са једноструко повезаном листом читалац се упознао кроз мноштво примера из претходних поглавља – под називом `PovezanaListaNiski` или `PovezanaLista<T>`.)

Елементи повезане листе могу бити распоређени на произвољним меморијским локацијама. Сваки елемент, поред податка који енкапсулира (на пример, целог броја), има и референце према следећем и претходном елементу. Изузетак су елементи на почетку и крају листе који имају само долазне референце. Набрајање елемената је, стога, могуће у оба смера.

Приступ елементу листе на случајној позицији има сложеност O(n), јер је у најгорем случају потребно проћи n/2 елемената листе (n је број елемената листе).

Са друге стране, операције додавања или уклањања елемента са почетка или краја имају O(1) сложеност. (`LinkedList`, поред интерфејса `List`, имплементира и раније поменуте интерфејсе `Queue` и `Deque`.)

Повезана листа је погодна и за операције додавања или уклањања елемента са случајне позиције. За реализацију ових операција није потребно померати остале елементе листе са својих меморијских локација, за разлику од додавања или уклањања елемента из низа.



Пример 3. Демонстрирати рад са класом `LinkedList`: додавање и уклањање елемената, приступ елементима на случајној позицији и набрајање свих елемената листе у оба смера. ☐

```
package rs.math.oop.g14.p03.povezanaLista;

import java.util.LinkedList;
import java.util.ListIterator;

public class RadSaPovezanomListom {
	public static void main(String[] args) {
		LinkedList<Integer> povezanaLista = new LinkedList<Integer>();
		povezanaLista.addFirst(33); // [33]
		povezanaLista.addLast(12); // [33, 12]
		povezanaLista.add(333); // [33, 12, 333] - додаје на крај
		povezanaLista.add(1, 43); // [33, 43, 12, 33] - убацује на позицију 1
		ListIterator<Integer> iterator = povezanaLista.listIterator();
		iterator.next(); // итератор између 33 и 43
		iterator.next(); // итератор између 43 и 12
		iterator.add(874); //[33, 43, 874, 12, 333] - итератор између 874 и 12
		iterator.previous(); // итератор између 43 и 874
		iterator.previous(); // итератор између 33 и 43
		iterator.remove(); // уклања се следећи тј. 43

		System.out.println("Елементи од почетка ка крају:");
		for(Integer e : povezanaLista)
			System.out.println(e);

		System.out.println("Елементи од краја ка почетку:");
		iterator = povezanaLista.listIterator(povezanaLista.size());
		while(iterator.hasPrevious()) // читање уназад
			System.out.println(iterator.previous());

		System.out.println("Елементи од почетка ка крају помоћу индекса:");
		for(int i=0; i<povezanaLista.size(); i++)
			System.out.println(povezanaLista.get(i)); // неефикасно O(n)
	}
}
```

Поред метода `add()`, који подразумевано додаје елемент на крај листе, постоје и методи за додавање на почетак `addFirst()` и на крај `addLast()`. Такође је могуће додати на задату позицију (индекс) помоћу преоптерећеног метода `add()`.

```
public void add(int index, E element);
```

Овај метод најпре позиционира итератор на одговарајућу позицију у листи, након чега се врши додавање. Уместо употребе овог метода, у примеру изнад је приказан основнији приступ за додавање елемента који користи метод `next()`, класе `ListIterator` у комбинацији са методом `add()` (додавање броја 874 на трећу позицију). По истом принципу функционишу и методи за уклањање елемената.

У примеру се такође демонстрира неколико начина за исписивање свих елемената листе. Први приступ користи колекцијску `for` наредбу, док други користи експлицитни итератор за читање уназад — итератор се најпре позиционира на крај листе, након чега се користи метод за кретање уназад `previous()`. Трећи приступ је неефикасан у случају

повезане листе, јер се елементима приступа помоћу метода `get()`, који враћа вредност на траженој позицији. Разлог је неефикасност `get()` метода, тј. сложеност O(n).
Следи испис произведен радом претходно наведеног програма.

```
Елементи од почетка ка крају:
33
874
12
333
Елементи од краја ка почетку:
333
12
874
33
Елементи од почетка ка крају помоћу индекса:
33
874
12
333■
```

Низовна листа, класа ArrayList

Низовна листа `ArrayList` је класа која енкапсулира низ објеката и омогућава манипулацију над тим низом помоћу метода за: приступ елементу низа на одређеној позицији, додавања елемента на одређену позицију, уклањања елемента са позиције итд.
(Напомена: слична структура података је `SamorastuciNiz<T>`, чија је имплементација дата у примеру 4 секције 13.2.1.)
Највећа погодност за програмера, у поређењу са обичним низом, је аутоматско проширивање или смањивање капацитета низа (реалокација).
Временска сложеност стандардних операција је иста као код низа објеката. Због тога је низовна листа бољи избор од повезане листе у ситуацијама када је потребно учестало приступати или мењати вредности елемената на случајним позицијама. Карактеристика брзог случајног приступа је формализована кроз имплементацију интерфејса `RandomAccess`. Овај интерфејс не садржи декларације метода. Његова улога је да укаже на то да класа имплементира брз случајни приступ (маркерски интерфејс попут `Cloneable` из секције 9.3.4). Ово надаље може бити корисно генеричким алгоритмима у одлучивању коју класу да примене у зависности од тога да ли је потребно извршити секвенцијални (редни) приступ или случајни приступ подацима.
Са друге стране, ако је потребно често додавати (уметати) елементе, повезана листа је погоднија, јер не захтева померање елемената који следе након позиције на коју се додаје елемент. Слично важи и за уклањање елемента са неке позиције.
Пример 4. Демонстрирати рад са класом `ArrayList`: додавање и уклањање елемената, приступ елементима на задатој позицији и набрајање свих елемената листе у оба смера. □

```
package rs.math.oop.g14.p04.nizovnaLista;

import java.util.ArrayList;
import java.util.ListIterator;

public class RadSaNizovnomListom {
```

```
	public static void main(String[] args) {
		ArrayList<Integer> nizovnaLista = new ArrayList<Integer>();
		nizovnaLista.add(33); // [33]
		nizovnaLista.add(43); // [33, 43]
		nizovnaLista.add(12); // [33, 43, 12]
		nizovnaLista.add(2, 871); // [33, 43, 871, 12]
		nizovnaLista.set(2, 874); // [33, 43, 874, 12]
		nizovnaLista.add(333); // [33, 43, 874, 12, 333]
		nizovnaLista.remove(1); // [33, 874, 12, 333]

		System.out.println("Елементи од почетка ка крају:");
		for (Integer e : nizovnaLista)
			System.out.println(e);

		System.out.println("Елементи од краја ка почетку:");
		ListIterator<Integer> iterator;
		iterator = nizovnaLista.listIterator(nizovnaLista.size());
		while (iterator.hasPrevious()) // читање уназад
			System.out.println(iterator.previous());

		System.out.println("Елементи од почетка ка крају помоћу индекса:");
		for (int i = 0; i < nizovnaLista.size(); i++)
			System.out.println(nizovnaLista.get(i)); // ефикасно O(1)
	}
}
```

Низовна листа, као и повезана, поседује метод за додавање елемента на крај `add()`. Уколико је капацитет енкапсулираног низа довољан за додавање елемента, овај метод се извршава у O(1) сложености. У супротном је потребна реалокација.

Подржан је и метод за додавање на задату позицију. За приступање задатој позицији је потребно O(1) корака, међутим, потребно је још O(n) корака како би се сви елементи од циљне позиције па надаље померили за једно место удесно (повећање позиције сваког за по 1). Стога је операција додавања елемента на задату позицију мање ефикасна него код повезане листе, код које је потребно O(n) корака за позиционирање, али надаље не постоји трошак померања елемената.

Операције приступа елементу листе на траженој позицији `get()` и постављању вредности на позицији `set()` имају O(1) временску сложеност.

Низовна листа подржава експлицитни рад са итератором (или имплицитни уз помоћ колекцијске `for` наредбе). За разлику од итератора повезане листе, који је заснован на „праћењу“ референци ка следећем/претходном елементу, овде је стање итератора у сваком моменту представљено тренутном позицијом (бројем).

Следи испис произведен радом програма.

```
Елементи од почетка ка крају:
33
874
12
333
Елементи од краја ка почетку:
333
12
874
33
Елементи од почетка ка крају помоћу индекса:
```

```
33
874
12
333 ■
```

Пример 5. Упоредити брзине рада низовне и повезане листе. Конкретно, потребно је: 1) упоредити брзине додавања великог броја елемената у листе; 2) набрајање елемената тако формираних листи и 3) брзину случајног приступа одређеном броју елемената тако формираних листи.□

```
package rs.math.oop.g14.p05.implementacijaNizovneListe;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;

public class PoredjenjeSlucajnogPristupa {
    public static void main(String[] args) {
        int n = 10000000;
        int m = 1000;
        List<Integer> povezana = new LinkedList<>();
        List<Integer> nizovna = new ArrayList<>();
        Random rg = new Random(12345);

        long pocetak = System.nanoTime();
        for(int i=0; i<n; i++)
               povezana.add(i);
        System.out.printf("Убацивање %d ел. на крај повезане трајало %.2f сек.%n",
                      n, (System.nanoTime()-pocetak)/1e9f);

        pocetak = System.nanoTime();
        for(int i=0; i<n; i++)
               nizovna.add(i);
        System.out.printf("Убацивање %d ел. на крај низовне трајало %.2f сек.%n",
                      n, (System.nanoTime()-pocetak)/1e9f);

        pocetak = System.nanoTime();
        for(Integer x : povezana)
               ;
        System.out.printf("Набрајање %d ел. повезане трајало %.2f сек.%n",
                      n, (System.nanoTime()-pocetak)/1e9f);

        pocetak = System.nanoTime();
        for(Integer x: nizovna)
               ;
        System.out.printf("Набрајање %d ел. низовне трајало %.2f сек.%n",
                      n, (System.nanoTime()-pocetak)/1e9f);

        pocetak = System.nanoTime();
        for(int i=0; i<m; i++)
               povezana.get(rg.nextInt(n));
        System.out.printf("Приступање %d ел. у повезаној трајало %.2f сек.%n",
                      m, (System.nanoTime()-pocetak)/1e9f);

        pocetak = System.nanoTime();
```

```
        for(int i=0; i<m; i++)
               nizovna.get(rg.nextInt(n));
        System.out.printf("Приступање %d ел. у низовној трајало %.2f сек.%n",
                       m, (System.nanoTime()-pocetak)/1e9f);
    }
}
```

Следи испис времена извршавања.

```
Убацивање 10000000 ел. на крај повезане трајало 1.49 секунди.
Убацивање 10000000 ел. на крај низовне трајало 0.34 секунди.
Набрајање 10000000 ел. повезане трајало 0.08 секунди.
Набрајање 10000000 ел. низовне трајало 0.02 секунди.
Приступање 1000 ел. у повезаној трајало 18.62 секунди.
Приступање 1000 ел. у низовној трајало 0.00 секунди.
```

Евидентно је да убацивање великог броја елемената на крај повезане или низовне листе има сличну ефикасност. Слична је ситуација и са набрајањем елемената редом – последица линеарног броја O(1) операција (код низа је то померање на следећу позицију, а код повезане листе праћење референце).

Са друге стране, примећује се велика разлика по питању ефикасности случајног приступа. С обзиром да су листе врло велике (имају по десет милиона елемената), приступ елементу повезане листе на случајној позицији је веома спор. Ово не представља проблем за низовну листу, јер она имплементира брз случајни приступ, те не зависи од броја елемената енкапсулираног низа.■

14.4.2. Скупови

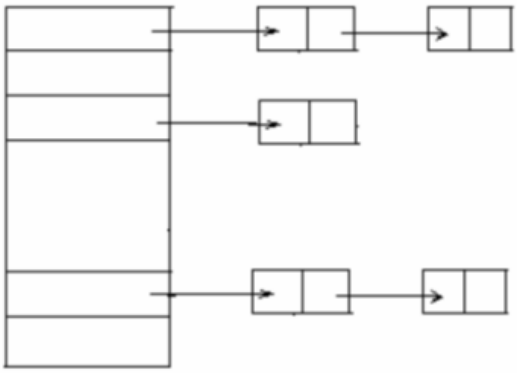
Овде ће бити описане две стандардне имплементације скупа које имају комплементарне карактеристике. `HashSet` је врло ефикасна имплементација која не подржава уређење, док је `TreeSet` мање ефикасна имплементација која подржава уређење. На крају ће бити кратко описано још неколико стандардних имплементација скупова попут `BitSet`, `HashLinkedSet` итд.

Хеш-скуп, класа HashSet

Ако је потребно пронаћи неки елемент у оквиру низа или листе, потребно је претражити, у најгорем случају, све елементе. Уколико је редослед елемената небитан, може се користити колекција која обезбеђује много бржи приступ елементима. Једино је незгодно што се тада не зна ништа о редоследу елемената, већ их та структура организује на произвољан начин.

Добро позната структура за брзо проналажење елемената је хеш табела. За разлику од низова, код којих су индекси цели бројеви, овде је позиција у структури одређена произвољним објектом. Ипак, имплицитно се захтева да сваком објекту буде придружен цели број тако што се рачуна његов хеш-код.

Отворена хеш табела је обично реализована као низ повезаних листи. За проналажење места објекта у хеш табели, израчунава се хеш-код (описан у секцији 8.5.4) и дели се по модулу са бројем елемената тог низа (повезаних листи). Резултат је индекс члана у низу тј. индекс повезане листе која садржи дати елеменат.



Неизбежно је да се понекад деси да има више елемената којима одговара исти индекс листе и тада долази до тзв. колизије. У том случају нови објекат се пореди са свим објектима из листе (метод `equals()`) како би се видело да ли је већ присутан у листи.
Дакле, хеш-код служи као апроксимација идентификатора објекта, што значи да ће у већини случајева (ако је `hashCode()` добро дефинисан) хеш-код омогућити лоцирање траженог објекта. У преосталим случајевима, када хеш-код није у стању да идентификује тражени објекат, идентификација ће бити извршена применом скупље операције `equals()` над свим елементима листе.
Ако хеш-кодови имају равномерну случајну расподелу и број листи је довољно велики, требало би да буде потребно свега неколико поређења. Ако се превише објеката убаци у хеш-табелу, број колизија расте, а перформансе хеш-табеле опадају. Обично се број листи поставља на вредност између 75% и 150% очекиваног броја елемената.
Стандардна библиотека за број листи користи степене двојке, подразумевано 16. (Свака вредност, која се зада за број листи, аутоматски бива заокружена на следећи степен двојке.)
Ако се хеш-табела препуни, неопходно је да буде рехеширана. Да би се табела рехеширала, неопходно је да се креира табела са већим бројем листи, а сви елементи убаце у нову табелу.

Пример 6. Реализовати генеричку класу за уређени пар уз редефинисање метода `equals()` и `hashCode()`. Након тога демонстрирати поређење уређених парова и приказати добијене хеш-кодове. Као основ за класу користити раније уведену генеричку класу `UredjeniPar` из поглавља 13, пример 3. □

```
package rs.math.oop.g14.p06.hesKod;

public class UredjeniParUporediv<T, S> extends UredjeniPar<T, S>{
    public UredjeniParUporediv(T vrednost1, S vrednost2) {
        super(vrednost1, vrednost2);
    }

    @Override
    public boolean equals(Object obj) {
        if(this==obj)
               return true;
        if(obj==null)
               return false;
        if(!(obj instanceof UredjeniParUporediv))
               return false;
        UredjeniParUporediv par =
                       (UredjeniParUporediv)obj;
```

```
        return getVrednost1().equals(par.getVrednost1()) &&
                       getVrednost2().equals(par.getVrednost2());
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 31 * hash
                       + (getVrednost1() == null ? 0 : getVrednost1().hashCode());
        hash = 31 * hash
                       + (getVrednost2() == null ? 0 : getVrednost2().hashCode());
        return hash;
    }

    public static void main(String[] args) {
        UredjeniParUporediv<Integer, Double> par1 =
                       new UredjeniParUporediv<Integer, Double>(30, 6.74);
        UredjeniParUporediv<Integer, Double> par2 =
                       new UredjeniParUporediv<Integer, Double>(30, 6.74);
        UredjeniParUporediv<String, Double> par3 =
                       new UredjeniParUporediv<String, Double>("30", 6.74);
        System.out.println("Поредим "+par1+" и "+par1);
        System.out.println(par1.equals(par1));
        System.out.println("Хеш кодови: "+par1.hashCode()
               +" и "+par1.hashCode());
        System.out.println("Поредим "+par1+" и "+par2);
        System.out.println(par1.equals(par2));
        System.out.println("Хеш кодови: "+par1.hashCode()
               +" и "+par2.hashCode());
        System.out.println("Поредим "+par1+" и "+par3);
        System.out.println(par1.equals(par3));
        System.out.println("Хеш кодови: "+par1.hashCode()
               +" и "+par3.hashCode());
    }
}
```

Приликом поређења два уређена пара, најпре се проверава да ли реферишу на исти објекат. Ако је то испуњено, онда су сигурно исти.

Даље се проверава да ли је прослеђени објекат `null`, и у том случају се враћа вредност `false` пошто актуелни објекат (`this`) сигурно није `null`.

Након тога се испитује да ли је прослеђени објекат одговарајућег типа (`UredjeniParUporediv`). Ако јесте, пореде се обе координате применом метода `equals()`.

Метод `hashCode()` је реализован на стандардни начин – о томе је било речи у секцији 8.5.4.

Испис произведен радом програма је дат испод. Може се приметити да кад-год `equals()` враћа `true`, хеш-кодови су једнаки. Пример у којем су хеш-кодови исти, а објекти различити, није приказан (тешко је осмилити такав сценарио), али се то теоријски може десити.

```
Поредим (30, 6.74) и (30, 6.74)
true
Хеш кодови: -817431779 и -817431779
Поредим (30, 6.74) и (30, 6.74)
true
```

```
Хеш кодови: -817431779 и -817431779
Поредим (30, 6.74) и (30, 6.74)
false
Хеш кодови: -817431779 и -817382210■
```

ЈДК садржи класу `HashSet` која имплементира скуп базиран на хеш табели (енкапсулира хеш табелу). Једина разлика у односу на хеш табелу је у томе што се не дозвољавају дупликати. Ово се разрешава приликом убацивања елемента у хеш табелу – ако је елемент већ унутра, онда се он не убацује у хеш табелу.

`HashSet` је генеричка класа која се може користити за све типове који адекватно редефинишу горепоменуте методе `equals()` и `hashCode()`. Ако методи нису редефинисани, поређење ће бити базирано на једнакости референци, а хеш-кодови ће бити различити за различите референце, без обзира на евентуалну једнакост према садржају (што може изазвати појаву дупликата). Следећи пример демонстрира исправну, али и погрешну употребу класе `HashSet`.

Пример 7. Демонострирати употребу `HashSet` за уграђене типове `Double`, `String`, као и за раније уведене типове `UredjeniPar` и `UredjeniParUporediv`. □

```java
package rs.math.oop.g14.p07.hesSkup;

import java.util.HashSet;

public class RadSaHesSkupom {
    public static void main(String[] args) {
        HashSet<Double> d1HesSkup = new HashSet<>();
        d1HesSkup.add(3.14);
        d1HesSkup.add(1.41);
        d1HesSkup.add(1.0);
        d1HesSkup.add(3.1400); // неће бити додат
        System.out.println("Први скуп реалних:");
        for (Double d : d1HesSkup)
               System.out.print(d + "\t");
        System.out.println();
        System.out.println(d1HesSkup.contains(1.0)); // true
        // contains() прихвата Object као аргумент
        // тако да нема аутоматске конверзије у Double
        System.out.println(d1HesSkup.contains(1)); // false

        HashSet<Double> d2HesSkup = new HashSet<>();
        d2HesSkup.add(4.2);
        d2HesSkup.add(1.00);
        System.out.println("Други скуп реалних:");
        System.out.println(d2HesSkup);

        d1HesSkup.retainAll(d2HesSkup);
        System.out.println("Пресек два скупа реалних:");
        System.out.println(d1HesSkup);

        HashSet<String> sHesSkup = new HashSet<>();
        sHesSkup.add("Скуп");
        sHesSkup.add("ниски");
        sHesSkup.add("елемент");
        sHesSkup.add("елемент");
        System.out.println("Скуп ниски са хеш кодовима:");
        for (String s : sHesSkup)
```

```
                System.out.println(String.format("%s\t%d", s, s.hashCode()));

        HashSet<UredjeniPar<String, Integer>> u1HesSkup
                = new HashSet<UredjeniPar<String, Integer>>();
        u1HesSkup.add(new UredjeniPar<String, Integer>("Новак", 26));
        u1HesSkup.add(new UredjeniPar<String, Integer>("Рафаел", 23));
        u1HesSkup.add(new UredjeniPar<String, Integer>("Роџер", 20));
        u1HesSkup.add(new UredjeniPar<String, Integer>("Новак", 26)); // дупликат
        System.out.println("Први скуп уређених парова:");
        System.out.println(u1HesSkup); // дупликат није елиминисан

        HashSet<UredjeniParUporediv<String, Integer>> u2HesSkup
                = new HashSet<UredjeniParUporediv<String, Integer>>();
        u2HesSkup.add(new UredjeniParUporediv<String, Integer>("Новак", 26));
        u2HesSkup.add(new UredjeniParUporediv<String, Integer>("Рафаел", 23));
        u2HesSkup.add(new UredjeniParUporediv<String, Integer>("Роџер", 20));
        u2HesSkup.add(new UredjeniParUporediv<String, Integer>("Новак", 26));
        System.out.println("Други скуп уређених парова:");
        System.out.println(u2HesSkup); // дупликат је елиминисан
    }
}
```

За типове `Double` и `String` методи `equals()` и `hashCode()` су дефинисани на адекватан начин па су скупови добро формирани (без дупликата). Када је у питању скуп објеката класе `UredjeniPar`, због неадекватне дефиниције поменутих метода, дупликати не бивају елиминисани. Са друге стране скуп објеката класе `UredjeniParUporediv` се понаша очекивано.

