

Владан Девеџић
Синиша Влајић
Саша Д. Лазаревић (Уредници)

Практикум за припремање пријемног испита за софтверско инжењерство

Beograd, 2017.

**Владан Девеџић
Синиша Влајић
Саша Д. Лазаревић (Уредници)**

Практикум за припремање пријемног испита за софтверско инжењерство

Beograd, 2017.

ПРАКТИКУМ ЗА ПРИПРЕМАЊЕ ПРИЈЕМНОГ ИСПИТА ЗА СОФТВЕРСКО ИНЖЕЊЕРСТВО

Уредници:

Владан Девецић
Синиша Влајић
Саша Д. Лазаревић

Аутори (наведени по азбучном редоследу презимена):

Илија Антовић
Синиша Влајић
Драган Ђурић
Јелена Јовановић
Саша Д. Лазаревић
Никола Миликић
Милош Милић
Душан Савић
Војислав Стanoјevић
Бојан Томић
Зоран Шеварац

Издавач: Факултет организационих наука, Јове Илића 154, Београд

Рецензенти:

Проф. др Драган Бојић
Проф. др Биљана Стаматовић

Штампа: Електронско издање

Издање: Прво

ISBN 978-86-7680-338-5

На основу одлуке Издавачког одбора Факултета организационих наука Универзитета у Београду 04-07 број 3/13 од 10.4.2017. одобрава се јавно постављање линка (на сајту факултета) са садржајем овог уџбеника и то као обавезне уџбеничке литературе за припрему пријемног испита на Мастер академским студијама за студијски програм Софтверско инжењерство и рачунарске науке.

©Copyright 2017. Факултет организационих наука. Издавачи задржавају сва права. Није дозвољена репродукција или емитовање поједињих делова или целине ове публикације на било који начин. Дозвољена је искључиво лична, некомерцијална, употреба публикације у сврху припремања пријемног испита на Мастер академским студијама за студијски програм Софтверско инжењерство и рачунарске науке.

САДРЖАЈ

I ДЕО – ОСНОВЕ ПРОГРАМСКОГ ЈЕЗИКА ЈАВА

I-1 .. I-123

Бојан Томић, Јелена Јовановић, Никола Миликић, Зоран Шеварац, Драган Ђурић

II ДЕО – НАПРЕДНЕ ЈАВА ТЕХНОЛОГИЈЕ

II-1 .. II-84

Синиша Влајић, Душан Савић, Илија Антовић, Војислав Стамојевић, Милош Милић

III ДЕО – ПРОЈЕКТОВАЊЕ СОФТВЕРА

III-1 .. III-60

Синиша Влајић

IV ДЕО – РЕЛАЦИОНИ УПИТНИ ЈЕЗИК

IV-1 .. IV-71

Саша Ђ. Лазаревић

I ДЕО – ОСНОВЕ ПРОГРАМСКОГ ЈЕЗИКА ЈАВА

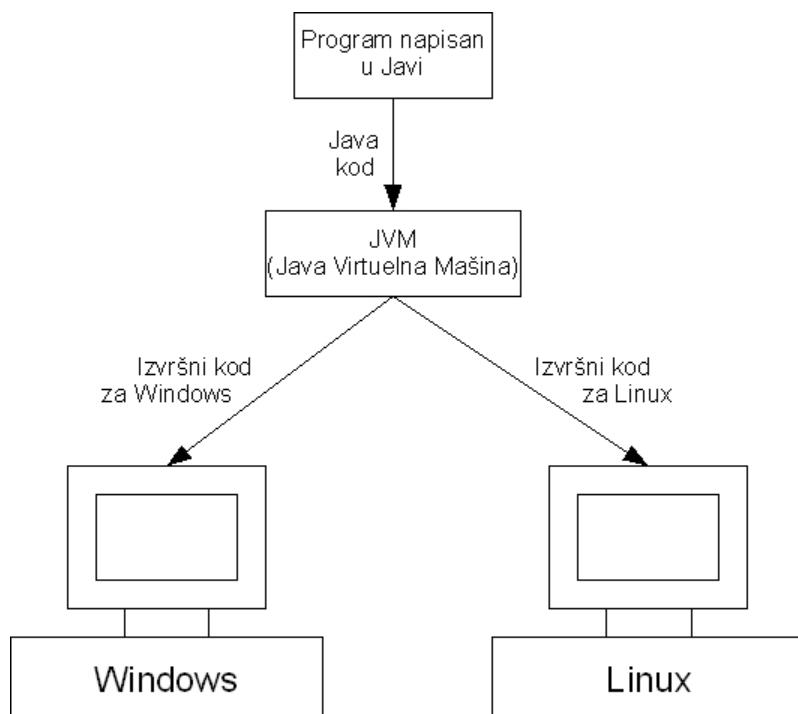
Бојан Томић, Јелена Јовановић, Никола Миликић, Зоран Шеварац, Драган Ђурић

САДРЖАЈ

| | |
|---|------------|
| 1 Програмски језик Јава - основне информације..... | 2 |
| 2 Класе, објекти и њихови елементи..... | 5 |
| 2.1 Класе..... | 5 |
| 2.2 Атрибути..... | 8 |
| 2.3 Објекти..... | 9 |
| 2.4 Методе..... | 15 |
| 2.5 Конструктори..... | 34 |
| 2.6 Глобалне променљиве и глобалне методе (резервисана реч STATIC)..... | 37 |
| 2.7 Константе (резервисана реч FINAL)..... | 39 |
| 2.8 Релације..... | 39 |
| 3 Наредбе за контролу тока извршавања програма..... | 46 |
| 3.1 IF наредба..... | 46 |
| 3.2 SWITCH наредба..... | 54 |
| 3.3 FOR наредба..... | 56 |
| 3.4 WHILE наредба..... | 62 |
| 3.5 DO-WHILE наредба..... | 64 |
| 4 Низови..... | 66 |
| 4.1 Вишедимензионални низови..... | 71 |
| 5 Класа String..... | 74 |
| 6 Наслеђивање, апстрактне класе, интерфејси и класа Object..... | 81 |
| 6.1 Наслеђивање | 81 |
| 6.2 Апстрактне класе..... | 89 |
| 6.3 Интерфејси..... | 91 |
| 6.4 Класа Object..... | 94 |
| 7 Нивои приступа, пакети и JavaBeans спецификација..... | 98 |
| 7.1 Нивои приступа..... | 98 |
| 7.2 JavaBeans спецификација..... | 100 |
| 7.3 Пакети..... | 101 |
| 8 Низови објекта..... | 105 |
| 9 Листе..... | 109 |
| 10 Изузети..... | 114 |
| 11 Литература..... | 123 |

1 Програмски језик Јава - основне информације

Јава је објектно-оријентисани програмски језик развијен од стране Sun Microsystems корпорације, а сад је у власништву Oracle корпорације. Овај језик је бесплатан за коришћење и може се преузети са сајта Oracle корпорације (<https://www.oracle.com/java/index.html>). Једна од карактеристика Јаве је да је то платформски независан језик. То значи да се, коришћењем Јаве, могу писати програми за Windows, Linux или скоро било који други оперативни систем. Јава виртуелна машина (“Java Virtual Machine” - JBM) омогућава да се програмски код написан у Јави преведе у извршне инструкције за одговарајући оперативни систем (1). На овај начин, један исти Јава програм може да се, потпуно неизмењен, користи на било ком рачунару са било којим оперативним системом. Другим речима, да би се Јава уопште могла користити, потребно је инсталаријати JBM на рачунар.



Слика 1: Јава Виртуелна Машина

Сама чињеница да је Јава бесплатна, је довела до тога да он постане изузетно популаран и широко рас прострањен програмски језик. Међутим, највећа моћ Јаве је управо у томе што пружа подршку за писање свих врста апликација (програма).

JSE (“Java Standard Edition”) је скуп Јава библиотека која омогућава писање тзв. десктоп апликација. Да би се десктоп апликација могла користити, потребно ју је инсталаријати на рачунар. То значи да се на сваком рачунару мора налазити по једна копија апликације. На пример, Microsoft Word и Microsoft Excel су десктоп апликације. Ове апликације најчешће користе графички интерфејс оперативног система рачунара на којем се извршавају (прозоре, меније, дугмиће итд.).