Поред метода `add()`, пример приказује употребу метода `contains()`, којим се проверава да ли је елемент у скупу, као и употребу скуповне операције пресека `retainAll()`. Унија и разлика скупова се може реализовати на сличан начин помоћу метода `addAll()` и `removeAll()`. Проблем са овим методима је што су деструктивни — мењају скуп над којим се скуповна операција примењује. Уколико је потребно формирати пресек/унију/разлику уз задржавање оригиналних скупова, најпре је потребно направити копију скупа, применом копирајућег конструктора, и након тога применити одговарајућу операцију.

Следи испис произведен радом програма.

```
Први скуп реалних:
1.41    1.0    3.14
true
false
Други скуп реалних:
[1.0, 4.2]
Пресек два скупа реалних:
[1.0]
Скуп ниски са хеш кодовима:
елемент    32352173
ниски    1035275716
Скуп    32563797
Први скуп уређених парова:
[(Роџер, 20), (Новак, 26), (Рафаел, 23), (Новак, 26)]
Други скуп уређених парова:
[(Новак, 26), (Роџер, 20), (Рафаел, 23)] ■
```

Дрво-скуп, класа TreeSet

Дрво-скуп (класа `TreeSet`), за разлику од хеш-скупа, представља уређену колекцију. То значи да се елементи у дрво-скуп убацују произвољним редоследом, а да се при уносу у скуп елемент смешта на одређено место према неком критеријуму уређења. Претпоставимо да се ниске додају у дрво-скуп у следећем редоследу.

```
SortedSet<String> sortiranSkup= new TreeSet<String>();
sortiranSkup.add("Марко");
sortiranSkup.add("Ана");
sortiranSkup.add("Кристина");
for (String s : sortiranSkup)
       System.out.println(s);
```

Без обзира на редослед уноса, приликом исписа, биће поштовано подразумевано лексикографско растуће уређење над нискама. То значи да ће бити исписана имена у редоследу: Ана, Кристина, Марко.

Инстанца класе `TreeSet<String> je` уопштена интерфејсом `SortedSet<String>`. Ово је могуће јер класа `TreeSet` имплементира интерфејс `SortedSet`. Генерички интерфејс `SortedSet<T>` наслеђује интерфејс `Set<T>` и при том захтева да тип `T` буде упоредив (путем интерфејса `Comparable`) или да се, приликом креирања инстанце дрво-скупа, преко конструктора проследи објекат који имплементира `Comparator` интерфејс (погледати секцију 9.3).

Као што име класе говори, сортирање се извршава по принципу уређене дрвоидне структуре података (бинарно дрво претраге). Сваки пут када се елемент дода у дрво, он се поставља на одговарајућу позицију унутар дрвета. Ако је дрво празно, нови елемент постаје корен. У супротном се нови елемент пореди се кореним елементом и:

- ако су исти, не ради се ништа;
- ако нови елемент претходи кореном елементу по уређењу, нови елемент се додаје рекурзивно у лево под-дрво;
- ако је нови елемент после кореног по уређењу, нови елемент се додаје рекурзивно у десно под-дрво.

Итератор над дрво-скупом је дефинисан тако да посећује елементе у сортираном поретку — прво рекурзивно посећује елементе левог под-дрвета, након тога корени елемент, и на крају рекурзивно елементе десног под-дрвета.

Додавање елемената у дрво има сложеност O(log(n)). На пример, ако дрво садржи 1000 елемената, додавање новог захтева око 10 поређења.

Поставља се питање да ли је за реализацију скупа боље користити дрво или хеш табелу?

Одговор је да то зависи од података који се смештају у колекцију:

- ако нису потребни сортирани подаци, нема разлога да се троши време на сувишно сортирање;
- много важније, неке податке је веома тешко сортирати.

Пример 8. Демонстрирати употребу `TreeSet` класе за уграђене типове: `Double`, `String` као и за раније уведени тип `UredjeniPar` уз употребу `Comparator` интерфејса. □

```
package rs.math.oop.g14.p08.drvoSkup;

import java.util.Comparator;
import java.util.SortedSet;
import java.util.TreeSet;
```

```
import rs.math.oop.g13.p03.genericiUredjeniPar.UredjeniPar;

public class RadSaDrvoSkupom {
	public static void main(String[] args) {
		SortedSet<Double> dDrvoSkup = new TreeSet<>();
		dDrvoSkup.add(3.14);
		dDrvoSkup.add(1.41);
		dDrvoSkup.add(1.0);
		dDrvoSkup.add(3.1400); // неће бити додат
		System.out.println("Скуп реалних: " + dDrvoSkup);

		SortedSet<String> sDrvoSkup = new TreeSet<>();
		sDrvoSkup.add("Скуп");
		sDrvoSkup.add("ниски");
		sDrvoSkup.add("елемент");
		sDrvoSkup.add("елемент");
		System.out.println("Скуп ниски: " + sDrvoSkup);

		SortedSet<UredjeniPar<String, Integer>> uDrvoSkup =
		new TreeSet<UredjeniPar<String, Integer>>(
			new Comparator<UredjeniPar<String, Integer>>() {
				@Override
				public int compare(UredjeniPar<String, Integer> o1,
					UredjeniPar<String, Integer> o2) {
					if (o2.getVrednost2()
							.compareTo(o1.getVrednost2())!=0)
						return o2.getVrednost2()
							.compareTo(o1.getVrednost2());
					else
						return o1.getVrednost1()
							.compareTo(o2.getVrednost1());
					}
				});
		uDrvoSkup.add(new UredjeniPar<String, Integer>("Новак", 26));
		uDrvoSkup.add(new UredjeniPar<String, Integer>("Рафаел", 23));
		uDrvoSkup.add(new UredjeniPar<String, Integer>("Роџер", 20));
		uDrvoSkup.add(new UredjeniPar<String, Integer>("Новак", 26));
		uDrvoSkup.add(new UredjeniPar<String, Integer>("Данил", 4));
		uDrvoSkup.add(new UredjeniPar<String, Integer>("Стефанос", 4));
		System.out.println("Скуп уређених парова: " + uDrvoSkup);
	}
}
```

За потребе дефинисања поретка над уређеним паровима коришћена је анонимна класа која имплементира `Comparator<UredjeniPar<String, Integer>>` интерфејс. Овде није било појаве дупликата, без обзира што је коришћена класа `UredjeniPar` уместо класе `UredjeniParUporediv`. Разлог је то што се задавањем критеријума поређења имплицитно задаје и критеријум поређења на једнакост — метод `compare()` враћа вредност 0 у случају једнакости са неким елементом дрвета, што имплицира да се елемент не додаје у дрво.

Испис произведен радом програма је следећи:

```
Скуп реалних: [1.0, 1.41, 3.14]
Скуп ниски: [Скуп, елемент, ниски]
Скуп уређених парова: [(Новак, 26), (Рафаел, 23), (Роџер, 20), (Данил, 4), (Стефанос, 4)]■
```

Остале имплементације скупова

Постоји још неколико уграђених имплементација скупа у Јави: `LinkedHashSet`, `BitSet`, `EnumSet` и друге.

Класа `LinkedHashSet` је уређена варијанта класе `HashSet`. За реализацију ове класе се користи структура хеш-скупа уз додатак двоструко повезане листе која повезује све елементе у редоследу убацивања елемената у скуп. То значи да је предност `LinkedHashSet` могућност набрајања елемената у истом (или обрнутом) редоследу убацивања. Мана је додатни меморијски трошак, потребан за одржавање двоструко повезане листе.

Пример 9. Упоредити употребу класа `HashSet` и `LinkedHashSet`. □

```
package rs.math.oop.g14.p09.povezaniHesSkup;

import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;

public class RadSaPovezanimHesSkupom {
	public static void main(String[] args) {
		Set<String> hesSkup = new HashSet<>();
		Set<String> povezaniHesSkup = new LinkedHashSet<>();
		String[] elementi = new String[]{
				"Ово", "је", "тачан", "редослед", "убацивања"};
		for(String element: elementi) {
			hesSkup.add(element);
			povezaniHesSkup.add(element);
		}
		System.out.println("Обичан хеш-скуп: "+hesSkup);
		System.out.println("Повезани хеш-скуп: "+povezaniHesSkup);
	}
}
```

Испис произведен радом програма је:

```
Обичан хеш-скуп: [тачан, редослед, убацивања, Ово, је]
Повезани хеш-скуп: [Ово, је, тачан, редослед, убацивања]■
```

`BitSet` је класа која омогућава рад са скупом ненегативних целих бројева. Ова класа не имплементира интерфејс `Set`, али се наводи овде јер суштински представља скуп.

Присуство елемента *(i)* у скупу подразумева постојање бита 1 на позицији *(i)*. С обзиром да је `BitSet` реализован помоћу низа, приступ елементу на случајној позицији је могућ у времену O(1). То значи да се операције убацивања, брисања или провере да ли је елемент у скупу извршавају у O(1) времену.

Унутрашња реализација класе `BitSet` је меморијски штедљива. Она је заснована на низу речи, при чему је реч подразумевано типа `long`, односно дужине 64 бита. Појединачним битовима се приступа помоћу битовских оператора. На пример, за представљање скупа од 750 елемената довољно је користити низ од свега 12 речи. Да би се приступило биту *(i)*, најпре се одређује редни број речи као цео део при дељењу броја *(i)* бројем битова у једној речи. Након тога, остатак при дељењу броја *(i)* бројем битова у речи се користи како би се помоћу битовских оператора приступило

одговарајућој позицији унутар циљне речи. На пример, бит 125 се налази на позицији 61 у оквиру речи на позицији 1.

`BitSet` динамички мења димензију енкапсулираног низа речи у складу са потребама корисника класе.

Пример 10. Демонстрирати употребу класе `BitSet`.□

```
package rs.math.oop.g14.p10.bitovskiSkup;

import java.util.BitSet;

public class RadSaBitovskimSkupom {
	public static void main(String[] args) {
		BitSet bitSkup1 = new BitSet();
		System.out.println("Иницијална величина скупа 1 "
				+bitSkup1.size()); // иницијално 1 реч тј. 64 бита
		bitSkup1.set(56);
		bitSkup1.set(23);
		System.out.println(bitSkup1.get(56)); // true
		System.out.println(bitSkup1.get(13)); // false
		System.out.println(bitSkup1); // елементи скупа {23, 56}
		BitSet bitSkup2 = new BitSet();
		bitSkup2.set(56);
		bitSkup2.set(325); // изазива динамичко повећање низа речи
		System.out.println(bitSkup2.size()); // 6 речи, тј. 384 бита
		System.out.println(bitSkup2.cardinality()); // кардиналност 2
		bitSkup1.and(bitSkup2);
		System.out.println(bitSkup1); // у пресеку је број 56
	}
}
```

Иницијална величина скупа је 64, што одговара једној речи типа `long`. Стога је могуће убацити у скуп елементе 56 и 23. За све остале елементе, који нису експлицитно убачени, сматра се да не припадају скупу, на пример, број 13. Ако је потребно убацити неки елемент који је већи од 63, на пример, број 325, унутрашњи низ се динамички повећава на 6 речи, тј. 384 бита.

Поред метода за рачунање величине скупа, убацивања елемента у скуп и проверу присуства елемента, подржане су и неке стандардне скуповне операције попут пресека, уније, комплемента и слично.

Испис произведен радом програма је следећи:

```
Иницијална величина скупа 1 64
true
false
{23, 56}
384
2
{56}■
```

`EnumSet` је класа слична `BitSet`, с тим што се, уместо рада са ненегативним целим бројевима, овде ради са енумерисаним типовима. Ограничење је да инстанца `EnumSet` може садржати искључиво вредности јединственог енумерисаног типа.

За разлику од `BitSet`, класа `EnumSet` је генеричка и имплементира интерфејс `Set`.

Пример 11. Демонстрирати употребу класе `EnumSet` над раније уведеним енумерисаним типом за представљање дана у недељи `DanUNedelji` (пример 1 у поглављу 12).□

```
package rs.math.oop.g14.p11.enumerisaniSkup;

import java.util.EnumSet;

public class RadSaEnumerisanimSkupom {
	public static void main(String[] args) {
		EnumSet<DanUNedelji> eSkup1 = EnumSet.of(DanUNedelji.CETVRTAK);
		eSkup1.add(DanUNedelji.CETVRTAK);
		eSkup1.add(DanUNedelji.PETAK);
		System.out.println(eSkup1);

		EnumSet<DanUNedelji> eSkup2 = EnumSet.of(DanUNedelji.UTORAK,
				DanUNedelji.CETVRTAK, DanUNedelji.SREDA);
		System.out.println(eSkup2);

		System.out.println(eSkup1.size()); // кардиналност је 2
		eSkup1.addAll(eSkup2);
		System.out.println(eSkup1);
	}
}
```

Као и у случају класе `BitSet`, и овде су подржане стандардне скуповне операције: додавање елемента у скуп, испитивање да ли је елемент у скупу, кардиналност скупа, унија скупова (`addAll()`) итд.
Следи испис произведен радом програма:

```
true
[CETVRTAK, PETAK]
[UTORAK, SREDA, CETVRTAK]
2
[UTORAK, SREDA, CETVRTAK, PETAK]■
```

14.4.3. Редови

О редовима је било речи у секцији 14.1 када су се разматрале две различите имплементације реда за исти ред интерфејс. Поред тога, у секцији 14.3.2 су појашњена два кључна генеричка интерфејса под називима `Queue` и `Deque`. У питању су били редови који се придржавају тзв. FIFO уређења (енг. first in - first out).
У овој секцији ће фокус бити на употреби уграђених имплементација не само FIFO редова, већ и тзв. приоритетног реда. Те имплементације су: `LinkedList`, `ArrayDeque` и `PriorityQueue`. Како је `LinkedList` класи већ посвећена довољна пажња у оквиру овог поглавља, овде је фокус на употреби преостале две имплементације.

Низовни ред са два краја, класа ArrayDeque

`ArrayDeque` имплементира интерфејс `Deque` употребом низа и помоћних индекса који у сваком моменту одржавају информацију о почетку и крају реда. Приликом додавања на крај, индекс краја се увећава. Слично се приликом уклањања са краја тај индекс умањује. По истом принципу функционише и одржавање индекса за почетак реда.
Пример 12. Демонстрирати употребу класе `ArrayDeque` у FIFO и LIFO (енг. last in - first out) режиму.☐

```
package rs.math.oop.g14.p12.nizovniDvostrukiRed;
```

```
import java.util.ArrayDeque;
import java.util.Deque;

public class RadSaNizovnimDvostrukimRedom {
	public static void main(String[] args) {
		Deque<String> red = new ArrayDeque<>();
		red.addLast("Петар");
		red.addLast("Ана");
		red.add("Марко");
		System.out.println(red);
		red.removeFirst();
		red.remove();
		System.out.println(red);
		red.addLast("Милана");
		red.addLast("Драган");
		System.out.println(red);

		Deque<String> stek = new ArrayDeque<>();
		stek.add("main");
		stek.addLast("fakt(4)");
		stek.addLast("fakt(3)");
		stek.addLast("fakt(2)");
		stek.addLast("fakt(1)");
		stek.addLast("fakt(0)");
		System.out.println(stek);
		stek.removeLast();
		stek.removeLast();
		System.out.println(stek);
	}
}
```

Метод `add()`, прописан интерфејсом `Collection`, ради исто што и `addLast()`. Слично, `remove()` ради исто што и `removeFirst()`.
Види се да је различитим комбинацијама ових метода могуће постићи функционалности реда и стека.
Следи испис произведен радом програма.

```
[Петар, Ана, Марко]
[Марко]
[Марко, Милана, Драган]
[main, fakt(4), fakt(3), fakt(2), fakt(1), fakt(0)]
[main, fakt(4), fakt(3), fakt(2)] ■
```

Ред са приоритетом, класа PriorityQueue

Редови са приоритетом враћају елементе у сортираном поретку иако су претходно унесени у произвољном поретку. Прецизније, кад год се позове метод `remove()`, уклања се најмањи елемент реда.
Редови са приоритетом користе елегантну и ефикасну структуру података која се зове гомила (енг. heap).
(Напомена: ову структуру података не треба поистовећивати са хип меморијом, немају никакве сличности, што, на пример, није случај када се о стеку прича из перспективе структуре података и сегмента меморије – ту постоји велика сличност.)
Гомила је самоорганизовано бинарно дрво у којем су операције додавања и уклањања реализоване тако да се након сваке измене дрвета у корену налази најмањи елемент.

При том су померања елемената унутар дрвета, потребна да би се ово својство одржавало, ефикасна и изискују O(log(n)) корака.
Редови са приоритетом могу да чувају елементе класа које су упоредиве. Типично коришћење редова са приоритетом је распоређивање послова:

- сваки посао има свој придружени приоритет;
- послови настају у произвољном временском тренутку и бивају додати у ред;
- посао се може започети у произвољном моменту након свог настанка — посао са највећим приоритетом (тј. елемент са најмањом вредношћу) се тада уклања из реда.

Пример 13. Демонстрирати употребу класе `PriorityQueue` у уређивању редоследа рада процеса. Процес је описан називом и приоритетом.□

```
package rs.math.oop.g14.p13.prioritetniRed;

import java.util.PriorityQueue;
import java.util.Queue;

public class RadSaPrioritetnimRedom {
    public static void main(String[] args) {
        Queue<Proces> procesi =new PriorityQueue<>();
        procesi.add(new Proces("chrome", 10));
        procesi.add(new Proces("cmd", 4));
        procesi.add(new Proces("taskmgr", 1));
        procesi.add(new Proces("explorer", 3));
        System.out.println(procesi);
        System.out.println(procesi.peek()); // на врху је taskmgr
        System.out.println(procesi.poll()); // уклања се taskmgr
        System.out.println(procesi);
        procesi.add(new Proces("Network service", 2)); // нови најбитнији
        System.out.println(procesi);
    }
}
```

Дакле, без обзира што је процес `taskmgr` додат у ред као трећи, он се налази у корену приоритетног реда (тј. први ће бити уклоњен), јер има најзначајнији приоритет – 1. Следи испис произведен радом наведеног програма.

```
[taskmgr:1, explorer:3, cmd:4, chrome:10]
taskmgr:1
taskmgr:1
[explorer:3, chrome:10, cmd:4]
[Network service:2, explorer:3, cmd:4, chrome:10] ■
```

14.5. Речници

Често је потребно на основу тзв. кључне информације неког објекта приступити осталим информацијама тог објекта. На пример, програмер може учитавати информације о особама из датотеке или са конзоле, и чувати их у меморији. Касније, када је потребно приступити информацијама конкретне особе, може се користити идентификатор попут јединственог матичног броја грађанина (ЈМБГ). Остале информације су одређене ЈМБГ-ом особе па оне нису неопходне за идентификацију.
Дакле, може се говорити о пресликавању скупа кључева (скуп ЈМБГ вредности) у колекцију вредности (на пример, колекција имена, презимена, година рођења итд. или

уопштено колекција особа). Структуре података које омогућавају оваква пресликавања се називају речници.

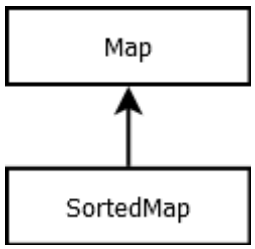
14.5.1. Интерфејс Map

Речник се састоји из три компоненте (подструктуре):

1. скуп кључева,
2. колекција вредности;
3. скуп парова кључ/вредност.

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K,V>> entrySet();
```

Колекције кључева и парова кључ/вредност су увек скупови, јер су кључеви јединствени. Интерфејс `Map.Entry<K, V>` описује уређени пар кључ/вредност и обезбеђује методе који враћају кључ и вредност.
Рад са речницима је дефинисан интерфејсом `Map`, док постоји и интерфејс `SortedMap` који подразумева да је скуп кључева сортиран `SortedSet`.



Интерфејси за речнике

Следи преглед неких значајних метода декларисаних у оквиру интерфејса `Map`.