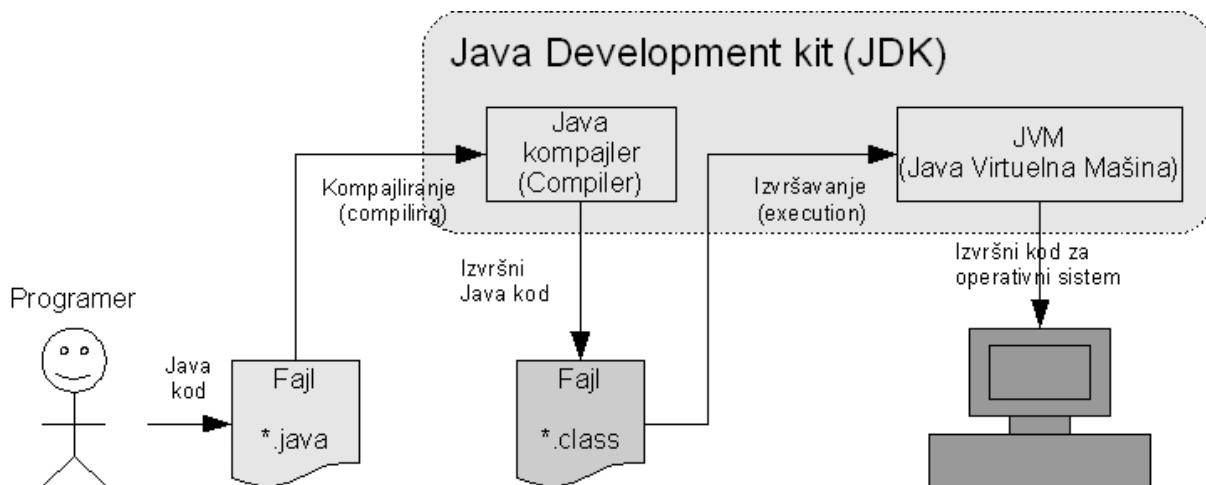
JEE (“Java Enterprise Edition”) је скуп Јава библиотека која омогућава писање тзв. enterprise апликација. Једна од основних карактеристика enterprise апликација је дистрибуираност. То

најчешће подразумева да једну копију (или евентуално неколико копија) апликације истовремено дели и користи више корисника, па инсталација на сваки рачунар није потребна. На пример, FaceBook и Google су ентерприсе апликације. Велики број ових апликација има интерфејс у виду интернет стране па им се приступа преко интернет читача ("browser").

JME ("Java Micro Edition") је скуп Јава библиотека која омогућава писање апликација за мобилне уређаје - мобилне телефоне, PDA итд.

JSE, JEE и JME се заједничким именом називају Јава издања (дистрибуције), јер се објављују у форми фајла који се може преузети са сајта Oracle корпорације. У основи, свако издање садржи JBM и одговарајуће библиотеке. За потребе упознавања са програмским језиком Јава, потребно је инсталирати JSE-JDK издање Јаве ("Java Standard Edition - Java Development Kit") која се може преузети са сајта Oracle корпорације.

Основни процес креирања програма у Јави се може видети на следећој слици (Слика 1). Програмер прво напише Јава код који онда сачува у фајлу са екstenзијом "java". Следећи корак је да се тај основни код компајлира тј. претвори у Јава извршни код. Компајлер врши овај корак по позиву и прави фајлове са екstenзијом "class". Тек онда се програм може покренути ("run"). Када се покрене, тек онда JBM претвара Јава извршни код у извршни код за оперативни систем и програм починje да се извршава.



Слика 1: Процес креирања, компајлирања и извршавања Јава програма

JDK садржи само неке основне библиотеке потребне за креирање, компајлирање и извршавање Јава програма. Он, међутим, не садржи никакав едиторе нити радно окружење па је писање Јава програма без тих додатака изузетно тешко и непрактично. Због тога постоји посебна класа програма који користе JDK или га надограђују комплетним графичким интерфејсом и многим другим алатима, а све то са циљем лакшег и бржег креирања Јава програма. Општи назив за овакав програм је интегрисано развојно окружење ("Integrated Development Environment" - IDE). Неки од најпопуларнијих бесплатних IDE за Јаву су:

- Eclipse (званична интернет адреса: <http://www.eclipse.org/>)
- NetBeans (званична интернет адреса: <http://www.netbeans.org/>)
- BlueJ (званична интернет адреса: <http://www.bluej.org/>)

Тек када се, поред JDK, инсталира и неки IDE, може да се почне са писањем Јава програма.

I ДЕО – ОСНОВЕ ПРОГРАМСКОГ ЈЕЗИКА JAVA

2 Класе, објекти и њихови елементи

У овом поглављу се обрађују основни концепти објектно-оријентисаног програмирања и њихова имплементација у оквиру програмског језика Јава. Прво, објашњава се појам класе и атрибута. Затим се уводи појам објекта и прави се јасно разграничење између објекта и класе. Методе и конструктори, као елементи класе који дефинишу понашање, се објашњавају у наставку, док се глобалне променљиве, глобалне методе и константе уводе након тога. Релације се обрађују као последња тема у оквиру овог поглавља.

2.1 Класе

Класа је општи представник неког скupa објеката (предмета или појава) који имају исту структуру и понашање. Класа је упрошћена слика ових реалних предмета и појава и обухвата њихове:

- карактеристике (атрибуте)
- понашања (методе)
- односе са другим класама (релације)

Атрибути, методе и релације се називају **елементи (чланице) класе**.

Пример 1

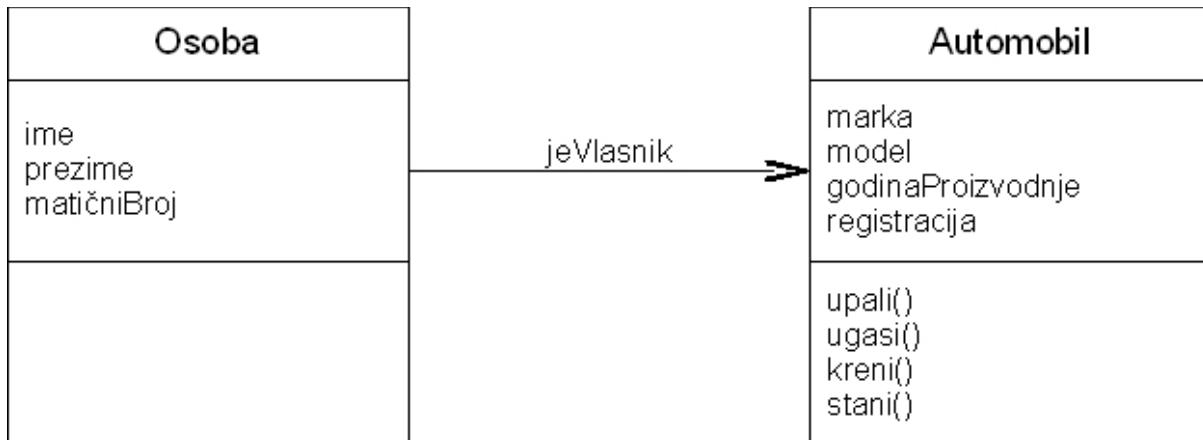
Пример класе је Аутомобил (Слика 2). Аутомобил има следеће карактеристике: марку, модел, годину производње и регистрациони број. Нека од понашања аутомобила су: упали, угаси, крени, стани. На слици се може видети да су ове карактеристике представљене преко атрибута, док су понашања представљена коришћењем метода. Битно је приметити да је ово упрошћена слика реалног аутомобила јер аутомобил може имати и низ других карактеристика (серийски број, боја...) и понашања (скрени, отвори врате...).

| Automobil |
|---|
| marka model godinaProizvodnje registracija |
| upali() ugasi() kreni() stani() |

Слика 2: Пример класе - Класа
Аутомобил

Пример 2

Други пример класе може да буде Особа. Речимо да особа има име, презиме и матични број, а да, у овом примеру, нема дефинисано никакво понашање. Ако се узме у обзир класа Аутомобил из претходног примера и класа Особа, између њих постоји одређени однос. Особа може да буде власник аута. Овај однос се представља путем релације између класа (Слика 3).



Слика 3: Пример релације између две класе

Детаљи у вези са атрибутима, методама и релацијама ће бити објашњени у оквиру наредних поглавља. Сада је битно видети како се врши дефиниција класе у Јави.

Општи облик дефиниције класе у Јави је следећи:

```

class NazivKlase {
    //definicije atributa...
    //definicije metoda...
}

```

Дефиниција почиње резервисаном речи “`class`”, после које иде назив класе. Овај први ред је **заглавље класе**. Следећи елемент је отворена витичаста заграда. После витичасте заграде почиње **тело класе** које садржи дефиниције атрибути и метода. Дефиниција класе се завршава затвореном витичастом заградом.

Јава је програмски језик у коме се **прави разлика између великих и малих слова** („case sensitive“). То значи да се називи класа, атрибути, метода могу разликовати у зависности од тога да ли се напишу великим или малим словима. На пример, “КонверторВалуте”, “конверторвалуте” и “КОНВЕРТОРВАЛУТЕ” ће да се третирају као потпуно различити називи.