```
public interface Map<K, V> {
        V get(K kljuc);
        V put(K kljuc, V vrednost);
        V remove(Object kljuc);
        boolean containsKey(Object kljuc);
        boolean containsValue(Object value);
        Set<K> keySet();
        Collection<V> values();
        Set<Map.Entry<K, V>> entrySet();
        ...
}
```

Метод `get()` прихвата кључ и на основу њега враћа придружену вредност. Ако кључ не постоји, враћа се `null`.
Метод `put()` убацује вредност за задати кључ. Ако речник већ садржи вредност за дати кључ, тренутна вредност се замењује новом вредношћу.
Метод `remove()` уклања уређени пар кључ/вредност из речника за задати кључ. При том, ако је успело уклањање, тј. уређени пар је постојао у речнику, враћа се вредност. У супротном се враћа `null`.
Методи `containsKey()` и `containsValue()` враћају `true`/`false` у зависности од тога да ли се задати кључ, односно вредност, налази у скупу кључева, односно колекцији вредности.
Интерфејс `SortedMap` наслеђује `Map` и притом додаје још неколико метода.

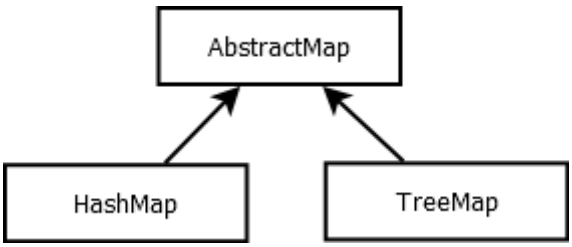
```
public interface SortedMap<K, V> extends Map<K, V> {
```

```
        Comparator<? super K> comparator();
        SortedMap<K, V> subMap(K kljucOd, K kljucDo);
        K firstKey();
        K lastKey();
        ...
}
```

Метод `comparator()` враћа `Comparator` објекат који се користи као основ поређења елемената скупа или `null` уколико се користи природно поређење прописано интерфејсом `Comparable`.
Метод `subMap()` враћа нови речник, добијен из тренутног задржавањем кључева који су по уређењу између граничних кључева (укључујући и границе). Методи `firstKey()` и `lastKey()` враћају први, односно последњи кључ, према дефинисаном уређењу.

14.5.2. Класе за речнике

Јава библиотека подржава две главне имплементације речника: `HashMap` и `TreeMap`. Обе класе индиректно имплементирају `Map` интерфејс тако што наслеђују класу `AbstractMap`. Хеш-речник не сортира кључеве, за разлику од дрвоидног речника који успоставља поредак кључева.
Да ли користити хеш-речник или дрвоидни речник?
Као и са скуповима, хеширање је нешто брже, и то је бољи избор уколико кључеви не морају бити сортирани.



Класе за речнике

Хеш-речник, класа HashMap

Скуп кључева хеш-речника је заснован на хеш-скупу, који је описан у секцији 14.4.2. Будући да је приступ елементима речника заснован на кључевима, временске сложености убацивања, проверавања постојања и уклањања елемената из хеш-речника су исте као код хеш-скупа, O(1).
Пример 14. Демонстрирати употребу класе `HashMap` у одржавању информација о особама. Особа се описује ЈМБГ-ом, именом, презименом и годином рођења. Потребно је омогућити претрагу скупа особа на основу ЈМБГ-а.□

```
package rs.math.oop.g14.p14.hesKatalog;

public class Osoba{
    private String JMBG;
    private String ime;
    private String prezime;
    private int godinaRodjenja;

    public Osoba(String JMBG, String ime, String prezime, int godinaRodjenja) {
```

```
        this.JMBG = JMBG;
        this.ime = ime;
        this.prezime = prezime;
        this.godinaRodjenja = godinaRodjenja;
    }

    public String getJMBG() {
        return JMBG;
    }

    @Override
    public String toString() {
        return String.format("%s\t%s\t%s\t%d", JMBG, ime, prezime, godinaRodjenja);
    }
}
```

```
package rs.math.oop.g14.p14.hesKatalog;

import java.util.HashMap;
import java.util.Map;

public class RadSaHesKatalogom {
    public static void main(String[] args) {
        Osoba[] osobe = new Osoba[] {
                new Osoba("1009987567890", "Марко", "Петровић", 1987),
                new Osoba("2001967567890", "Ана", "Ковачевић", 1967),
                new Osoba("1009997567890", "Марија", "Мирковић", 1997),
                new Osoba("0302000567890", "Јована", "Драшковић", 2000),
                new Osoba("1504981567890", "Петар", "Марковић", 1981),
                new Osoba("1504981567890", "Марко", "Марковић", 1981),
        };
        Map<String, Osoba> osobeKatalog = new HashMap<String, Osoba>();
        for(Osoba o : osobe)
                osobeKatalog.put(o.getJMBG(), o);

        for(String jmbg: osobeKatalog.keySet())
                System.out.println(String.format("%s\t->\t%s",
                                jmbg, osobeKatalog.get(jmbg)));
    }
}
```

Због прегледности, у примеру је унос у речник спроведен на основу низа особа. (Алтернативно су информације о особама могле бити учитане са конзоле, из базе података, датотеке итд.)

За потребе исписа се обично користи итератор над скупом кључева (хеш-скупом) након чега се, на основу кључа, приступа и вредности, у овом случају објекту `Osoba`.

У наведеном примеру постоје две особе са истим ЈМБГ. Иако се ово у пракси јако ретко дешава (не би требало да се дешава уопште), овде је намера била демонстрација замене вредности за дати кључ у случају да је вредност за дати кључ већ у речнику.

Извршавањем програма добија се:

```
1009997567890    ->    1009997567890    Марија    Мирковић    1997
1504981567890    ->    1504981567890    Марко     Марковић    1981
2001967567890    ->    2001967567890    Ана       Ковачевић   1967
1009987567890    ->    1009987567890    Марко     Петровић    1987
```

```
0302000567890    ->    0302000567890    Јована    Драшковић    2000■
```

Дрво-речник, класа TreeMap

Скуп кључева дрво-речника је заснован на дрво-скупу, који је описан у секцији 14.4.2. Слично као и код хеш-речника, и овде су временске сложености свих операција зависне од имплементације скупа кључева. То значи да се операције убацивања, проверавања постојања и уклањања извршавају у сложености O(log(n)), где је n број кључева.

Пример 15. Пребројати појављивања речи у задатом тексту. Након тога исписати на конзоли првих 10 речи и њихових појављивања у складу са лексикографским уређењем речи. За реализацију користити уграђену класу `TreeMap`. Игнорисати знаке интерпункције и величину слова.□

```
package rs.math.oop.g14.p15.drvoKatalog;

import java.util.Map;
import java.util.TreeMap;

public class RadSaDrvoKatalogom {
    public static void main(String[] args) {
        String tekst = "Електрични аутомобили у свету нису више реткост "
                     + "већ редовна појава на улицама. Тај тренд би требало "
                     + "да расте, што показује и недавна најава великог Форда "
                     + "да ће у Европи сви његови аутомобили бити електрични "
                     + "до 2030. године, а сличне изјаве се могу чути и од "
                     + "осталих произвођача возила на батерије. "
                     + "Према резултатима прошлогодишње студије немачког "
                     + "Удружења аутомобилске индустрије, више од 80 одсто "
                     + "испитаних компанија полази од тога да ће се е-возила "
                     + "етаблирати као нови стандард. Исто толико фирми уверено "
                     + "је да је већ започео процес преласка на електричну "
                     + "мобилност. Ипак, 80 процената испитаника рачуна "
                     + "да ће се тек од 2030. године, или чак и касније, "
                     + "догодити потпуна замена, односно доминација е-мотора. "
                     + "Један део испитаних добављача очекује да би могли "
                     + "да се етаблирају и мотори с горивим ћелијама, "
                     + "на синтетичка горива.";
        char[] interpunkcija = new char[] { '.', ',', ';', ':', '?', '!' };
        for (char c : interpunkcija)
               tekst = tekst.replace(c, ' ');
        String[] reciNeprecisceno = tekst.toLowerCase().split("\s+");
        Map<String, Integer> reciPojavljivanja = new TreeMap<>();
        for (String r : reciNeprecisceno)
               if (reciPojavljivanja.containsKey(r))
                      reciPojavljivanja.put(r, reciPojavljivanja.get(r) + 1);
               else
                      reciPojavljivanja.put(r, 1);

        int k = 1;
        for (String r : reciPojavljivanja.keySet())
               if (k > 10)
                      break;
               else {
                      System.out.println(k + ".\t" + r + "\t"
                                                    + reciPojavljivanja.get(r));
```

```
                        k++;
                }
        }
}
```

Приликом убацивања речи у речник, најпре се проверава да ли она већ постоји од раније. Ако не постоји, онда се убацује у речник и придружује јој се вредност 1, што значи да је у питању прво појављивање те речи. У супротном се претходни број појављивања повећава за 1.

У овом примеру се користи природно уређење типа `String` (растуће лексикографско). Ако је потребно дефинисати другачије уређење за тип `String`, на пример опадајуће лексикографски, то се може постићи прослеђивањем објекта анонимне класе типа `Comparator` конструктору класе `TreeMap`.

Извршавањем програма добија се:

```
1.    2030          2
2.    80            2
3.    а             1
4.    аутомобили    2
5.    аутомобилске  1
6.    батерије      1
7.    би            2
8.    бити          1
9.    великог       1
10.   већ           2■
```

14.6. Генерици и колекције

Генерички типови су разматрани у поглављу 13, и тада је било речи о њиховој употреби у контексту низова. Примена и начин рада генеричких типова у контексту колекција је слична као код низова. Разлика је у томе што постоји велики број различитих колекција па су генерички колекцијски интерфејси формирани на начин који омогућава уопштавање.

Налажење максималног елемента низовне листе може се представити следећим делом кода:

```
if(v.size() == 0)
      throw new NoSuchElementException();
T maks = v.get(0);
for (int i = 1; i < v.size(); i++)
      if(maks.compareTo(v.get(i)) < 0)
              maks = v.get(i);
```

Ако је исти алгоритам потребно реализовати над повезаном листом, може се користити итератор, јер употреба приступа на основу индекса није ефикасна код повезане листе:

```
if (l.isEmpty())
      throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T maks = iter.next();
while(iter.hasNext()) {
T sled = iter.next();
       if(maks.compareTo(sled) < 0)
              maks = sled;
}
```

Пожељно је избећи постојање вишеструких дефиниција метода који примењују исти алгоритам над различитим колекцијама:

```
static <T extends Comparable> T max(ArrayList<T> lista);
static <T extends Comparable> T max(LinkedList<T> lista);
...
```

Ту на сцену ступају колекцијски интерфејси.
Израчунавање максимума се, у општем случају, може реализовати итерирањем кроз елементе и евентуалним ажурирањем актуелног максимума у свакој итерацији. Дакле, метод `max()` прихвата објекат ма које класе која имплементира интерфејс `Collection`, јер овај интерфејс обезбеђује постојање итератора:

```
public static <T extends Comparable> T max(Collection<T> c){
        if (c.isEmpty())
             throw new NoSuchElementException();
       Iterator<T> iter = c.iterator();
        T maks = iter.next();
        while (iter.hasNext()) {
               T sled = iter.next();
               if (maks.compareTo(sled) < 0)
                      maks = sled;
        }
        return maks;
}
```

14.6.1. Џокер тип

Знак питања (`?`) је у Јава генеричком програмирању познат под називом џокер тип (енг. wildcard). Постоје два џокер типа:

1. џокер тип горње границе (енг. upper bounded wildcard) и
2. џокер тип доње границе (енг. lower bounded wildcard).

Џокер тип горње границе се користи када је потребно релаксирати ограничење над генеричким типом. На пример, ако је потребно дефинисати метод за преписивање бројева из `List<Integer>` у `List<Integer>` или из `List<Double>` у `List<Double>`, онда би декларација метода могла да изгледа овако:

```
static void prepisi(List<? extends Number> ciljna, List<? extends Number> izvorna);
```

Дакле, џокер се користи у комбинацији са кључном речи `extends` како би се ограничио са горње стране класом `Number`.
Јасно је да се слична функционалност може постићи и раније уведеним генеричким типовима из поглавља 13. На пример, могао би се креирати самостални генерички метод са следећим потписом:

```
static <T extends Number> void prepisi(List<T> ciljna, List<T> izvorna);
```

Штавише, употребом генеричких типова се може постићи ограничавање одозго са вишеструким типовима, од којих је највише један класа и неограничен број интерфејса (секција 13.4).
Међутим, уочава се и разлика између употребе генеричког и џокер типа у овом примеру. Наиме, применом џокер горњег ограничења могуће је преписати низ целих бројева у низ реалних и обратно – што није дозвољено претходном дефиницијом метода који користи генерички тип. Али, уз мало другачију дефиницију самосталног генеричког метода, могуће је постићи и такво понашање:

```
static <T1 extends Number, T2 extends Number> void prepisi(List<T1> ciljna,  List<T2>
izvorna);
```

Дакле, једина суштинска предност џокер типа у односу на генерички је могућност успостављања и доњег и горњег ограничења – код генеричког типа је могуће само горње.

Пример 16. Дефинисати и тестирати метод за сумирање листе бројева при чему елементи листе могу бити цели или реални бројеви.☐

```
package rs.math.oop.g14.p16.dzokerTipGornjeGranice;

import java.util.Arrays;
import java.util.List;

class SumaBrojeva {
    public static void main(String[] args) {
        List<Integer> lista1 = Arrays.asList(24, 56, 61, 7);
        System.out.println("Сума целих бројева је: " + suma(lista1));
        List<Double> lista2 = Arrays.asList(14.1, 52.1, 6.121);
        System.out.print("Сума реалних бројева је: " + suma(lista2));
    }

    private static double suma(List<? extends Number> lista) {
        double suma = 0.0;
        for (Number i : lista)
                suma += i.doubleValue();
        return suma;
    }
}
```

С обзиром да је апстрактна класа `Number` наткласа класа `Integer`, `Float`, `Double`, `Short` итд., конструкцијом `<? extends Number>` се успоставља горња граница (уопштење) над захтеваним типом података.

Испис произведен радом програма је следећи:

```
Сума целих бројева је: 148.0
Сума реалних бројева је: 72.321 ■
```

Џокер тип доње границе се користи у ситуацијама када би употреба конкретног генеричког типа била превише рестриктивна. Синтакса је слична као код горње границе, с тим што се, уместо кључне речи `extends`, користи `super`. Мотивација за примену џокер типа доње границе ће бити најбоље демонстрирана примером.

Пример 17. Дефинисати класу `OsobaUporediva` као поткласу `Osoba` из примера 14. Ова класа треба да имплементира интерфејс `Comparable<OsobaUporediva>`. Поређење треба да буде засновано на вредности ЈМБГ. Такође, дефинисати класу `Student` као поткласу класе `OsobaUporediva`. Студент, поред тога што је особа, има и број индекса и просечну оцену, а начин поређења је исти као код особе. Испитати употребу уграђеног метода за сортирање `Collections.sort()` над листом студената. Да ли класа `Student` може да имплементира интерфејс `Comparable<Student>`? Како се постиже да метод за сортирање ради над листом студената? Који је потпис овог метода и која ограничења постоје на џокер типу који се овде користи? ☐

```
package rs.math.oop.g14.p17.dzokerTipDonjeGranice;

public class OsobaUporediva extends Osoba implements Comparable<OsobaUporediva>{
```

```
    public OsobaUporediva(String JMBG, String ime, String prezime,
                int godinaRodjenja) {
        super(JMBG, ime, prezime, godinaRodjenja);
    }

    @Override
    public int compareTo(OsobaUporediva o) {
        return this.getJMBG().compareTo(o.getJMBG());
    }
}
```

```
package rs.math.oop.g14.p17.dzokerTipDonjeGranice;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Student extends OsobaUporediva{

    private String indeks;
    private double prosecnaOcena;

    public Student(String jMBG, String ime, String prezime, int godinaRodjenja,
                String indeks, double prosecnaOcena) {
        super(jMBG, ime, prezime, godinaRodjenja);
        this.indeks = indeks;
        this.prosecnaOcena = prosecnaOcena;
    }

    @Override
    public String toString() {
        return super.toString()+"\t"+indeks+"\t"+prosecnaOcena;
    }

    public static void main(String[] args) {
        List<Student> studenti = new ArrayList<>();
        studenti.add(new Student(
                    "1009987567890", "Марко", "Петровић", 1987, "23", 9.33));
        studenti.add(new Student(
                    "2001967567890", "Ана", "Ковачевић", 1967, "13", 8.43));
        studenti.add(new Student(
                    "1009997567890", "Марија", "Мирковић", 1997, "111", 9.36));
        Collections.sort(studenti);
        for(Student s: studenti)
               System.out.println(s);
    }
}
```

Класа `Student` не може да имплементира интерфејс `Comparable<Student>`, јер би у том случају класа `Student` била подтип два различита генеричка типа `Comparable<OsobaUporediva>` и `Comparable<Student>`, што није дозвољено. Имајући то у виду, метод `Collections.sort()` је декларисан на следећи начин:

```
public static <T extends Comparable<? super T>> void sort(List<T> list);
```

уместо на начин који се могао очекивати:

```
public static <T extends Comparable<T>> void sort(List<T> list);
```

Разлог је управо флексибилност метода у раду са изведеним генеричким типовима. На пример, библиотечка имплементација сортирања `Collections.sort()` сада може да се примењује не само над генеричким колекцијама упоредивих типова података `T`, већ и над генеричким колекцијама типова података `S` који су изведени из `T`, а који при том немају потребу да мењају начин поређења.
У случају да изведени генерички тип `S` треба да користи другачији начин поређења у односу на тип `T`, може се креирати одговарајући `Comparator` објекат за циљни генерички тип, на пример `Comparator<Student>`.
Испис произведен радом програма је:

```
1009987567890    Марко    Петровић    1987    23    9.33
1009997567890    Марија    Мирковић    1997    111    9.36
2001967567890    Ана    Ковачевић    1967    13    8.43 ■
```

14.6.2. Генерички колекцијски методи имплементирани у JDK

Генерици су моћан концепт који се користи у сортирању, бинарној претрази и још неким корисним алгоритмима.

Сортирање колекције

Како метод `Collections.sort()` сортира произвољну колекцију? Алгоритми за сортирање се у литератури обично дефинишу за низове. Кључна погодност низова је брз случајни приступ елементима. Међутим, колекција, у општем случају, нема случајан приступ. Имплементација у Јави једноставно копира све елементе колекције у низ (применом итератора), сортира га применом метода `Arrays.sort()`, а затим копира сортирану секвенцу натраг у колекцију. О методу `Arrays.sort()` је било речи у секцији 7.6.

Мешање колекције

Класа `Collections` поседује метод `shuffle()` који случајно пермутује (меша) елементе. На пример, следећи кôд омогућава генерисање случајне пермутације величине 10.

```
List<Integer> permutacija = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
Collections.shuffle(permutacija, new Random(12345));
System.out.println(permutacija); // [4, 3, 1, 6, 9, 10, 7, 8, 5, 2]
```

Мешање је могуће извршити над било којим објектима, а не само над бројевима.

Бинарна претрага

Метод `Collections.binarySearch()` имплементира бинарну претрагу над колекцијом. Идеја алгоритма је иста као и у случају низова (секција 7.6). Дакле, колекција мора претходно бити сортирана, у супротном ће алгоритам потенцијално вратити погрешан резултат. Следе примери позивања овог метода:

```
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, komparator);
```

Са `c` је означена колекција у којој се претражује елемент `element`. У другом позиву се, уместо природног уређења дефинисаног имплементацијом интерфејса `Comparable`, користи ad-hoc уређење дефинисано `komparator` објектом типа `Comparator`. Бинарна претрага захтева од колекције могућност брзог случајног приступа. Стога метод `binarySearch()` проверава да ли задата колекција имплементира интерфејс `RandomAccess`, објашњен у секцији 14.4.
Ако да, онда врши бинарну претрагу, у супротном врши линеарну претрагу.

Преглед значајнијих метода услужне класе Collections

У наставку је дат преглед још неких значајнијих метода класе `Collections`, са кратким објашњењима (коментарима).

```
// Враћа компаратор који сортира елементе у обрнутом поретку
// од поретка одређеног методом compareTo() интерфејса Comparable.
static <T> Comparator<T> reverseOrder();

// Враћа компаратор који сортира елементе у обрнутом поретку
// од поретка одређеног компаратором komp.
static <T> Comparator<T> reverseOrder(Comparator<T> komp);

// Минимум колекције у складу са природним уређењем.
static <T extends Comparable<? super T>> T min(Collection<T> elementi);

// Максимум колекције у складу са природним уређењем.
static <T extends Comparable<? super T>> T max(Collection<T> elementi);

// Минимум колекције у складу са уређењем датим компаратором komp.
static <T> min(Collection<T> elementi, Comparator<? super T> komp);

// Максимум колекције у складу са уређењем датим компаратором komp.
static <T> max(Collection<T> elementi, Comparator<? super T> komp);

// Копирање елемената из изворне листе на исте позиције у циљној листи.
static <T> void copy(List<? super T> ciljna, List<T> izvorna);

// Попуњавање свих позиције у листи задатом вредношћу.
static <T> void fill(List<? super T> l, T vrednost);

// Ажурирање вредности елемената.
static <T> boolean replaceAll(List<T> l, T staraVrednost, T novaVrednost);

// Тражење подлисте l једнаке s или -1 ако нема такве подлисте у l.
static int indexOfSubList(List<?> l, List<?> s);

// Размењивање елемената на датим позицијама.
static void swap(List<?> l, int i, int j);

// Обртање листе.
static void reverse(List<?> l);

// Циклично померање елемената листе за d позиција.
static void rotate(List<?> l, int d);

// Враћање броја елемената колекције који су једнаки датом објекту o.
```

```
static int frequency(Collection<?> c, Object o);

// Испитивање да ли су колекције дисјунктне.
boolean disjoint(Collection<?> c1, Collection<?> c2);
…
```

14.7. Апстрактне класе као основа за колекције

Када се прегледа API документација, уочава се да постоји још један скуп класа, чије име почиње са `Abstract` (на пример класа `AbstractSet`). Хијерархија апстрактних колекцијских класа је усаглашена са хијерархијом интерфејса који су претходно уведени. На пример, интерфејс `Collection` има свој пандан `AbstractCollection`, за `Set` постоји `AbstractSet` итд. Свака од апстрактних класа имплементира одговарајући интерфејс, па се намеће питање зашто оне уопште постоје?

Одговор је да ове апстрактне класе поседују имплементације за неке од метода који су прописани интерфејсима, што надаље омогућава програмеру да, приликом писања својих конкретних класа, не мора да имплементира све методе прописане интерфејсима (којих је обично доста).

Како је могуће имплементирати метод апстрактне класе „унапред“, тј. пре него што је позната унутрашња структура конкретних колекцијских класа? (На пример, у случају интерфејса `List` могуће је користити низ, али и повезану листу за имплементацију.)

Одговор је да неки методи не зависе од унутрашњих структура података, јер се ослањају на друге методе, који могу, а не морају бити имплементирани на нивоу апстрактне класе. Наредни пример демонстрира како се метод `containsAll()`, који је прописан интерфејсом `Collection`, може реализовати посредно позивањем метода `contains()` и итератора на којем је заснована колекцијска наредба `for`. Ово ће бити могуће, без обзира што ће имплементације метода `contains()` и итератора бити познате тек у будућности, када се буде реализовала конкретна класа.

```
public abstract class AbstractCollection<T>{
   ...
   boolean containsAll(Collection<?> c){
      for (Object o: c)
         if (!contains(o))
             return false;
      return true;
   }
   ...
}
```

Ако програмер жели директно да имплементира интерфејс `Collection`, потребно је да имплементира 13 метода. Ако би се одлучио да, уместо тога, наследи класу `AbstractCollection`, овај број се смањује на свега 2 метода: `iterator()` и `size()`. То значи да је и метод `contains()`, из претходног примера, посредно реализован преко најосновнијих метода (`iterator()` и `equals()`).

```
public abstract class AbstractCollection<T>{
   ...
   boolean contains(Object o){
      Iterator<T> it = iterator();
      if (o == null){
```

```
            while (it.hasNext())
               if (it.next() == null)
                   return true;
        } else {
            while (it.hasNext())
               if (o.equals(it.next()))
                   return true;
        }
        return false;
    }
    ...
}
```

Пример 18. У примеру 4 (секција 13.2.1) је дефинисан генерички интерфејс за стек и две имплементације овог интерфејса: помоћу саморастућег низа и повезане листе. Сада је потребно реализовати исти интерфејс и класе, са том разликом да генерички интерфејс `Stek<T>` проширује (наслеђује) основни колекцијски интерфејс `Collection<T>`. Ово ће омогућити бољу спрегнутост између будућих имплементација интерфејса `Stek<T>` и великог броја механизама (услужних библиотека) које раде са колекцијама.
(Напомена: Да би се једноставније достигле функционалности које прописује `Collection<T>` интерфејс, класе које имплементирају `Stek<T>` могу да наследе класе `AbstractCollection<T>`.)□

```
package rs.math.oop.g14.p18.stekKolekcijaListaNiz;

import java.util.Collection;

public interface Stek<T> extends Collection<T>
{
    void dodaj(T elem);
    T ukloni();
}
```

Овде стек интерфејс сада не дефинише метод `brojElemenata()`, као што је то раније био случај. Разлог је то што у интерфејсу `Collection<T>` већ постоји одговарајући метод за ту намену и зове се `size()`.

```
package rs.math.oop.g14.p18.stekKolekcijaListaNiz;

import java.util.AbstractCollection;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class StekPrekoNiza<T> extends AbstractCollection<T> implements Stek<T> {
    private SamorastuciNiz<T> elementi;
    private int vrhSteka;

    {
        elementi = new SamorastuciNiz();
        vrhSteka = -1;
    }

    @Override
    public void dodaj(T elem) {
```

```
        elementi.postaviNa(++vrhSteka, elem);
    }

    @Override
    public T ukloni() {
        if (vrhSteka == -1) {
                System.err.println("Грешка при уклањању: стек је празан!");
                return null;
        }
        return elementi.uzmiSa(vrhSteka--);
    }

    @Override
    public boolean add(T elem) {
        elementi.postaviNa(++vrhSteka, elem);
        return true;
    }

    @Override
    public Iterator<T> iterator() {
        return new StekIterator();
    }

    @Override
    public int size() {
        return (vrhSteka + 1);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("<<");
        for (T elem : this)
                sb.append(elem + " ");
        sb.append(">>");
        return sb.toString();
    }

    private class StekIterator implements Iterator<T> {
        private int pozicija;

        public StekIterator() {
                pozicija = elementi.brojElemenata() - 1;
        }

        public T next() {
                if (!hasNext())
                        throw new NoSuchElementException();
                T r = (T) elementi.uzmiSa(pozicija);
                pozicija--;
                return r;
        }

        public boolean hasNext() {
                return pozicija >= 0;
        }
```

```
        public void remove() {
                throw new UnsupportedOperationException();
        }
    }
}
```

Реализација класе `StekPrekoNiza<T>` је, слично као и раније, директно заснована на употреби класе `SamorastuciNiz<T>` (њена имплементација није приказана, јер је иста као у примеру 4 из секције 13.2.1).

С обзиром да класа `Stek<T>` имплементира индиректно интерфејс `Collection<T>` и при том наслеђује класу `AbstractCollection<T>`, неопходно је реализовати само методе који се налазе у разлици скупова метода интерфејса `Collection<T>` и метода који су реализовани у класи `AbstractCollection<T>`. Ту разлику чине методи: `add()`, `size()` и `iterator()`. Јасно је да се `add()` и `size()` директно ослањају на позивање метода из класе `SamorastuciNiz<T>`. Са друге стране, за потребе реализације метода `iterator()` се дефинише унутрашња класа под називом `StekIterator<T>` која имплементира интерфејс `Iterator<T>`.

```
package rs.math.oop.g14.p18.stekKolekcijaListaNiz;

import java.util.*;

public class StekPrekoListe<T> extends AbstractCollection<T> implements Stek<T> {

    private class Cvor {
        private Cvor sledeci;
        private T vrednost;

        public Cvor(T item) {
                this.vrednost = item;
                sledeci = null;
        }

        public String toString() {
                return "елеменат листе: " + vrednost;
        }
    }

    private Cvor glava = null;

    private void dodajNaPocetak(T elem) {
        if (glava == null) {
                glava = new Cvor(elem);
                return;
        }
        Cvor e = new Cvor(elem);
        e.sledeci = glava;
        glava = e;
    }

    @Override
    public boolean add(T elem) {
        dodajNaPocetak(elem);
        return true;
    }
```

```
private T ukloniSaPocetka() throws Exception {
    if (glava == null) {
            throw new Exception("Колекција је празна");
    }
    Cvor e = glava;
    glava = e.sledeci;
    return e.vrednost;
}

public StekPrekoListe() {
}

public StekPrekoListe(Collection<T> c) {
    Iterator<T> it = c.iterator();
    while (it.hasNext()) {
            dodajNaPocetak(it.next());
    }
}

@Override
public Iterator<T> iterator() {
    return new IteratorSteka<T>();
}

@Override
public int size() {
    Iterator<T> it = this.iterator();
    int ret = 0;
    while (it.hasNext()) {
            ret++;
            it.next();
    }
    return ret;
}

@Override
public void dodaj(T elem) {
    dodajNaPocetak(elem);
}

@Override
public T ukloni() {
    T ret = null;
    try {
            ret = ukloniSaPocetka();
    } catch (Exception e) {
            System.out.println(e);
    }
    return ret;
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder("<<");
    for (T elem : this)
```

```
                sb.append(elem + " ");
        sb.append(">>");
        return sb.toString();
    }

    private class IteratorSteka<E> implements Iterator<E> {
        private Cvor tekuci;

        public IteratorSteka() {
                tekuci = glava;
        }

        @Override
        public boolean hasNext() {
                return tekuci != null;
        }

        @Override
        public E next() {
                E vrednost = (E) tekuci.vrednost;
                tekuci = tekuci.sledeci;
                return vrednost;
        }
    }
}
```

Слична је ситуација и код класе `StekPrekoListe<T>`, с тим да је реализација зависне класе `PovezanaLista<T>` овде интегрисана у кôд класе `StekPrekoListe<T>`. Разлог је то што унутрашња класа `StekIterator<T>`, која итерира над повезаном листом, захтева приступ глави (почетном чвору) листе референци на следећи чвор.

```
package rs.math.oop.g14.p18.stekKolekcijaListaNiz;

import java.util.*;

public class StekoviKolekcije {
    private static Stek<String> kreiraj(Scanner sc) {
        Stek stek = null;
        System.out.print("Унеси л (за листу) или н (за низ): ");
        char ulaz = sc.nextLine().trim().toLowerCase().charAt(0);
        switch (ulaz) {
        case 'л':
        case 'l':
                stek = new StekPrekoListe();
                break;
        case 'н':
        case 'n':
                stek = new StekPrekoNiza();
        }
        return stek;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Stek<String> stek = kreiraj(sc);
        sc.close();
```

```
        System.out.println("Хеш скуп");
        Set<String> skup = new HashSet<String>();
        skup.add("Александар");
        skup.add("Бранко");
        skup.add("Владимир");
        skup.add("Горан");
        skup.add("Душан");
        skup.add("Ђорђе");
        System.out.println(skup);

        System.out.println("Стек");
        stek.addAll(skup);
        for (String s : stek)
               System.out.print(s + " ");
        System.out.println("У стек се додаје 'Емил'");
        stek.dodaj("Емил");
        System.out.println(stek);
        System.out.println("Величина стека: " + stek.size());

        System.out.println("Низовна листа");
        List<String> lista = new ArrayList<>(stek);
        lista.addAll(stek);
        System.out.println(lista);

    }
}
```

Класа `StekoviKolekcije` демонстрира спрегнутост различитих типова колекција, јер све имплементирају интерфејс `Collection<T>`. Најпре се креира празан стек заснован на листи или саморастућем низу. Након тога се креира скуп у који се убацују властита имена. После тога се сви елементи тог скупа додају на стек. На крају се креира листа (библиотечка имплементација) у коју се уписују сви елементи стека.
Примери извршавања су дати у наставку.

```
Унеси л (за листу) или н (за низ): н
Хеш скуп
[Бранко, Горан, Александар, Душан, Владимир, Ђорђе]
Стек
Ђорђе Владимир Душан Александар Горан Бранко У стек се додаје 'Емил'
<<Емил Ђорђе Владимир Душан Александар Горан Бранко >>
Величина стека: 7
Низовна листа
[Емил, Ђорђе, Владимир, Душан, Александар, Горан, Бранко, Емил, Ђорђе, Владимир, Душан,
Александар, Горан, Бранко]

Унеси л (за листу) или н (за низ): л
Хеш скуп
[Бранко, Горан, Александар, Душан, Владимир, Ђорђе]
Стек
Ђорђе Владимир Душан Александар Горан Бранко У стек се додаје 'Емил'
<<Емил Ђорђе Владимир Душан Александар Горан Бранко >>
Величина стека: 7
Низовна листа
[Емил, Ђорђе, Владимир, Душан, Александар, Горан, Бранко, Емил, Ђорђе, Владимир, Душан,
Александар, Горан, Бранко] ■
```

14.8. Резиме

Поред познавања начина употребе, разумевање начина рада и познавање временске сложености колекција и речника је од великог значаја за програмера: 1) он може да направи прави избор у складу са потребама програма; 2) не мора да реализује структуре података од нуле, па је убрзан развој програма; 3) анализом организације интерфејса и увидом у начин имплементације класа програмер може да усвоји нове принципе ООП; 4) може да користи придружене (обично генеричке) алгоритме итд.
У овом поглављу се могао осетити значај раздвајања интерфејса од имплементације као и пажљивог планирања хијерархије интерфејса и класа. Такође се показало да су генерички типови неизоставан концепт када је у питању писање проширive, прегледне и интуитивне подршке за рад са колекцијама и речницима.

14.9. Питања и задаци

1. Које су разлике, а које су сличности између колекције и речника? Како се речник може представити употребом колекција?
2. Описати интерфејс `Collection` и навести примере употребе.
3. Објаснити везу између интерфејса `Iterable` и интерфејса `Iterator`.
4. Написати Јава програм који из колекције уклања све елементе који задовољавају дати услов. На пример, ако је у питању колекција целих бројева уклонити све парне бројеве или све бројеве који се завршавају датом цифром.
5. Примерима илустровати ситуације када је корисно користити колекцију:
    - листа;
    - ред;
    - скуп.
6. Шта су апстрактне колекцијске класе и која је њихова предност у односу на одговарајући интерфејс?
7. Објаснити хијерархију колекцијских класа у Јави.
8. Упоредити две имплементације `List` интерфејса – `LinkedList` и `ArrayList`. Примером илустровати ситуације када је погодније користи једну, а када другу имплементацију.
9. Како је имплементирана и како се користи класа `HashSet`? Зашто је битно да методи `hashCode()` и `equals()` буду конзистентно (ре)дефинисани?
10. Како је имплементирана и како се користи класа `TreeSet`? Примером илустровати ситуације када је погодније користи `HashSet` имплементацију, а када `TreeSet` имплементацију скупа.
11. Које још уграђене имплементације скупа постоје у програмском језику Јава? Да ли све имплементирају интерфејс `Set`?
12. Користећи класу `PriorityQueue` имплементирати алгоритам за сортирање који ради по принципу `heap sort-a`.
13. Шта је џокер тип и када се користи? Примерима илустровати употребу џокер типа горње границе и џокер типа доње границе.
14. Објаснити како се врши сортирање колекције употребом метода `Collections.sort()`.

15. Карта је описана знаком (пик, каро, херц, треф) и вредношћу (ас, краљ, дама, жандар и бројчане вредности од 2 до 10). Написати Јава програм који генерише, а затим меша стандардни шпил од 52 и исписује их на екрану.
16. Примерима илустровати употребу неких значајнијих метода класе `Collections`.

15. Улаз и излаз

Један од фундаменталних аспеката програмског језика је учитавање улазних података у програм и приказивање или складиштење излазних података произведених радом програма. Објектно оријентисани програмски језици теже ка омогућавању униформне, вишеструко искористиве (рејузабилне) и прошириве функционалности за рад са улазом и излазом.

У програмском језику C методи за форматирани улаз са конзоле, датотеке и ниске се реализују засебно помоћу функција: `scanf()`, `fscanf()` и `sscanf()`. Са друге стране, у Јави се полази од интерфејса и апстрактних класа које прописују методе за читање са апстрактног улаза, а након тога се у конкретним изведеним класама реализују методи за различите типове улаза попут: конзоле, датотеке, ниске, мрежног порта и слично.

У Јави постоје две библиотеке за рад са улазом и излазом. Прва је Јава IO библиотека која постоји још од самог почетка Јаве. Друга је новијег датума (почев од верзије 1.4 и касније ревидирана у 1.7) и зове се Јава NIO (енг. non-blocking IO).

Јава IO библиотека (пакет `java.io`) је заснована на употреби тзв. токова података. Ток података омогућава секвенцијални приступ бајтовима (или карактерима). Ток је увек једносмеран (симплекс), тј. постоји извор података (енг. data source) из којег се помоћу улазног тока података доводе подаци у Јава програм. Потом се, помоћу одлазног тока података, подаци из Јава програма одводе ка циљу (енг. data sink). За реализацију двосмерне комуникације (пуни дуплекс) потребно је направити додатну независну једносмерну комуникацију у другом смеру (од циља ка извору). Приликом читања или писања по току, Јава не памти (не кешира) податке. То значи да се програмер мора сам бринути о вишеструким употребама података из тока.

Јава IO реализује своје основне функционалности кроз употребу блокирајућих метода за читање и писање (`read()` и `write()`). Блокираност подразумева да нит која приступа подацима помоћу ових метода остаје неактивна све док подаци нису доступни, на пример, ако оперативни систем не може одмах да достави садржај датотеке.

Јава NIO библиотека (пакет `java.nio`) користи другачији приступ када је у питању рад са улазним и излазним подацима. Наиме, подаци се смештају у бафере који служе као посредници у комуникацији између извора/циља и Јава програма, док се између њих налазе тзв. канали (енг. channel). Употреба канала омогућава нити, која захтева податке од канала, да ради нешто друго док канал покушава да приступи траженим подацима. Из тог разлога се Јава NIO библиотека зове неблокирајућа.

Употреба бафера има и своје мане — потребно је проверавати да ли бафер садржи све податке неопходне за неку обраду, и ако не садржи, онда је потребно учитати у бафер још података. Може се десити да се учитавањем тих додатних података „прегазе“ подаци који још нису обрађени — због фиксиране величине бафера.

С обзиром да је употребна вредност неблокирајуће библиотеке за рад са улазом и излазом већа у контексту вишенитног и асинхроног програмирања, које није у фокусу ове књиге, у даљем тексту биће обрађена библиотека Јава IO.

15.1. Блокирајући улаз и излаз — java.io

Улаз и излаз у Јава IO су реализовани преко токова података. Основу токова чине две апстрактне класе: `InputStream` и `OutputStream`. Улаз и излаз се, у овом случају, организују преко тока бајтова. Поступа се тако што се креира ток који ће, приликом позива конструктора бити придружен датотеци, конзоли или мрежном порту, а улазно/излазне операције се реализују позивима одговарајућих метода над тако креираним током.

Скоро сви улазно-излазни методи могу генерисати изузетке, па они обично у декларацији садрже `throws IOException`.

Поред токова података, за улаз и излаз се још користе читачи и писачи (варијанте токова). Они су прилагођени читању карактера, док су `InputStream` и `OutputStream` намењени читању бинарних садржаја. Основу чине две апстрактне класе: `Reader` и `Writer`. Као и код токова, поступак је да се креира читач/писач који се потом придружује датотеци, конзоли, мрежном порту итд. Улазно/излазне операције се такође реализују позивима одговарајућих метода над креираним читачем/писачем.

Постоји неколико фундаменталних интерфејса које токови података и читачи/писачи имплементирају. Преглед имплементираних интерфејса је следећи:

- `InputStream` имплементира `Closeable` и `AutoCloseable`.
- `OutputStream` имплементира `Closeable`, `AutoCloseable` и `Flushable`.
- `Reader` имплементира интерфејсе `Closeable`, `AutoCloseable` и `Readable`.
- `Writer` имплементира `Closeable`, `AutoCloseable`, `Flushable` и `Appendable`.

Интерфејс `Closeable` декларише метод `close()` чијим позивом се ослобађа ток и сви њему придружени ресурси.

`AutoCloseable` такође подразумева метод `close()`, с тим што се овај метод позива аутоматски (уз употребу `try-with` наредбе која није обрађена у овој књизи). `Flushable` декларише метод `flush()` којим се форсира тренутно писање у излазни ток података (понекад је писање одложено због побољшања ефикасности). `Appendable` и `Readable` интерфејси, редом, декларишу методе `append()` и `read()` за надовезивање садржаја на ток, односно, читање са истог.

15.1.1. Улазни и излазни токови података

Како су `InputStream` и `OutputStream` апстрактне класе, то се улаз/излаз реализује преко њихових потокласа, као што су:

- `FileInputStream` и `FileOutputStream`;
- `DataInputStream` и `DataOutputStream`.

Раније коришћени токови података `System.in` и `System.out` представљају примерке потокласа `InputStream` и `OutputStream`. Приметимо да су ови токови дефинисани као статичка поља у оквиру класе `System` — они представљају унапред припремљене (иницијализоване) токове за двосмерну интеракцију са стандардним улазом и излазом.

„Стандардни“ излазни ток `System.out` је отворен након покретања главног програма и спреман за прихватање података. Обично овај ток одговара излазу конзоле или другом одредишту за излаз који је одредило окружење домаћина.

„Стандардни” улазни ток `System.in` је већ отворен и спреман за прихват улазних података. Обично овај ток одговара улазу са тастатуре или неком другом улазном извору који је корисник прогласио као стандардни преко Јава окружења.

Улазни ток података, InputStream

Основни метод у класи `InputStream` је `read()`. Он чита један бајт (број 0-255, остатак опсега `int` се игнорише). Ако се при читању препозна крај тока, метод враћа -1. Потпис овог метода је:

```
public abstract int read() throws IOExeption;
```

Методи за читање низа бајтова се реализују вишеструким позивима горенаведеног апстрактног метода (па они нису апстрактни):

```
public int read(byte b[]) throws IOException
public int read(byte b[], int pocetak, int duzina) throws IOException
```

Улазне операције реализоване преко класе `InputStream` су операције тзв. ниског нивоа. Рад на том нивоу није атрактиван, нити ефикасан (са тачке гледишта типичног програмера) па је развијен велики број поткласа за организацију улаза „специјалних“ врста података, на пример, ниски, бројева, датума итд. Неке од ових поткласа су: `FileInputStream`, `FilterInputStream`, `ByteArrayInputStream`, `ObjectInputStream` итд.

Метод `close()` затвара улазни ток података и ослобађа све системске ресурсе додељене току.