Тренутно, у оквиру тела класе која је дата као пример се налазе само два коментара. Уопштено гледано, **коментари** служе томе да се напише нека порука особи која ће читати или мењати изворни код. То може бити неко објашњење или напомена који помажу лакшем разумевању написаног кода. При компајлирању, Јава компајлер игнорише коментаре и не тумачи их као програмски код, па садржај коментара може бити било какав текст.

Коментари се у Јави могу писати на два начина. Једну варијанту представљају кратки,

једнолинијски коментари. Почетак оваквог коментара се означава дуплом косом цртом, а цео текст мора да стане у један ред. Пример једнолинијског коментара је:

```
//definicije atributa...
```

Друга врста коментара су **вишелинијски коментари**. Они се пишу у случају да је потребно дати неко детаљније или дуже објашњење. Наравно, у том случају је могуће написати више једнолинијских коментара, али је много једноставније цео текст коментара уоквирити као један вишелинијски коментар. Вишелинијски коментар почиње косом цртом и звездом ("/*") а завршава се звездом и косом цртом ("*/"):

```
/* Ovo je  
višelinijjski  
komentar */
```

Пример 3

Направити (дефинисати) Класу Аутомобил. Ова класа не би требало да има ниједан атрибут нити методу. Написати ову чињеницу у форми једнолинијског коментара.

```
class Automobil {  
    //Ova klasa nema nijedan atribut niti metodu.  
}
```

Пример 4

Направити Класу Особа. Ова класа не би требало да има ниједан атрибут нити методу. Написати ову чињеницу у форми вишелинијског коментара.

```
class Osoba {  
    /* Ova klasa nema nijedan  
       atribut niti metodu. */  
}
```

Потребно је напоменути да се у објектно-оријентисаним програмским језицима, у принципу, поштује **неписано правило да се називи класа пишу са првим великим словом** (“Аутомобил” а не “аутомобил”). Ако се назив класе састоји из више речи, онда се све речи пишу спојено а прво слово сваке речи је велико (“КонверторВалуте”, “ПословниЦентар”). Назив **не сме да има ниједан бланко знак** (ово важи и за називе свих других елемената - атрибута, метода...).

Обично се изворни **код сваке класе чува у посебном фајлу**. Тако, изворни код класе “Automobil” може да буде сачуван у фајлу “Automobil.java”, а класе “Osoba” у фајлу “Osoba.java”. Када се компајлира, Јава извршни код ће да буде сачуван у фајловима “Automobil.class” и “Osoba.class”. Међутим, **могуће је сачувати изворни код више класа у једном фајлу**. Другим речима, изворни код класа “Automobil” и “Osoba” би могао бити сачуван у једном фајлу нпр. “Automobil.java”. Али, **компајлирањем би се опет добила два фајла**: “Automobil.class” и “Osoba.class”.

2.2 Атрибути

Као што је већ речено, **атрибути** представљају неке карактеристике (особине) класа. За класу Особа, то може бити име, презиме, пол, старост итд. Ове особине се најчешће могу изразити путем неког броја, слова или низа слова.

Дефинисање атрибута је релативно једноставно и изгледа овако:

```
tip_podatka nazivAtributa;
```

Тип податка представља скуп могућих вредности атрибута - цео број, реалан, број, слово, низ слова или нешто друго. Према неписаним правилама, **називи атрибута би требало да починju малим словом** (“старост”, “марка”, “модел” итд). Ако се назив атрибута састоји од више речи, све речи се пишу спојено. Прва реч почиње малим словом, а све остale великом (‘матичниБрој’, ‘броДрота’ итд). Дефиниција се завршава тачка-зарезом.

Једно јако важно правило за писање команди у Јави је да се **све команде морају завршити знаком тачка-зарез**. Неки најчешћи коришћени **типови података** у Јави су:

| Назив типа податка | Опис | Примери |
|--------------------|---|--|
| int | цели бројеви | 1, -55, 0, 100000 |
| double | реални бројеви | 12.55, -234.77, 0.21 |
| char | знак (слово, цифра или неки други знак) | 'a', 'A', 'e', '!', ';', ''(бланко знак), '4', '9' |
| boolean | логичка променљива | true, false (само ове две вредности) |
| String | низ знакова | “Река”, “Пера”, “123”, “зх!” |
| Calendar | датум и време | 2008-12-31 10:50 |

Типови int, double, char и boolean су **прости типови података**, док су String и Calendar **сложени типови**. String је низ више знакова, док Calendar садржи дан, месец, годину, сат, минут итд. Сви сложени типови података у Јави се представљају коришћењем класа, па су String и Calendar заправо две предефинисане Јава класе. Потребно је напоменути да постоје и још неки прости типови података али да се не користе толико често: long (скуп целих бројева чији је распон већи него код int типа), short (скуп целих бројева чији је распон мањи него код int типа) и float (скуп реалних бројева, али са мање децималних места него код double типа).

Такође, у Јави се **char вредности (знакови) морају писати под једноструким знацима навода (апострофима)**, док се **String вредности обавезно пишу под двоструким знацима навода**.

Пример 5

Направити Класу АутоматНовца. Ова класа би требало да има само атрибут “станje” који представља износ новца који се тренутно налази у аутомату (реалан број).

```
class AutomatNovca {
    double stanje;
```

```
}
```

Пример 6

Направити Класу Рачунар. Ова класа би требало да има следеће атрибуте: такт процесора (реалан број, нпр. 4.0 GHz), радна меморија (реалан број, нпр. 2.0 Gb), хард диск (цео број, нпр. 120 Gb).

```
class Racunar {  
  
    double taktProcesora;  
    double radnaMemorija;  
    int hardDisk;  
  
}
```

Редослед у којем се дефинишу атрибути у оквиру класе није битан. Атрибутима је могуће доделити и неку подразумевану, почетну вредност одмах при дефинисању класе. То се ради на следећи начин:

```
tip_podatka nazivAtributa = vrednost;
```

Пример 7

Преправити класу Рачунар из претходног примера тако да почетна вредност за такт процесора буде 4.0 (GHz), радна меморија износи 2.0 (Gb) а хард диск има 120 (Gb).

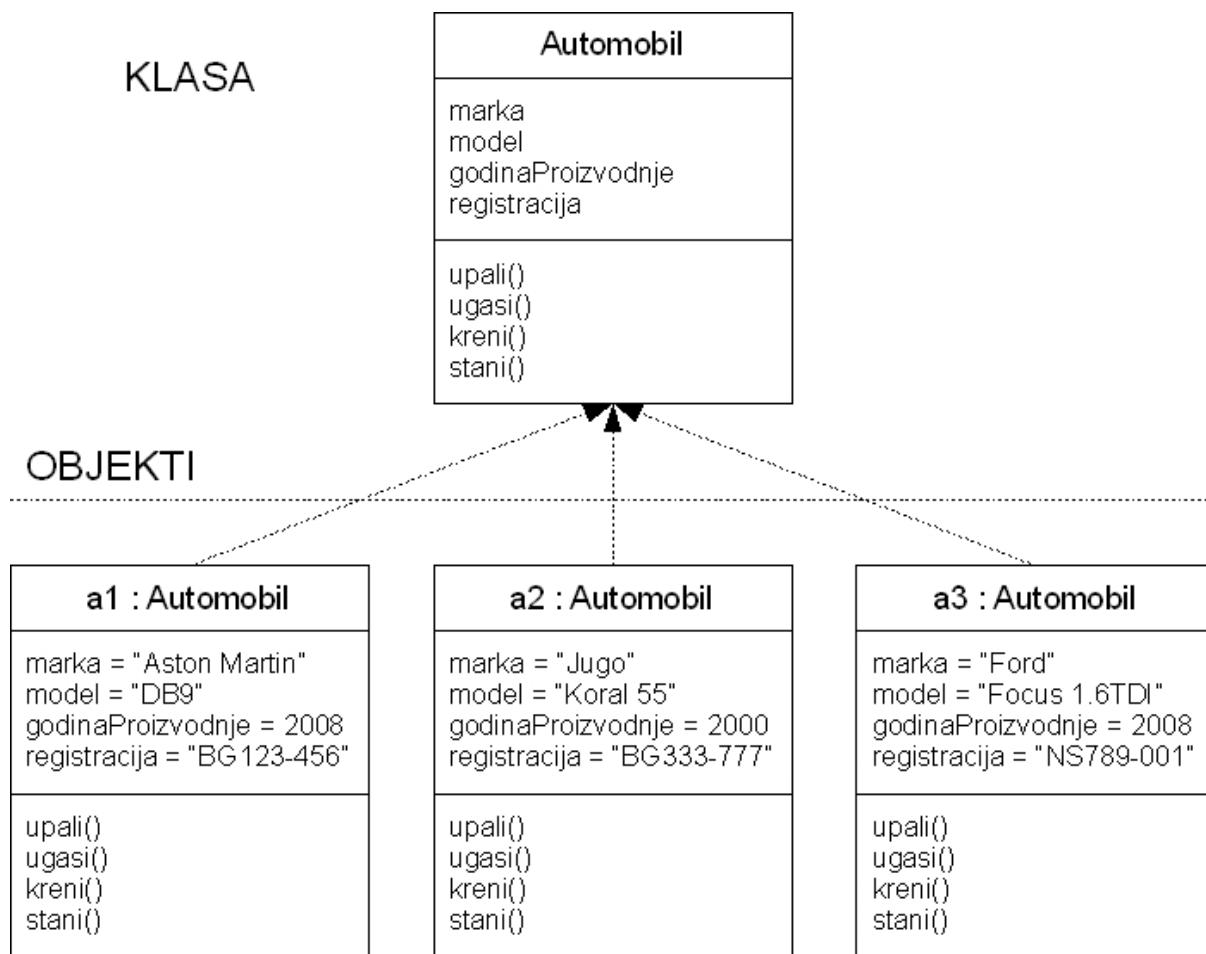
```
class Racunar {  
  
    double taktProcesora = 4.0;  
    double radnaMemorija = 2.0;  
    int hardDisk = 120;  
  
}
```