```
public void close() throws IOException;
```

Хијерархија изведена из апстрактне класе `InputStream` је врло разграната.

- `AudioInputStream` — за представљање различитих аудио формата различитих дужина.
- `ByteArrayInputStream` — садржи унутрашњи бафер који похрањује унапред наредне бајтове спремне за читање.
- `FileInputStream` — ток специјализован за читање бинарног садржаја из датотеке.
- `FilterInputStream` — користи неки други улазни ток као своју основу и над њим врши трансформације тока података у ходу. Постоји велики број поткласа ове класе за различите намене, а само неке од њих су:
  - `BufferedInputStream` — користи интерни бафер како би операције читања од стране Јава програма биле ефикасније извршаване и како би програм имао могућност накнадног консултовања прећашњег садржаја (у зависности од величине бафера).
  - `CipherInputStream` — користи се обично у читању шифрованог садржаја тако што у ходу чита основни ток и дешифрује га.
  - `DataInputStream` — омогућава читање Јава примитивних типова.
- `ObjectInputStream` — омогућава читање објеката који су раније сачувани у неком извору (на пример датотеци, бази података и слично). Корисно за чување стања програма и накнадно учитавање тог стања, на пример, у видео играма.
- `PipedInputStream` — овај улазни ток се повезује са својим парњаком `PipedOutputStream` и надаље омогућава читање онога што се пише по `PipedOutputStream`. Обично се користи за комуникацију између више нити извршавања.
- `SequenceInputStream` — омотач око секвенце других улазних токова. Функционише тако што чита све из првог тока док не стигне до краја, потом прелази на други и тако даље док не стигне до краја последњег тока.

Пример 1. Демонстрирати употребу `FileInputStream` за читање података из датотеке чија је путања задата као аргумент командне линије. Прочитане бајтове исписати у формату карактера на конзоли. Тестирати читање из бинарне датотеке, на пример, слике у PNG формату, као и из текстуалне TXT датотеке у ASCII и UTF-8 кодним странама. ☐

```
package rs.math.oop.g15.p01.ulazniTokDatoteka;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ProcitajDatoteku {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
               System.err.println("Аргумент командне линије "
                      + "мора садржати путању до датотеке.");
               System.exit(1);
        }
        String putanja = args[0];
        FileInputStream fin = null;
        try {
               fin = new FileInputStream(putanja);
               int i = 0;
               while ((i = fin.read()) != -1)
                      System.out.print((char) i);
        } catch (FileNotFoundException e) {
               // FileNotFoundException
               System.err.println(e.getMessage());
        } finally {
               fin.close();
        }
    }
}
```

Програм је тестиран за три различите датотеке. Све три датотеке се налазе у оквиру кореног директоријума Јава пројекта, у под-директоријуму „ostalo“.
Очекивано, испис је смислен само за оне датотеке у којима је интерпретација бајтова карактерска. Код UTF-8 датотеке је текст написан мешовито ћирилицом и енглеским алфабетом: „Ово је пример UTF8 текста.“. Стога је коректно приказан само део текста. Када је у питању приказ слике у знаковној репрезентацији, јасно је да то нема смисла за највећи део садржаја, који је бинаран. Изузетак су одређени метаподаци који се налазе на почетку и на крају PNG датотеке.

```
java ProcitajDatoteku "ostalo/tekstASCII.txt"
Ovo je primer ASCII teksta.

java ProcitajDatoteku "ostalo/tekstUTF8.txt"
ÐÐ²Ð¾ ÑÐµ Ð¿ÑÐ¸Ð¼ÐµÑ UTF8 ÑÐµÐºÑÑÐ°.

java ProcitajDatoteku "ostalo/cmd.png"
PNG
...
óáÿw¾¬¤ IEND®B`¬¤■
```

Излазни ток података, OutputStream

У класи `OutputStream` најчешће се користи метод `write()`. Слично као и у претходном случају, излазне операције се обично не реализују директним позивима метода `write()`.

```
public abstract void write(int b) throws IOException;
public void write(byte b[]) throws IOException
public void write(byte b[], int offset, int len) throws IOException
```

Метод `flush()` служи за пражњење излазног бафера:

```
public void flush() throws IOException
```

Метод `close()` постоји и код излазних токова и улога му је иста као код улазних токова. Излазне операције реализоване преко класе `OutputStream` су операције тзв. ниског нивоа. Рад на том нивоу није атрактиван ни ефикасан (као и код улазног тока) па је развијен велики број потклаcа за организацију излаза „специјалних“ врста података.
Дакле, излаз се обично организује преко потклаcа класе `OutputStream` као што су:
`FileOutputStream`, `FilterOutputStream`, `ByteArrayOutputStream`, `ObjectOutputStream` итд.
Наравно, у овим класама се, по потреби, редефинишу методи класе `OutputStream`, али се уводе и нови методи.
Као и код улазних токова, хијерархија изведена из апстрактне класе `OutputStream` је врло разграната.

- `ByteArrayOutputStream` — садржи унутрашњи бафер из којег се одложено уписују подаци у низ бајтова. Применом метода `flush()` се форсира писање у низ бајтова и пражњење бафера.
- `FileOutputStream` — ток специјализован за писање бинарног садржаја у циљну датотеку.
- `FilterOutputStream` — користи неки други излазни ток као своју основу и над њим врши трансформације података у ходу. Постоји велики број потклаcа ове класе за различите намене, а само неке од њих су:
  - `BufferedOutputStream` — користи интерни бафер како би операције писања од стране Јава програма биле одложене. Ово доводи до тога да се смањује број системских позива за писање, што побољшава перформансе.
  - `CipherOutputStream` — користи се обично при писању шифрованог садржаја тако што у ходу чита основни ток са сировим подацима и шифрује га.
  - `DataOutputStream` — омогућава писање Јава примитивних типова на преносив начин (независан од платформе).
- `ObjectOutputStream` — омогућава писање објеката у циљну меморију (на пример датотеку, базу података и слично). Применом класе `ObjectInputStream` могуће је сачувани објекат учитати у меморију програма и користити тај објекат на исти начин као да је креиран применом оператора `new`.
- `PipedOutputStream` — овај излазни ток се повезује са својим парњаком `PipedInputStream` што надаље омогућава писање у излазни ток и читање са придруженог улазног. Обично се користи за комуникацију између више нити извршавања.

Пример 2. Демонстрирати употребу `DataOutputStream` за писање различитих примитивних типова на стандардни излаз, тј. конзолу. Такође, у одвојеној класи, демонстрирати употребу `DataOutputStream` и `FileOutputStream` за писање различитих примитивних типова у бинарну датотеку. Потом ту датотеку отворити и прочитати употребом `FileInputStream` и `DataInputStream` (инверзни кораци) . □

```
package rs.math.oop.g15.p02.ispisPodataka;

import java.io.DataOutputStream;
import java.io.IOException;

public class IspisiPodatkeKonzola {
    public static void main(String[] args) throws IOException {
        DataOutputStream tok = new DataOutputStream(System.out);
        try {
                tok.write(65);
                tok.flush();
                tok.writeChars(System.lineSeparator());
                tok.writeDouble(3432.3);
                tok.writeChars(System.lineSeparator());
                tok.writeUTF("Ovo je UTF8 текст са različitim писмима.");
                tok.writeChars(System.lineSeparator());
        } catch (IOException e) {
                System.err.println(e.getMessage());
        } finally {
                tok.close();
        }
    }
}
```

`DataOutputStream` је класа предвиђена за рад са бинарним садржајем те није најбољи избор у раду са конзолом, која је предвиђена за испис текстуалног садржаја. Кључни проблем је то што бинарни запис неких типова података, попут реалних бројева, нема лепу текстуалну репрезентацију.
У наставку следи испис произведен радом програма.

```
A

@Й

Ovo je UTF8 текст са različitim писмима.
```

Карактер ‘A’, тј. његова кодна репрезентација (65) не напуни излазни бафер, па је из тог разлога извршено експлицитно позивање пражњења бафера (метод `flush()`).
Приметити да испис реалног броја није разумљив – ово је очеивано, јер је у питању бинарни садржај који не мора имати смислену карактерску репрезентацију на конзоли.
Сличан испис се добија ако се `DataOutputStream` користи за писање у датотеку.

```
package rs.math.oop.g15.p02.ispisPodataka;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
public class IspisiPodatkeDatoteka {
    public static void main(String[] args) throws IOException {
        FileOutputStream fTokIzlaz = null;
        DataOutputStream tokIzlaz = null;
        try {
               fTokIzlaz = new FileOutputStream("izlaz.txt");
               tokIzlaz = new DataOutputStream(fTokIzlaz);
               tokIzlaz.write(65);
               tokIzlaz.writeDouble(3432.3);
        } catch (FileNotFoundException e) {
               System.err.println(e.getMessage());
        } finally {
               if (tokIzlaz != null)
                      tokIzlaz.close();
               if (fTokIzlaz != null)
                      fTokIzlaz.close();
        }
        FileInputStream fTokUlaz = null;
        DataInputStream tokUlaz = null;
        try {
               fTokUlaz = new FileInputStream("izlaz.txt");
               tokUlaz = new DataInputStream(fTokUlaz);
               int bajt = tokUlaz.read();
               System.out.println(bajt); // 65
               double broj = tokUlaz.readDouble();
               System.out.println(broj); // 3432.3
        } catch (FileNotFoundException e) {
               System.err.println(e.getMessage());
        } finally {
               if (tokUlaz != null)
                      tokUlaz.close();
               if (fTokUlaz != null)
                      fTokUlaz.close();
        }

    }
}
```

Претходни кôд демонстрира употребу инверзних функционалности: писање примитивних типова у бинарну датотеку и њихово накнадно читање из те исте датотеке. Резултат рада програма је:

```
65
3432.3
```

Као што се види, вредности које су унете у датотеку касније су из ње и прочитане и исписане на конзоли.
Следи садржај новоформиране датотеке.

```
A@ªÐ™™™™š
```

Може се приметити да је у питању бинарни формат, без јасне текстуалне репрезентације.■

15.1.2. Читачи и писачи

На основу примера из претходне секције се могло закључити да класе `InputStream` и `OutputStream`, као и њихове потклассе, не представљају добар основ за рад са текстуалним подацима. `Reader` и `Writer` класе су доста сличне токовима, с тим што уместо са бајтовима, раде са карактерима.
Како су `Reader` и `Writer` апстрактне класе, то се улаз/излаз реализује преко њихових потклаcа, као што су: `InputStreamReader`, `InputStreamWriter`, `FileReader`, `FileWriter`.

Читачи

Основни метод у класи `Reader` је `read()`. Он чита један цео број који представља кôд Unicode знака. Ако се при читању препозна крај улаза, метод враћа -1. Потпис овог метода је:

```
public abstract int read() throws IOExeption;
```

Поред метода `read()`, у овој класи су дефинисани и методи: `skip()`, `ready()`, `mark()`, `reset()` и `close()`. Метод `close()` има исту улогу као и код токова. Метод `skip()` прескаче задати број карактера, `ready()` враћа `true` или `false` у зависности од тога да ли је читач спреман за рад или не. Методи `mark()` и `reset()` функционишу заједно тако што `mark()` означи тренутну позицију у току. Након тога се ток може даље читати, а позивом метода `reset()` могуће је враћање на позицију означену са `mark()`.
Хијерархија класа изведених из `Reader` је структуирана по угледу на `InputStream`.

- `BufferedReader` — класа која се понаша слично попут `InputStreamReader` класе, с тим што ради са карактерима уместо са бајтовима. Подразумева се постојање основног читача чије услуге користи `BufferedReader`.
- `CharArrayReader` — читач који као извор података користи низ карактера прослеђен као аргумент конструктора.
- `FilterReader` — апстрактна класа која се понаша слично попут `FilterInputStream`.
    - `PushBackReader` — поред метода `read()` за читање следећег карактера, ова класа омогућава и метод за враћање тока на позицију претходног карактера помоћу метода `unread()`. Ово може бити корисно у ситуацијама када је потребно манипулисати тренутним карактерима на основу вредности будућих.
- `InputStreamReader` — ова класа је спона између бајтовских и карактерских токова података. Уз прецизирање кодне стране у стању је да конвертује ток бајтова у ток карактера.
- `PipedReader` — има сличну улогу као класа `PipedInputStream`, само за токове карактера.
- `StringReader` — слична је класи `CharArrayReader`, с тим што се као извор података прослеђује ниска.

Пример 3. За дату ниску, применом `StringReader` класе, приказати само оне карактере дате ниске такве да се у N наредних карактера након њих не налази ознака за празан простор. Поред метода `read()`, користити и методе `mark()` и `reset()`.☐

```
package rs.math.oop.g15.p03.citacNiske;

import java.io.IOException;
import java.io.Reader;
import java.io.StringReader;

public class CitajReci {
    public static void main(String[] args) throws IOException {
        String niska = "Пример неког кратког текста.";
        int N = 3;
        Reader citac = new StringReader(niska);
        try {
               while (true) {
                      int c = citac.read();
                      if (c == -1)
                             break;
                      citac.mark(N);
                      boolean prazan = false;
                      for (int i = 0; i < N; i++) {
                             char a = (char) citac.read();
                             if (a == -1)
                                    break;
                             if (a == ' ') {
                                    prazan = true;
                                    break;
                             }
                      }
                      citac.reset();
                      if (!prazan)
                             System.out.print((char)c);
               }
               System.out.println();
        } catch (IOException e) {
               System.err.println(e.getMessage());
        } finally {
               citac.close();
        }
    }
}
```

Након креирања читача ишчитава се карактер по карактер, све док се не прочита специјална ознака -1. Методом `mark()` се означи тренутна позиција, а као аргумент овог метода наведе се и колико највише карактера после те означене позиције ће бити прочитано (N). Након тога се чита наредних N карактера и проверава да ли се међу њима појављује празан карактер. Методом `reset()` се потом читач враћа на раније

означену позицију. Ако је у наредних N карактера било појављивања празног карактера, онда се тренутни карактер не исписује, а супротном се исписује.
Слична функционалност се може постићи и горепоменутим читачем под називом `PushBackReader`.
Након извршавања програма, добија се:

```
При не крат текста.■
```

Писачи

Сви писачи су изведени из апстрактне класе `Writer`. Метод `write()`, декларисан у класи `Writer`, преоптерећује се на следећи начин:

```
public abstract void write(byte b) throws IOException;
public void write(char cbuf[]) throws IOException
abstract public void write(char cbuf[], int off, int len) throws IOException
public void write(String str) throws IOException
public void write(String str, int off, int len) throws IOException
```

Поред метода `write()`, ту су још и:

- метод `append()` који служи за додавање знака/знакова у писач;
- метод `flush()` који служи за пражњење излазног бафера писача;
- метод `close()` који затвара писач, чиме се омогућује боље коришћење ресурса.

Хијерархија класа изведених из `Writer` је следећа:

- `BufferedWriter` — класа која врши испис карактера уз примену бафера, што доводи до ефикаснијег рада, јер се тај начин смањује број системских позива за упис у меморију.
- `CharArrayWriter` — писач који користи интерни карактерски бафер са динамичким растом (прилагођава се позивима метода за писање). Помоћу метода `toCharArray()` и `toString()` могуће је, у било којем моменту, вратити тренутну репрезентацију секвенце карактера у виду ниске.
- `FilterWriter` — апстрактна класа која се понаша слично попут класе `FilterOutputStream`.
- `OutputStreamWriter` — ова класа је спона између карактерских и бајтовских токова података. Уз прецизирање кодне стране у стању је да конвертује ток карактера у ток бајтова (инверзно понашање у односу на `InputStreamReader`).
    - `FileWriter` — наслеђује функционалности `OutputStreamWriter` класе и комбинује их са `FileOutputStream` функционалностима, што омогућава уписивање карактера у датотеку, чија се путања може подесити као аргумент конструктора.
- `PipedWriter` — понаша се као класа `PipedOutputStream` само за токове карактера.
- `PrintWriter` — класа за испис текста. Поседује преоптерећене методе `print()` и `println()` (са новим редом на крају) за испис примитивних типова и објеката, као и метод за форматирани испис `printf()`.
- `StringWriter` — слична је класи `CharArrayWriter` с тим што се као унутрашњи бафер користи објекат класе `StringBuffer` уместо ниске карактера.

Пример 4. Реализовати програм који задати низ ниски уписује у датотеку чија је путања дата као аргумент командне линије. За испис у датотеку користити класу `PrintWriter`.□

```
package rs.math.oop.g15.p04.ispisPodatakaKarakteri;

import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class IspisiNiskiUDatoteku {
    public static void main(String[] args) {
        if (args.length != 1) {
               System.err.println("Аргумент командне линије "
                             + "мора садржати путању до датотеке.");
               System.exit(1);
        }
        String putanja = args[0];

        String[] niske = new String[] {
                       "Низ од", "неколико ниски",
                       "свака", "записана у", "засебном реду",
                       "излазне", "датотеке."
        };

        PrintWriter pisac = null;
        try {
               pisac = new PrintWriter(putanja);
               for(String niska: niske)
                      pisac.println(niska);
        } catch (FileNotFoundException e) {
               System.err.println(e.getMessage());
        }finally {
               if(pisac!=null)
                      pisac.close();
        }
    }
```

Класа `PrintWriter` се може искористити и за писање по стандардном излазу, тј. подразумевано конзоли ако се, као аргумент конструктора, проследи `System.out`. `System.out` представља инстанцу класе `PrintStream` која је, према списку метода, слична класи `PrintWriter`. Разлика је у томе што `PrintStream` исписује ток бајтова, а `PrintWriter` ради са карактерима. Међутим, у већини случајева је употреба `System.out` придружених метода за испис на конзоли довољно добра (на пример преоптерећени метод `System.out.println()`).

Резултат рада програма је:

```
java IspisNiskiUDatoteku "izlaz.txt"
Низ од
неколико ниски
свака
записана у
засебном реду
излазне
датотеке.■
```

15.1.3. Уланчавање токова

У неким од ранијих примера са токовима података демонстрирана је могућност комбиновања различитих токова. У примеру 2 је току `DataOutputStream` прослеђен ток `FileOutputStream` како би се примитивни типови записали у датотеку. Овај концепт је заступљен и код читача/писача, и назива се уланчавање. Уланчавање функционише тако што се основни ток проследи као аргумент конструктора другом току. Други ток (надток) даље користи услуге основног тока у реализацији својих метода. На овај начин се врши надограђивање функционалности једне класе, без употребе наслеђивања — наслеђивање није ни могуће употребити, јер сви токови већ наслеђују класу `InputStream`.

Сличан приступ се користи и код читача и писача, чак је могуће уланчавање читача/писача са улазним/излазним токовима, што демонстрира наредни пример.

Пример 5. Прочитати карактере из задате текстуалне датотеке у UTF-8 формату. У реализацији користити уланчавање класа `FileInputStream` (за приступ току бајтова датотеке), `InputStreamReader` (за конверзију бајтова у карактере) и `BufferedReader` (за побољшање перформанси).□

```
package rs.math.oop.g15.p05.ulancavanjeTokova;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;

public class ProcitajDatotekuKaraktera {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
               System.err.println("Аргумент командне линије "
                             + "мора садржати путању до датотеке.");
               System.exit(1);
        }
        String putanja = args[0];

        InputStream fTok = null;
        Reader citac = null;
        Reader bCitac = null;
        try {
               fTok = new FileInputStream(putanja);
               citac = new InputStreamReader(fTok);
               bCitac = new BufferedReader(citac);
               int c;
               do {
                      c = bCitac.read();
                      System.out.print((char) c);
               } while (c != -1);
               System.out.println();
        } catch (Exception e) {
               System.err.println(e.getMessage());
        } finally {
               if (fTok != null)
                      fTok.close();
```

```
                if (citac != null)
                        citac.close();
                if (bCitac != null)
                        bCitac.close();
        }
    }
}
```

Као што се може видети, најпре је креиран ток типа `FileInputStream`. Његова надлежност је да чита бајт-по-бајт из улазне датотеке. Тај ток-објекат је прослеђен конструктору `InputStreamReader`. Ток-објекат класе `InputStreamReader` позивима метода `read()` над енкапсулираним ток-објектом `FileInputStream` формира сложеније типове података на основу добијених сирових бајтова. На крају, ток-објекат типа `BufferedReader` енкапсулира ток-објекат типа `InputStreamReader`, па прочитане податке баферише (одлаже њихово слање програму) како би се повећала ефикасност читања.
Приказ рада програма за аргумент командне линије “ostalo/tekstUTF8.txt” :

```
java ProcitajDatotekuKaraktera "ostalo/tekstUTF8.txt"
Ово је пример UTF8 текста.■
```

15.1.4. Рад са датотекама — класа File

У претходним секцијама било је речи о начелном раду са токовима бајтова и карактера, без превеликог осврта на извор у којем су ти бајтови/карактери похрањени.
У овој секцији посвећује се специјална пажња једном од могућих извора – датотекама. Поред тога, у овој секцији се говори и о организацији система датотека путем добро познатог концепта директоријума (фолдера), као и о метаподацима који додатно описују датотеке и директоријуме.

Подаци у спољашњој меморији рачунарског система су обично организовани у виду датотека и директоријума. Датотека представља колекцију података који чине једну логичку целину, а директоријуми (фолдери) служе за груписање датотека.
У програмском језику Јава, у оквиру библиотеке Java IO, за рад са датотекама и директоријумима се користи класа `File`.
Инстанца класе `File` заправо не представља датотеку, већ енкапсулира путању до нечега што може, а не мора бити датотека или директоријум. `File` објекат са путањом до неке датотеке или директоријума не значи да сама та датотека или директоријум постоји. Често се дефинише `File` објекат који енкапсулира путању до нове датотеке или новог директоријума који ће тек касније бити креиран.
У класи `File` постоји неколико конструктора. Први очекује као аргумент ниску са путањом датотеке или директоријума:

```
File dir = new File("C:/Program Files/Java");
File dir = new File("C:\\Program Files\\Java");
```

Следећа два конструктора омогућују да се одвојено задају родитељски директоријум и име датотеке:

```
File dir = new File("C:/Program Files/Java");
File dat = new File(dir, "Primer.java");
File dat = new File("C:/Program Files/Java", "Primer.java");
```

Приликом рада са `File` објектима, могу се користити и апсолутне и релативне путање.
Пример коришћења релативне путање:

```
File dat = new File("izlaz.txt");
```

Путања је релативна у односу на текући директоријум па се датотека "output.txt" налази у директоријуму где је и програм.
Класа `File` садржи преко тридесет метода.

Метода	**Опис**
`getName()`	Враћа име датотеке, не укључујући путању. Ако објекат представља директоријум, враћа само име директоријума.
`getPath()`	Враћа путању, укључујући име датотеке или директоријум.
`getAbsolutePath()`	Враћа апсолутну путању датотеке односно директоријума реферисаног текућим `File` објектом.
`isAbsolute()`	Враћа `true` ако је путања апсолутна, `false` иначе.
`getParent()`	Враћа име родитељског директоријума (за датотеку или директоријум).
`getParentFile()`	Враћа родитељски директоријум као `File` објекат или `null` ако не постоји родитељ.
`toString()`	Враћа исту ниску исти као `getPath()`.
`equals(File f)`	Користи се за поређење два `File` објекта. Пореде се путање.
`exists()`	Испитује да ли датотека/директоријум постоји.
`isDirectory()`	Испитује да ли је у питању директоријум.
`isFile()`	Испитује да ли је у питању датотека.
`isHidden()`	Испитује да ли је датотека/директоријум сакривен (на нивоу ОС).
`canRead()`	Испитује да ли се датотека може прочитати, директоријум листати.
`canWrite()`	Испитује да ли се по датотеци може писати, садржај директоријума мењати.
`list()`	Ако текући `File` објекат представља директоријум, метод враћа низ ниски са именима чланова директоријума, иначе враћа `null`.
`listFiles()`	Ако је текући објекат директоријум, враћа низ `File` објеката који одговарају датотекама и директоријумима у том директоријуму.
`length()`	Враћа вредност типа `long` која представља величину датотеке на коју реферише текући `File` објекат (у бајтовима). Ако се ради о датотеци који не постоји, враћена дужина биће 0. Ако објекат реферише на директоријум, није дефинисано шта је повратна вредност метода.
`lastModified()`	Враћа вредност типа `long` која представља време када је датотека или директоријум, реферисан текућим објектом, последњи пут измењен. Време је изражено бројем милисекунди протеклих од поноћи 1. јануара 1970. по Гриничу.
`listRoots()`	Статички метод класе `File` враћа низ `File` објеката, при чему сваки елемент у низу одговара кореном директоријуму текућег система датотека. Путања сваке од датотека у систему почиње неким од корених директоријума. (На Unix систему враћени низ имаће само један елемент који одговара јединоm

	кореном директоријуму ‘/’. Под Windows-ом низ садржи по елемент за сваки логички уређај, укључујући floppy, CD, DVD.)
`renameTo(String n)`	Датотека/директоријум реферисан текућим објектом бива преименован у складу са аргументом. Датотека/директоријум на који реферише текући објекат, након овога више не постоји, јер сада има ново име, а можда је и у другом директоријуму.
`setReadOnly()`	Враћа `true` ако је операција успела.
`mkdir()`	Креира директоријум са путањом одређеном текућим објектом. Метод не успева ако родитељски директоријум не постоји. Враћа `true` ако је операција успела.
`mkdirs()`	За разлику од претходног, креира неопходне родитељске директоријуме. Враћа `true` ако је операција успела. Чак и ако операција не успе, могуће је да су креирани неки од родитељских директоријума.
`createNewFile()`	Креира нову празну датотеку на путањи задатој текућим објектом, ако таква већ не постоји. Метод креира датотеку само у постојећем директоријуму (не креира директоријуме одређене путањом). Враћа `true` ако је операција успешна.
`createTempFile(String n, String e)`	Статички метод који креира привремену датотеку у задатом директоријуму (опциони трећи аргумент), са именом које одређују прва два аргумента метода. При томе, први аргумент одређује почетни део имена датотеке, а други његову екстензију. За име и екстензију се могу проследити `null` вредности и у том случају ће систем одредити насумично име и подразумевану екстензију “tmp”.
`delete()`	Брише датотеку/директоријум представљен текућим објектом и враћа `true` ако је брисање успело. Не брише директоријуме који нису празни. Да би се обрисао директоријум, најпре се мора обрисати сав његов садржај.
`deleteOnExit()`	Датотека/директоријум, представљен текућим објектом, биће избрисан по завршетку рада програма. Метод нема повратну вредност. Брисање ће бити покушано само ако се рад JVM заврши нормално.

Пример 6. Корисник уноси путању датотеке или директоријума као аргумент командне линије. Програм треба да изврши рекурзивни обилазак система датотека у дубину, почев од те путање. При том је потребно исписивати пуну апсолутну путању свих датотека (не и директоријума) на које метод наиђе.☐

```
package rs.math.oop.g15.p06.obilazakSistemaDatoteka;

import java.io.File;

public class RekurzivniObilazak {
    private static void obidjiUDubinu(String putanja) {
        File fAktivni = new File(putanja);
        if (!fAktivni.exists())
                return;
        if (fAktivni.isFile()) {
                System.out.println(fAktivni.getAbsolutePath());
                return;
        }
        File[] fPotomci = fAktivni.listFiles();
        if (fPotomci != null)
                for (File fp : fPotomci)
                        obidjiUDubinu(fp.getAbsolutePath());
    }

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

```
        if (args.length != 1) {
                System.err.println("Аргумент командне линије "
        + "мора садржати путању до датотеке.");
                System.exit(1);
        }
        obidjiUDubinu(args[0]);
    }
}
```

Метод `obidjiUDubinu()`, као што име каже, реализује обилазак дрвета система датотека у дубину. Излаз из рекурзије је двојак и дешава се: 1) ако актуелни `File` објекат реферише на датотеку или директоријум који не постоји или 2) ако актуелни `File` објекат представља датотеку. У супротном се врши рекурзивни обилазак за сваки од потомака текућег `File` објекта – потомцима се приступа позивањем метода `listFiles()`.

Приказ рада програма је:

```
java RekurzivniObilazak "C://Windows/Boot"
C:\Windows\Boot\BootDebuggerFiles.ini
C:\Windows\Boot\DVD\EFI\BCD
C:\Windows\Boot\DVD\EFI\boot.sdi
C:\Windows\Boot\DVD\EFI\en-US\efisys.bin
C:\Windows\Boot\DVD\EFI\en-US\efisys_noprompt.bin
C:\Windows\Boot\DVD\PCAT\BCD
…
C:\Windows\Boot\PCAT\zh-TW\memtest.exe.mui
C:\Windows\Boot\Resources\bootres.dll
C:\Windows\Boot\Resources\en-US\bootres.dll.mui■
```

15.2. Парсирање приликом читања/уписа — класа Scanner

Класа `Scanner`, описана је у секцији 6.4. Демонстрирана је њена употреба у парсирању података који долазе са стандардног улаза, тј. конзоле.

Ова класа не припада хијерархији токова нити читача, већ је директно изведена из класе `Object`. Међутим, она имплементира неке од интерфејса карактеристичних за улазне токове и читаче (`Closeable`, `AutoCloseable`) и додатно интерфејс `Iterator<String>`. Такође, приликом креирања, објекти класе `Scanner` кроз аргумент конструктора могу прихватити било који тип који имплементира интерфејс `Readable`, па самим тим и било који читач. Поред читача, могуће је проследити и објекат класе `File`, `String`, `InputStream` итд. Због овога се класа `Scanner` врло често користи као алтернатива читачима и улазним токовима или чак као примарно решење у обради улазних података.

Пример 7. Из датотеке “ostalo/studenti.txt” која има следећи садржај:

```
1009987567890 Марко Петровић 1987 23 9.33
2001967567890 Ана Ковачевић 1967 13 8.43
1009997567890 Марија Мирковић 1997 111 9.36
```

учитати студенте у листу објеката класе `Student`, који се описују редом ЈМБГ-ом, именом, презименом, годином рођења, бројем индекса и просечном оценом. Потом исписати елементе листе.

За реализацију програма може се користити раније дефинисана класа `Student` из примера 17 секције 14.6. За учитавање података из датотеке користити класу `Scanner`

са подешеном кодном страном UTF-8. За испис на конзолу користити `PrintWriter` са подешеном кодном страном UTF-8.☐

```
package rs.math.oop.g15.p07.ucitavanjeIzDatotekeSkener;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class UcitajStudente {
    public static void main(String[] args) {
        Scanner skener = null;
        try {
                List<Student> studenti = new ArrayList<>();
                skener = new Scanner(new File("ostalo/studenti.txt"), "UTF-8");
                while (skener.hasNext()) {
                        String JMBG = skener.next();
                        String ime = skener.next();
                        String prezime = skener.next();
                        int godinaRodjenja = skener.nextInt();
                        String indeks = skener.next();
                        double prosecnaOcena = skener.nextDouble();
                        Student student = new Student(JMBG, ime, prezime,
                                        godinaRodjenja, indeks, prosecnaOcena);
                        studenti.add(student);
                }
                for (Student student : studenti)
                        System.out.println(student);

        } catch (FileNotFoundException e) {
                System.err.println(e.getMessage());
        } finally {
                if (skener != null)
                        skener.close();
        }
    }
}
```

У програму се претпоставља да је улазна датотека адекватно структуирана, стога се не испитује постојање наредног податка (`hasNext()` метод).
Приказ рада програма је следећи:

```
1009987567890    Марко    Петровић    1987    23    9.33
2001967567890    Ана    Ковачевић    1967    13    8.43
1009997567890    Марија    Мирковић    1997    111    9.36■
```

15.3. Резиме

У овом поглављу је дат преглед неких основних техника за рад са улазом и излазом, тј. за читање и писање садржаја по токовима података. Ток података апстрахује реализацију метода за различите изворе (циљеве) података – није неопходно правити вишеструке реализације исте функционалности за, на пример, датотеку, базу података, стандардни улаз/излаз итд.

У оквиру пређашњих поглавља су се као извори (циљеви) података користили стандардни улаз и стандардни излаз, прецизније, тастатура и конзола. Сада је специјална пажња посвећена и датотекама. С обзиром на универзалност концепта тока података, његово разумевање би требало да омогући програмеру лако прилагођавање било ком новом извору (циљу) података.

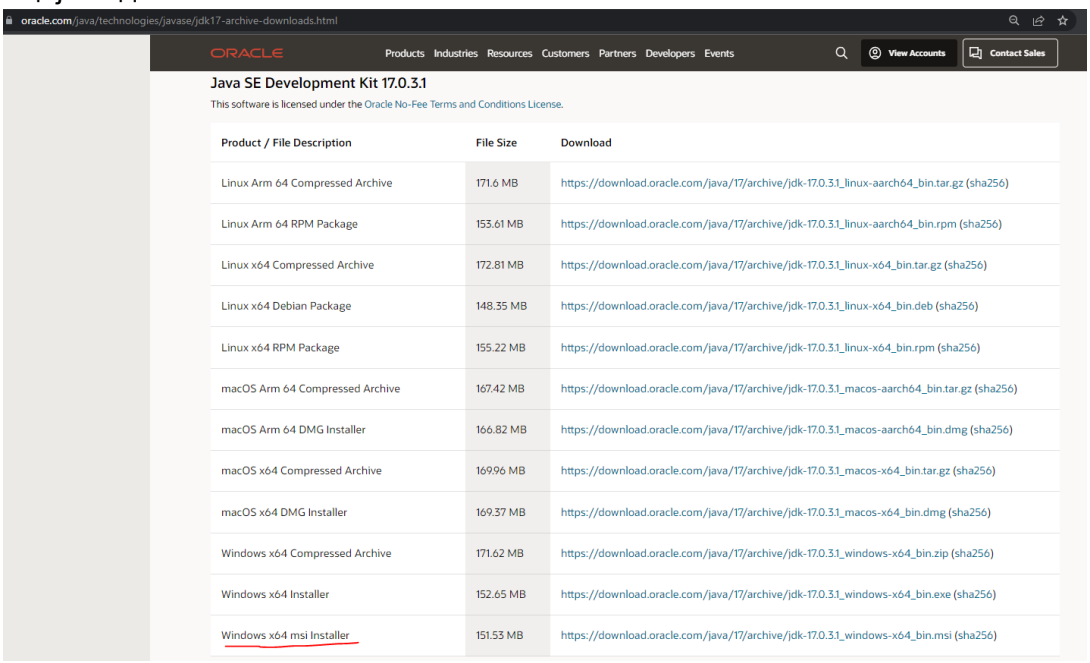
15.4. Питања и задаци

1. Упоредити рад са улазним и излазним подацима Јава IO библиотеке и Јава NIO библиотеке.
2. Које су класе изведене из апстрактне класе `InputStream`? Примером илустровати употребу неких од њих.
3. Које су класе изведене из апстрактне класе `OutputStream`? Које од тих класа имају своје парњаке у хијерархији класа изведених из `InputStream`? Илустровати примером употребу неких од парова.
4. Навести и примером илустровати употребу метода класе `Reader`.
5. Навести и примером илустровати употребу метода класе `Writer`.
6. Шта је уланчавање токова, како функционише и зашто се користи? Илустровати примером.
7. Написати Јава програм који за путању датотеке или директоријума коју корисник уноси са тастатуре, исписује све информације до којих се може доћи употребом метода класе `File`.
8. Објаснити како се класа `Scanner` може користи као алтернатива читачима и улазним токовима. Илустровати примером.
9. Податке из улазне датотеке треба преписати у излазну датотеку. Појединачна линија улазне датотеке садржи информацију о податку и његовом приоритету. Након учитавања 10 линија улазне датотеке у излазну датотеку се исписује 5 података који имају највиши приоритет. Након тога се учитава следећих 10 линија улазне датотеке, а у излазну се поново преписује 5 података који имају највиши приоритет. Поступак се понавља док се не дође до краја улазне датотеке, након чега се остатак података преписује у излазну датотеку по приоритету.
10. У улазној датотеци се налазе информације о студентима. Појединачна линија улазне датотеке садржи информације о једном студенту: број индекса (облика број/година уписа), име, презиме, година студија, начин финансирања и просечна оцена. Написати Јава програм који у излазну датотеку исписује информације о студентима који су уписани на факултет 2020. године, финансирају се из буџета и имају просечну оцену већу од 8.00.

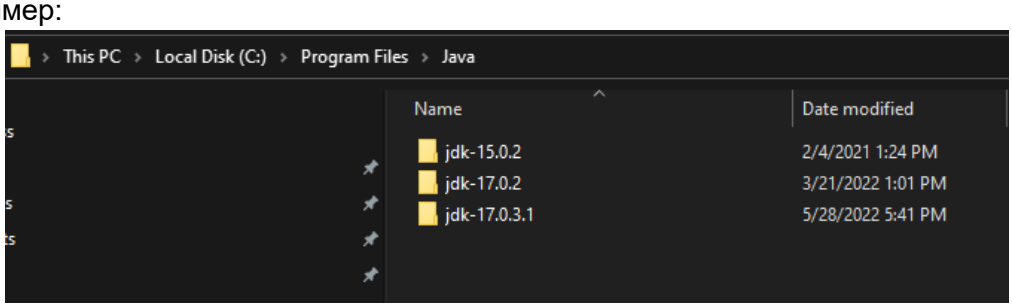
Додатак А – Инсталација Јаве и развојног окружења под Windows оперативним системом

Инсталација JDK

1. Први корак је преузимање одговарајуће JDK инсталације са Oracle веб сајта. Конкретно, за верзију Јаве 17 LTS адреса је: https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html
2. Потом је потребно преузети одговарајућу инсталацију, а најпрактичнији је избор опције под називом: “Windows x64 msi Installer”.



3. Након тога покренути преузету датотеку и довршити њену инсталацију.
4. По успешној инсталацији требало би да је креиран под-директоријум за одговарајућу инсталацију у оквиру директоријума “C://Program Files/Java”, на пример:

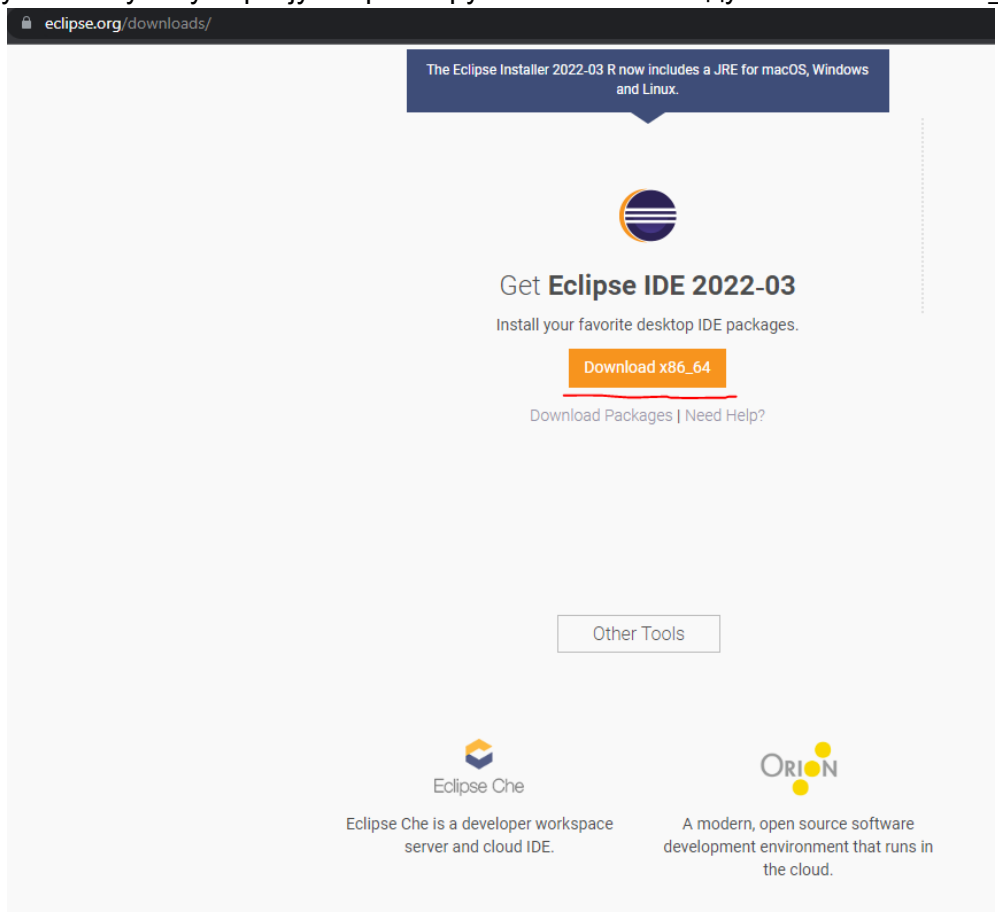


На слици се види да је могуће имати истовремено више инсталираних JDK. Која од њих ће се користити приликом развоја програма зависи од подешавања развојног окружења.

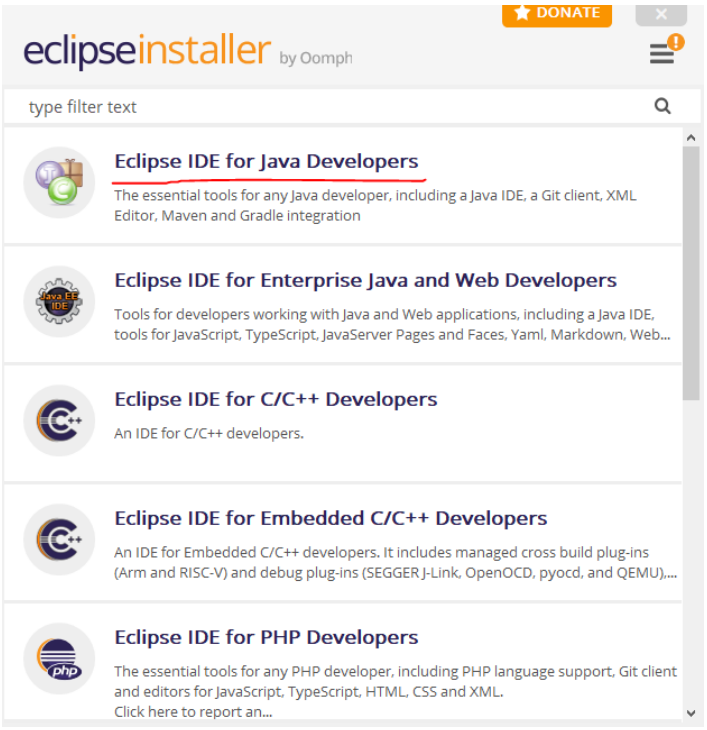
Инсталација развојног окружења Eclipse

1. Отићи на веб сајт Eclipse организације за преузимање програма, конкретно на адресу: https://www.eclipse.org/downloads/

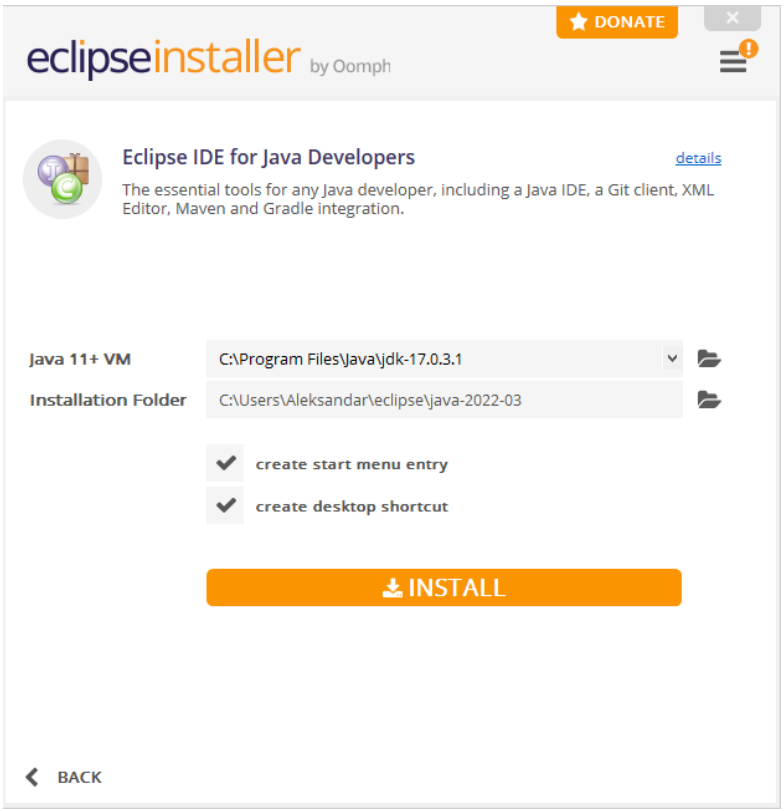
2. Преузети актуелну верзију Eclipse окружења кликом на дугме “Download x86_64”.