2.3 Објекти

Објекат представља један конкретан примерак (инстанцу) или појављивање неке класе. Према томе, класа може да се дефинише као скуп објеката који имају исте особине. Однос класе и објекта је приказан у следећем примеру.

Пример 8

Аутомобил представља класу јер је то описи нацрт неких карактеристика и понашања које сваки аутомобил има. Међутим, "Aston Martin DB9" са годином производње 2008 и регистрацијом "БГ123-456" представља један конкретан примерак аутомобила тј. објекат класе аутомобил. Однос класе и објекта је приказан на следећој слици (Слика 4).



Слика 4: Однос класе и објекта

У Јави, објекти се декларишу на сличан начин као и атрибути класе. Прво се наведе назив класе па онда назив конкретног објекта. На овај начин се ствара променљива која ће да референцира конкретан објекат:

```
NazivKlase nazivobjekta;
```

Међутим, **да би објекат могао да се користи (да се позивају његове методе, мењају вредности атрибута итд.), потребно га је иницијализовати**. Ако се објекту покуша приступити без иницијализације, Јава јавља грешку. Иницијализација се врши коришћењем наредбе “new” на следећи начин:

```
nazivobjekta = new NazivKlase();
```

Шта се заправо дешава када се иницијализује објекат? Наиме, када се декларише објекат (променљива), само се створи показивач који ће да референцира објекат. Али, када се изврши иницијализација, тек се онда алоцира део меморије рачунара (део RAM меморије) у коме ће да буде објекат и повеже се са показивачем - показивач тада садржи адресу објекта у меморији (Слика 5). Тек сада објекат може да се користи.

ImeKlase objekat1;

objekat1 = new ImeKlase();

objekat1

null

Promenljiva objekat1 ima null vrednost
jer objekat nije inicijalizovan. Objekat
još uvek ne može da se koristi.