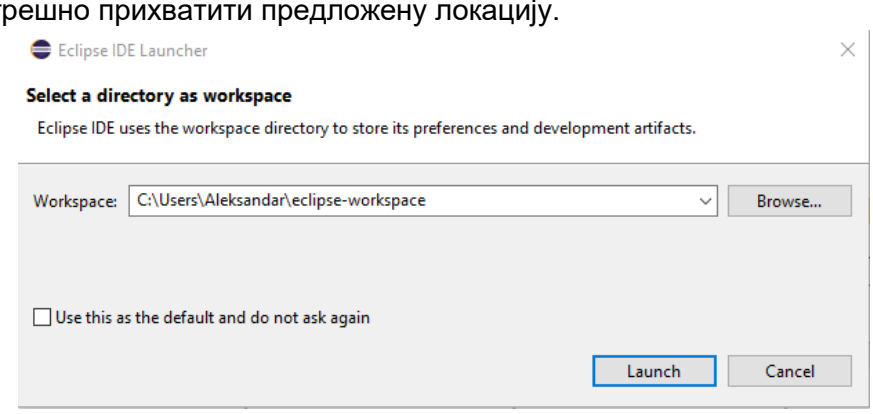
3. Након тога покренути преузету датотеку и започети њену инсталацију.
4. У оквиру првог инсталационог дијалога одабрати опцију “Eclipse IDE for Java Developers”:



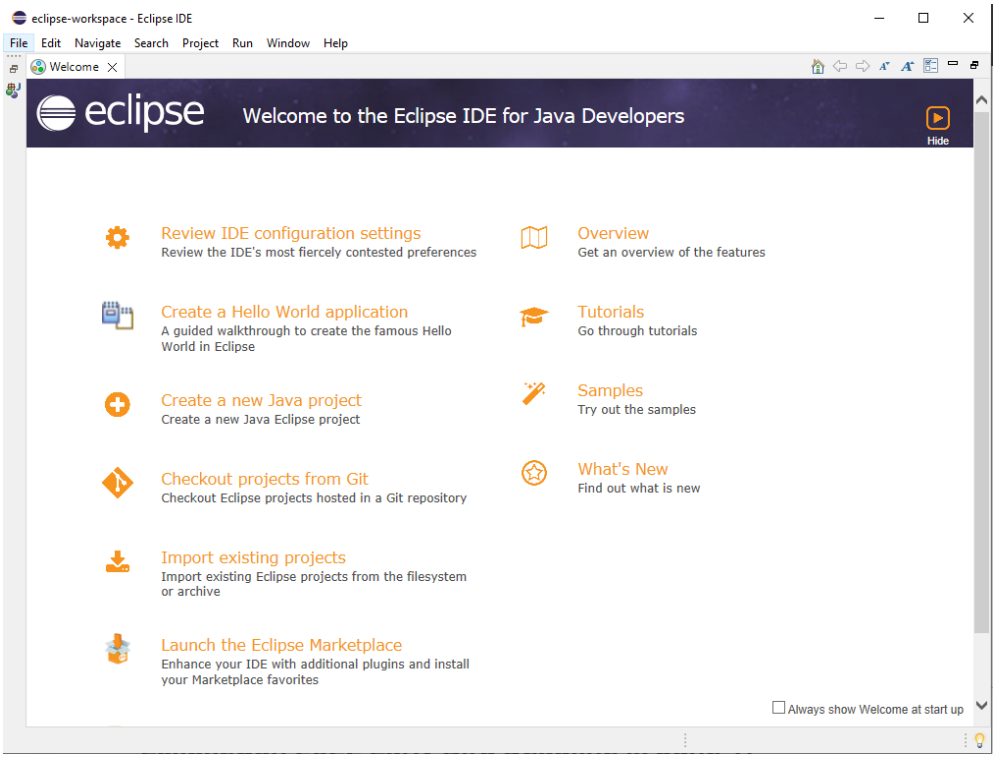
5. Следећи инсталациони дијалог ће укључивати неколико питања. Најважније је питање где се налази претходно инсталирана Јава.



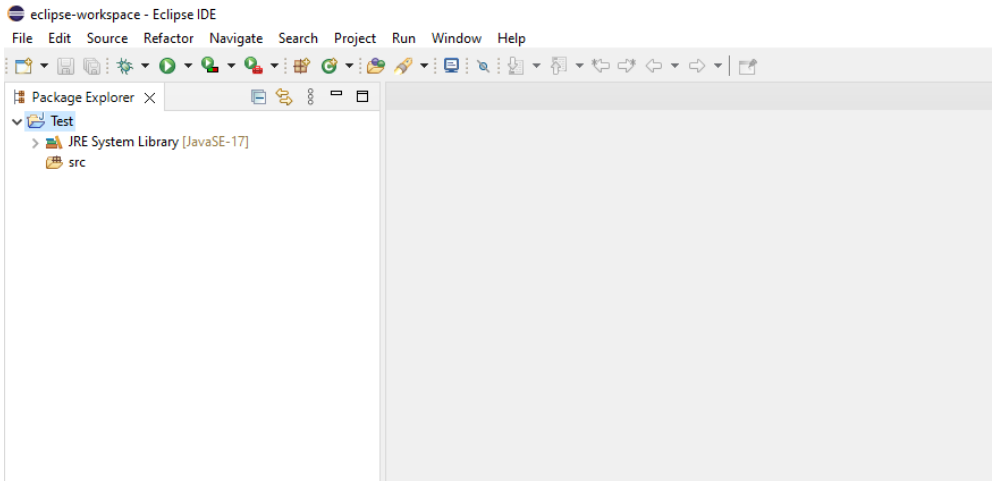
6. По успешној инсталацији могуће је покренути Eclipse развојно окружење кликом на дугме “LAUNCH” или алтернативно кликом на иконицу на радној површини.
7. Приликом покретања Eclipse, биће постављено питање где корисник жели да позиционира директоријум са кодовима (енг. workspace). Није погрешно прихватити предложену локацију.



8. Покренуто Eclipse окружење изгледа овако:

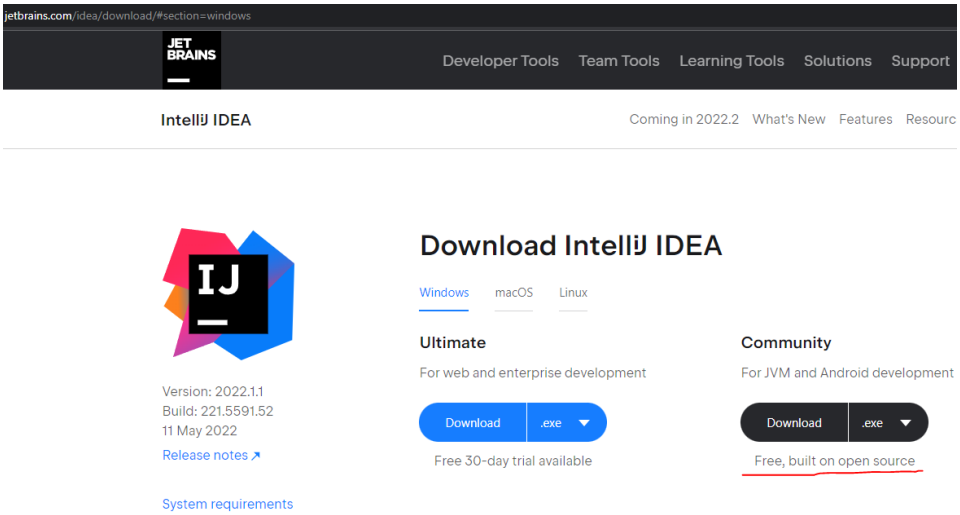


9. Угасити прозор добродошлице (енг. welcome).
10. Сада опцијом “File→New→Java Project” и задавањем имена пројекта креирати први пројекат.
11. Следећи дијалог може поставити питање у вези употребе система модула, уколико не планирате да их користите, одаберите опцију “Don’t Create”.
12. Тиме је завршена инсталација развојног окружења и покренут је пројекат, што би требало да изгледа оквирно овако:

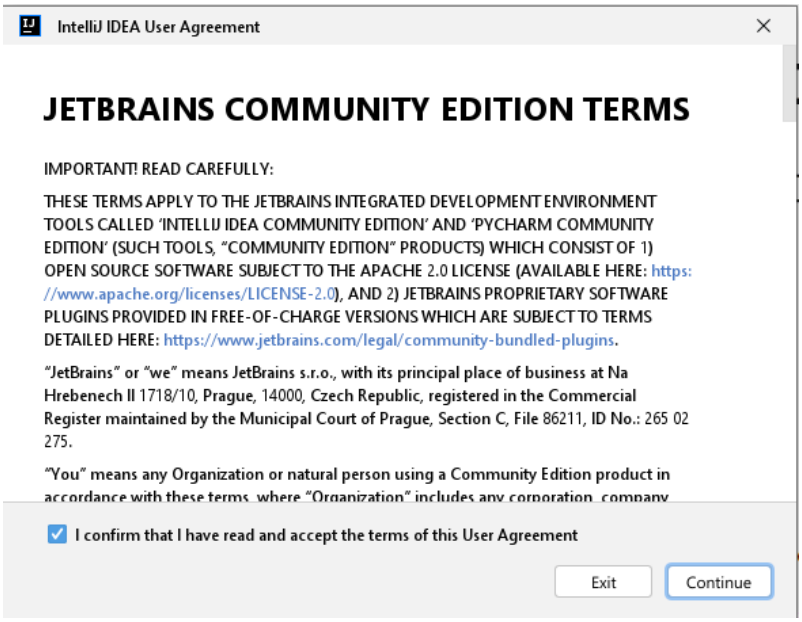


Инсталација развојног окружења IntelliJ

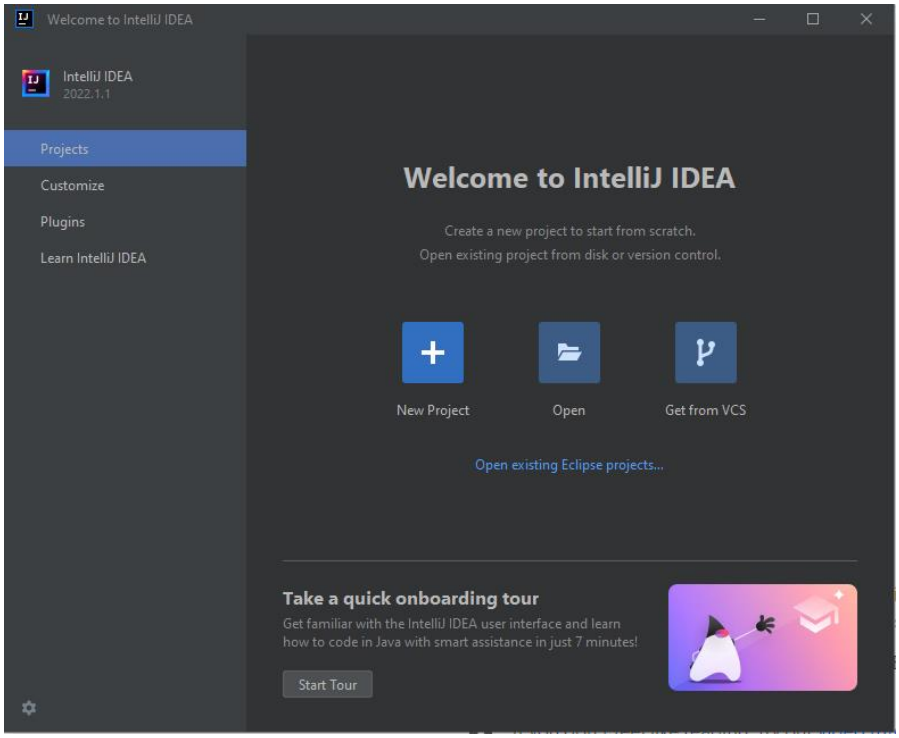
1. Посетити сајт компаније JetBrains на следећој веб локацији: https://www.jetbrains.com/idea/download/
2. Преузети “Community” верзију IntelliJ развојног окружења.



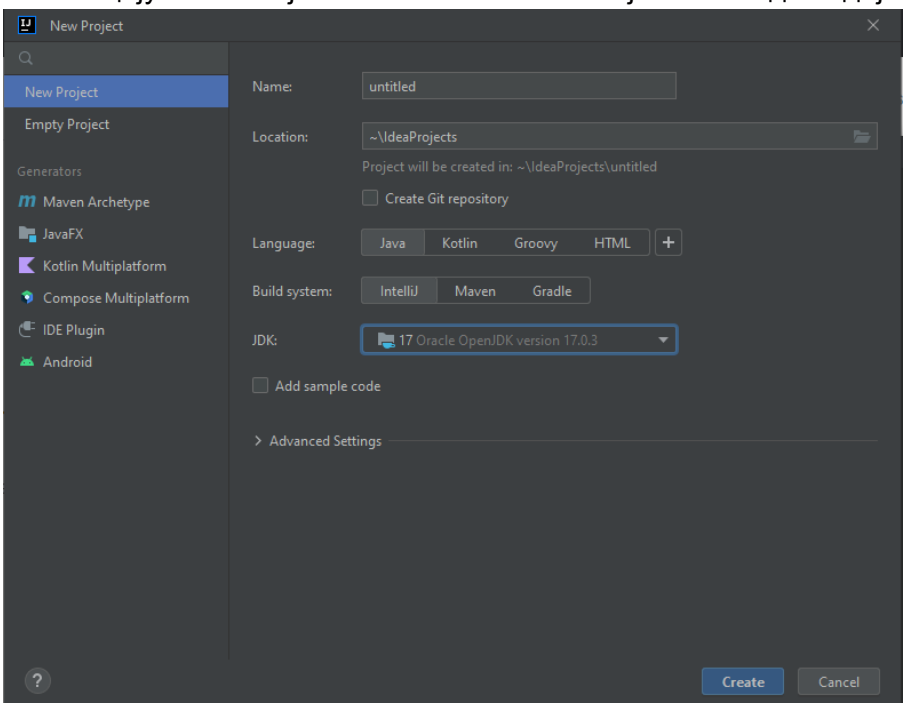
3. Након преузимања покренути инсталациону датотеку и проћи кроз процес инсталације (не би требало да буде нејасних питања).
4. По завршетку инсталације могуће је одмах покренути развојно окружење, а могуће је и накнадно путем иконице на радној површини и слично.
5. Појавиће се следеће питање приликом покретања:



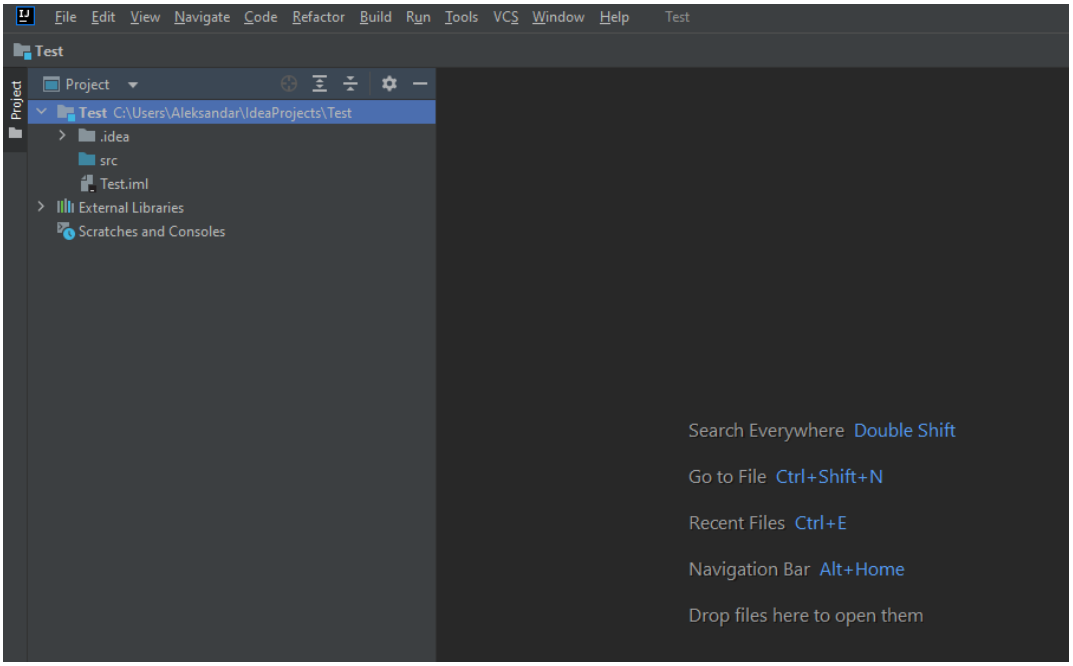
6. Почетни екран након покретања изгледа овако:



7. Кликнути на опцију “New Project”. Након тога ће се појавити следећи дијалог:



8. Унети назив пројекта и проверити да ли је JDK адекватно препознат од стране IntelliJ окружења. Потом кликнути на дугме “Create”.
9. Тиме је завршена инсталација развојног окружења и покренут је пројекат, што би требало да изгледа оквирно овако:

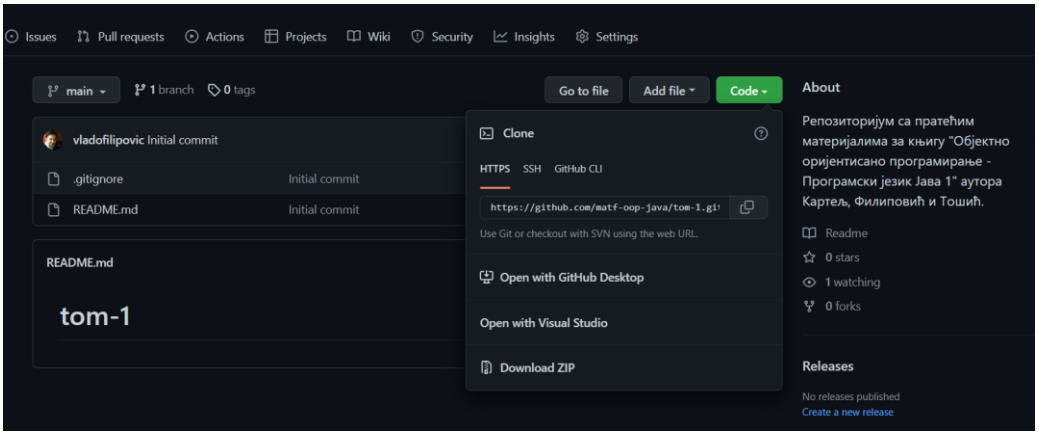

File Edit View Navigate Code Refactor Build Run Tools VCS Window Help Test
Test
Project
Test C:\Users\Aleksandar\IdeaProjects\Test
.idea
src
Test.iml
External Libraries
Scratches and Consoles
Search Everywhere Double Shift
Go to File Ctrl+Shift+N
Recent Files Ctrl+E
Navigation Bar Alt+Home
Drop files here to open them

Додатак Б – Упутство за употребу GitHub репозиторијума и подешавање ћирилице

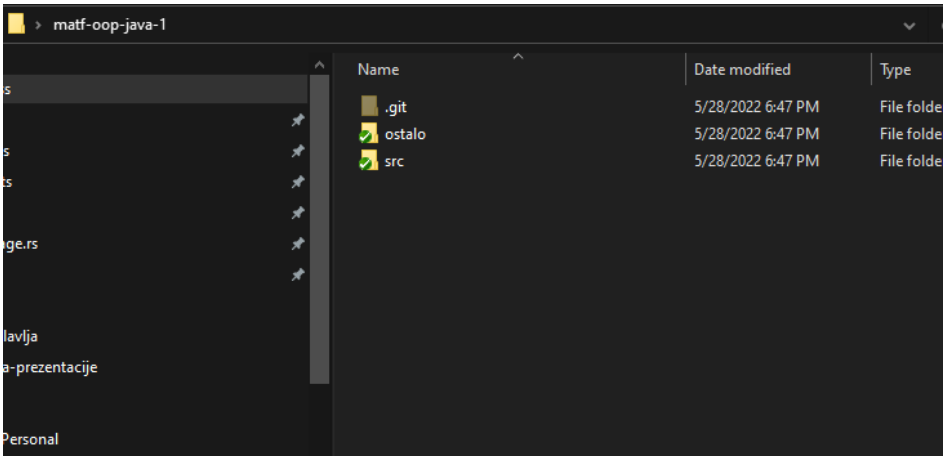
Сви примери коришћени у књизи налазе се на GitHub репозиторијуму:
https://github.com/matf-oop-java/tom-1
Следи упутство за преузимање (клонирање) овог репозиторијума и његову интеграцију са претходно наведеним развојним окружењима Eclipse и IntelliJ.

Преузимање (клонирање) GitHub репозиторијума

1. Отићи на веб сајт:
2. https://github.com/matf-oop-java/tom-1
   Кодови и пратећи подаци се могу преузети на најмање два начина: у виду .zip архиве или применом одговарајћег Git клијента.
3. За преузимање у виду .zip архиве кликнути на дугме “Code” и након тога одабрати опцију “Download ZIP”.
   Након преузимања, архиву распаковати на погодној локацији.



4. За преузимање путем Git клијента (клонирање) користити одговарајући клијент, нпр. TortoiseGit који се може преузети са адресе:
   https://tortoisegit.org.
   (Развојна окружења обично нуде и интегрисану подршку за Git, али то овде није објашњено.)
   a. За потребе клонирања помоћу TortoiseGit алата позиционирати се у одговарајући директоријум где желите да буде преузет репозиторијум.
   b. Потом десни клик мишем било где у отвореном директоријуму након чега се одабере опција “Git Clone”.
   c. Појавиће се следећи дијалог у који треба унети Git адресу репозиторијума https://github.com/matf-oop-java/tom-1.git
   и кликнути на дугме “OK”.
   d. Клонирани директоријум би требало да садржи под-директоријуме: “src” и “ostalo”.

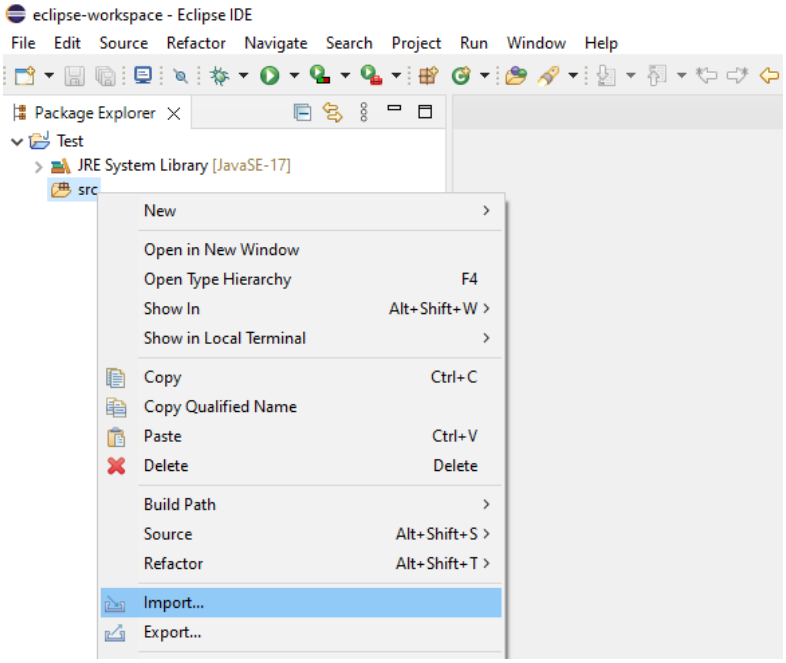


У оквиру директоријума “src” се налазе програмски кодови свих примера из књиге, док директоријум “ostalo” садржи неке пропратне датотеке које се користе у неким од примера.

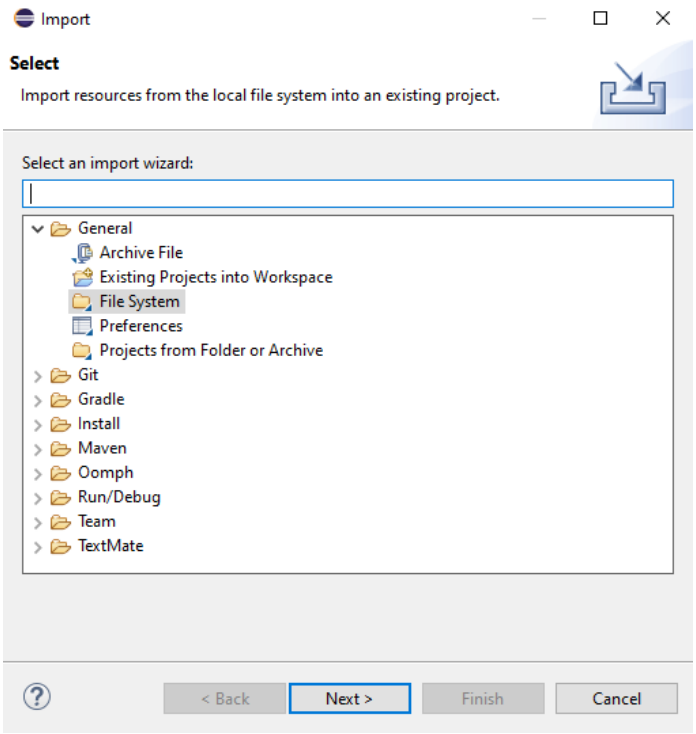
5. Након што је преузет и распакован или клониран материјал са GitHub-a, исти се може уградити у постојећи Eclipse или IntelliJ пројекат на начине описане у следећим секцијама.

Уграђивање GitHub материјала у постојећи Eclipse пројекат

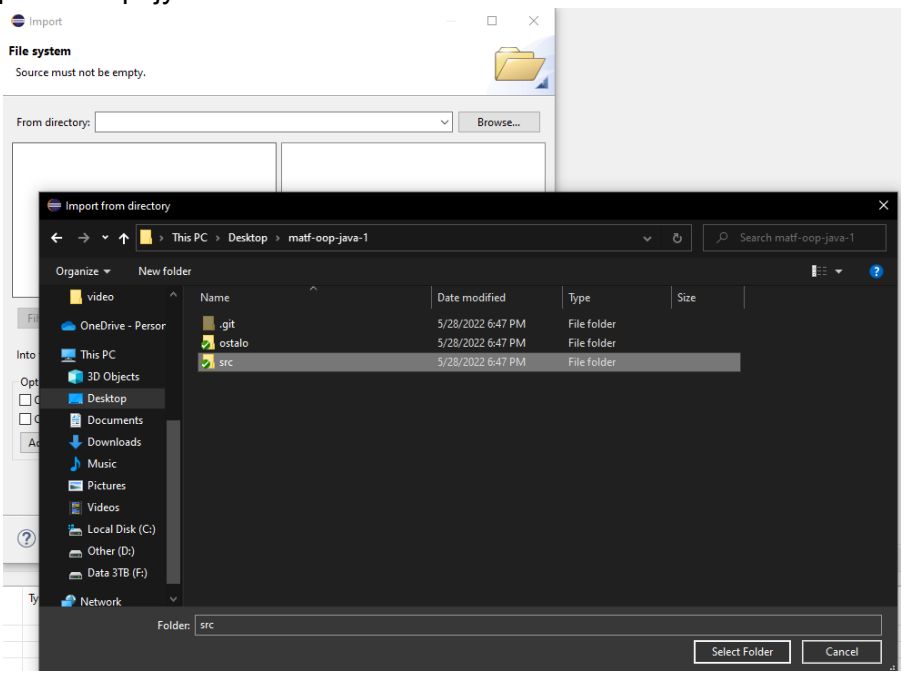
1. У оквиру раније креираног Јава пројекта обележити директоријум “src”, потом десни клик мишем и онда опција “Import”.



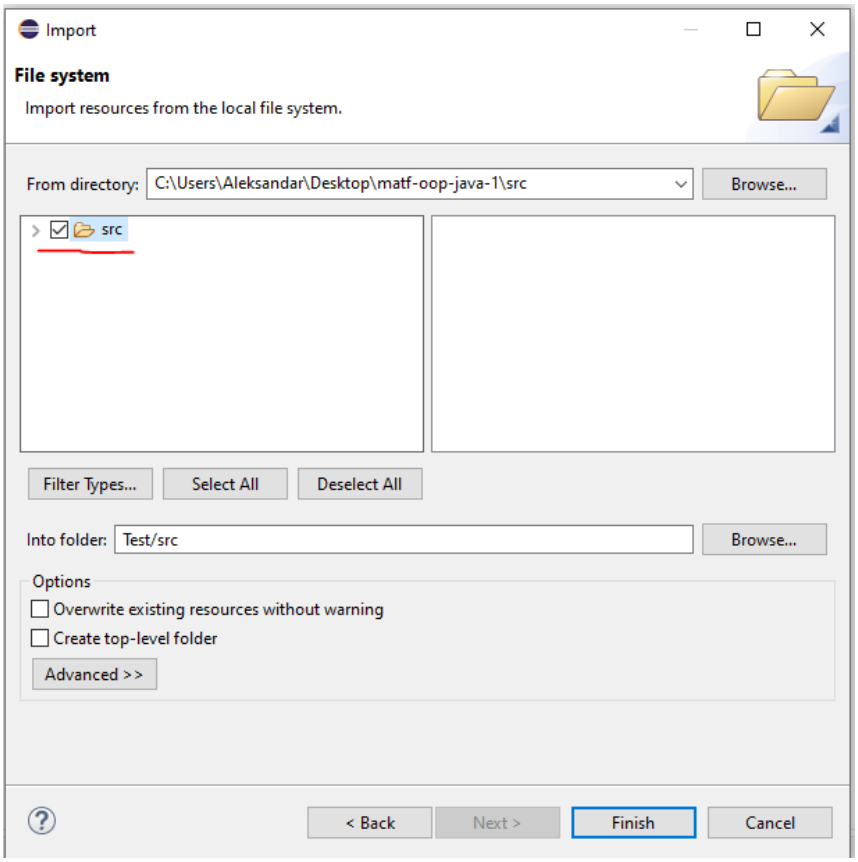
2. Одабрати потом извор одакле се увози “General→File System”.



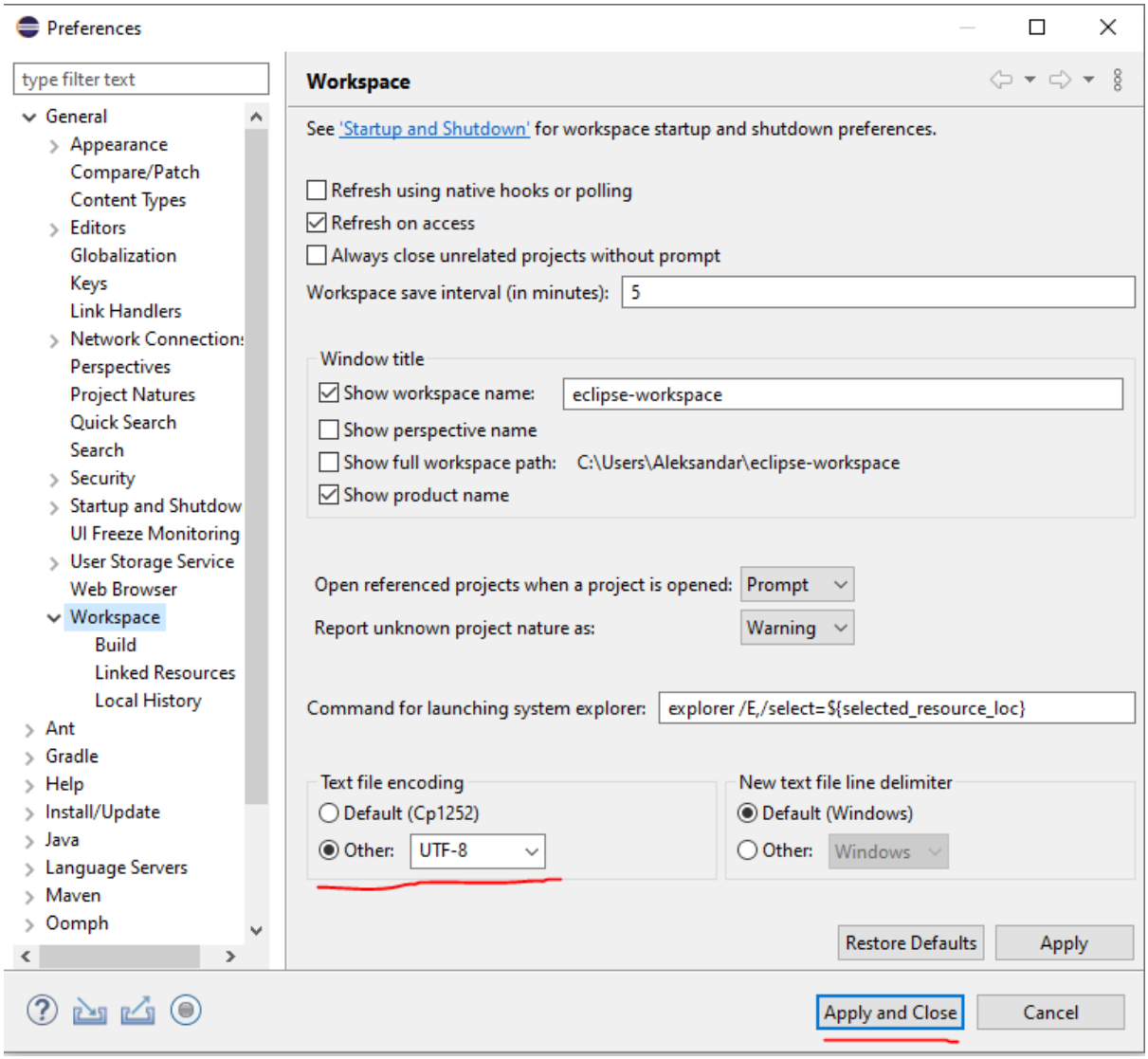
3. За опцију “From directory:” поставити “src” директоријум у оквиру преузетог GitHub репозиторијума.



4. Чекирати опцију поред ознаке “src” и кликнути дугме “Finish”.



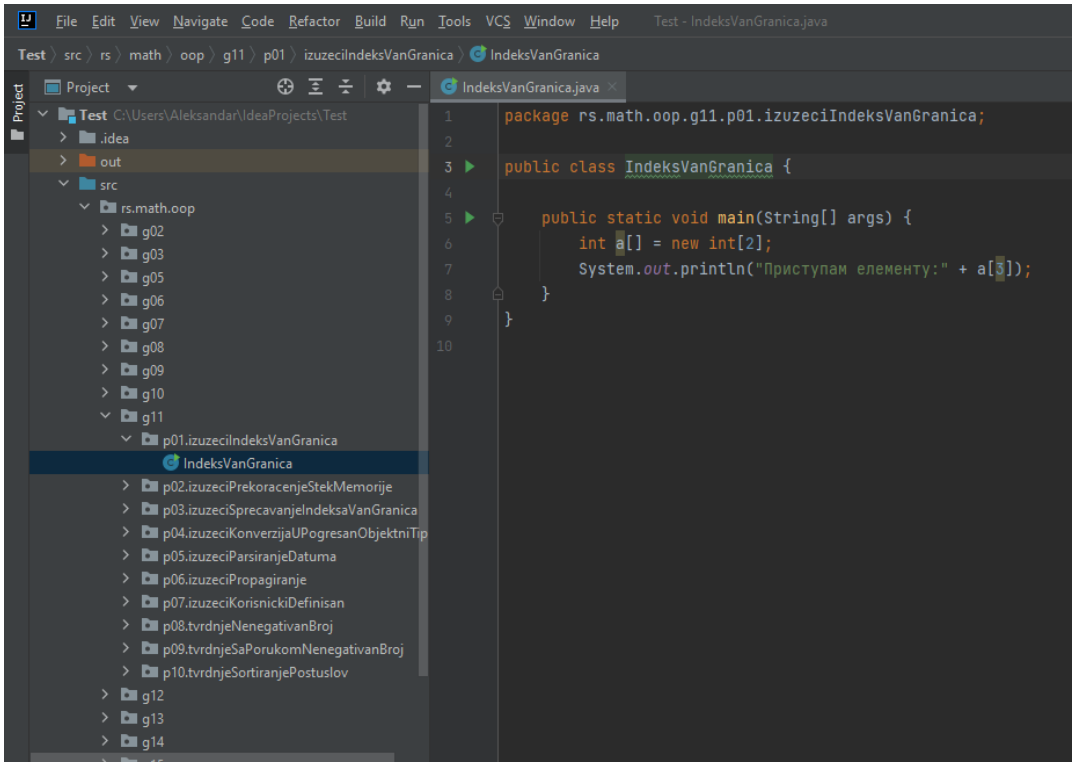
5. Након успешног учитавања свих примера вероватно ће се појавити одређене грешке. Разлог је у томе што већина примера користи ћирилично писмо које није подразумевано подешено у оквиру Eclipse пројекта.
   Да би се решио овај проблем потребно је отићи на опцију “Window→Preferences→General→Workspace” и након тога поставити “Text file encoding” на UTF-8, затим сачувати кликом на “Apply and Close”.



6. Након тога би требало да су сви примери функционални, да нема грешака и да се ћирилично писмо уредно приказује у Јава текстуалном едитору.

Уграђивање GitHub материјала у постојећи IntelliJ пројекат

1. Обележити директоријум “rs” који се налази у директоријуму “matf-oop-java-1/src” и копирати га (на тастатури CTRL+C или десни клик мишем па опција “Copy”).
2. У оквиру раније креираног Јава пројекта означити мишем директоријум “src”, и у њега налепити претходно копирани директоријм “rs” (на тастатури CTRL+V или десни клик мишем па опција “Paste”) .
3. Требало би да су након тога сви примери (из свих поглавља) учитани и да је кодна страна аутоматски подешена.


File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Test - IndeksVanGranica.java
Test src rs math oop g11 p01 izuzeciIndeksVanGranica IndeksVanGranica
Project
IndeksVanGranica.java
Test C:\Users\Aleksandar\IdeaProjects\Test
.idea
out
src
rs.math.oop
g02
g03
g05
g06
g07
g08
g09
g10
g11
p01.izuzeciIndeksVanGranica
IndeksVanGranica
p02.izuzeciPrekoracenjeStekMemorije
p03.izuzeciSprecavanjeIndeksaVanGranica
p04.izuzeciKonverzijaUPogresanObjektniTip
p05.izuzeciParsiranjeDatuma
p06.izuzeciPropagiranje
p07.izuzeciKorisnickiDefinisan
p08.tvrdnjeNenegativanBroj
p09.tvrdnjeSaPorukomNenegativanBroj
p10.tvrdnjeSortiranjePostuslov
g12
g13
g14
package rs.math.oop.g11.p01.izuzeciIndeksVanGranica;
public class IndeksVanGranica {
public static void main(String[] args) {
int a[] = new int[2];
System.out.println("Приступам елементу:" + a[3]);
}
}

Садржај

Литература

Arnold K, J.Gosling i D. Holmes, Programski jezik Java, CET, Beograd, 2001.

Arnold K, Gosling J: The Java Programming Language, Addison Wesly, 1996.

Eckel B: Thinking in Java, Prentice Hall, 2003.

Findlay W, Watt D: PASCAL, An Introduction to Methodical Programming, Pitman, London, 1978.

Flanders H: Scientific PASCAL, Reston Publishing Company, Reston, 1984.

Grogono P: Programming in PASCAL, Addison-Wesley, Reading, Massachusetts, 1980.

Horstmann C, Cornell G: Core JAVA, Volume I Fundamentals, Sun Microsystems, Inc. 2005.

Horstmann C, Cornell G: Core JAVA, Volume II Advanced Fetures, Sun Microsystems, Inc. 2005.

Horton I: Java2 - JDK 1.5, CET, Beograd, 2006.

Lemay L, Cadenhead R: Java 1.2, Kompjuter biblioteka, Čačak, 2001.

Lemay L, Perkins Ch.L: Teach Yourself JAVA in 21 Days, Sums-Net, 1996.

Niemeyer P, Peck J: Exploring Java, O'Reilly & Associates, Inc.1996.

Schildt H: JavaProgramming Cookbook, McGraw-Hill, 2008.

Stojković V, Tošić D, Stojmenović I: Programski jezik Pascal, Naučna knjiga, Beograd, 1990.

Simonyi C: Hungarian Notation, MSDN Library, Microsoft Corp, novembar 1999.

Tenenbaum A.M, Augenstein M.J: Data Structures Using PASCAL, Prentice-Hall International, Englewood Cliffs, New Jersey, 1981.

Tošić D: Pascal-osnovi programiranja, Studentski trg, Beograd, 1977.

Tošic D, Stojković V: Programski jezik Pascal – zbirka rešenih zadataka, Tehnička knjiga, Beograd, 1991.

Тошић Д: Рачунарство и информатика за III разред гимназије, Завод за уџбенике, Београд, 2011.

Чабаркапа М: Рачунарство и информатика – уџбеник са збирком задатака за 4.разред гимназије природно-математичког и општег смера, Круг, Београд, 2008.

Wirth N: Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs, New Jersey, 1979.