objekat1

30

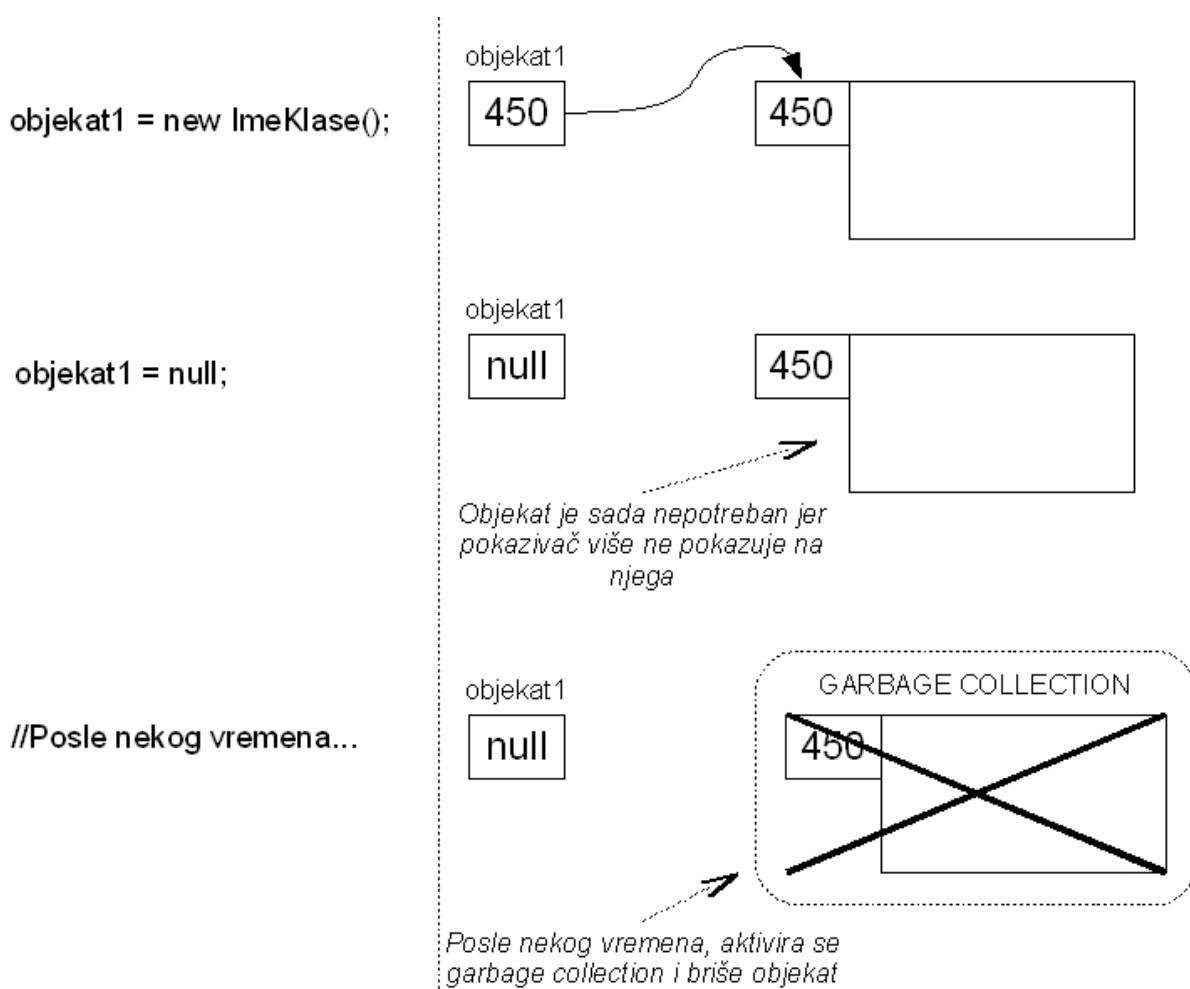
30

Promenljiva objekat1 sada sadrži
memorijsku adresu 30

Objekat je inicijalizovan i može da
se koristi (nalazi se na
memorijskoj adresi 30)

Слика 5: Иницијализација објекта

Било би занимљиво објаснити и како се бришу објекти у Јави. Потпуно је нормална ситуација да се у току извршавања неког озбиљнијег програма за кратко време створи (иницијализује) на стотине или хиљаде објеката. Сваки од тих објеката заузима неки део меморије рачунара, а често се дешава да многи од њих уопште више нису потребни, па би их требало обрисати тј. избацити из меморије. Ако се ово не уради, може доћи до преоптеређења меморије. Јава има механизам који се зове “garbage collection” (сакупљање смећа) који врши аутоматско брисање непотребних објеката. Објекат је непотребан ако на њега не показује ни једна променљива нити показивач. Другим речима, доволно је променљивој (показивачу) доделити нулл вредност или иницијализовати нови објекат преко исте променљиве па да стари објекат буде аутоматски избрисан после неког времена (Слика 6).



Слика 6: Брисање објекта у Јави ("гарбаже коллеџион")

Да би неки Јава програм могао да се покрене, потребно је да има тзв. "main" методу¹.
Заглавље ове методе је увек исто, а њена дефиниција се пише у оквиру тела класе на следећи начин:

```
public static void main (String[] args) {
    //naredbe...
}
```

Пример 9

Направити класу Мотоцикл која има:

- Атрибут маркаИМодел.
- Атрибут кубикажа (цео број).

Направити класу Тест која садржи main методу и, у оквиру ње, прави два објекта класе Мотоцикл.

¹ Ниједан Јава пример нити задатак који је приказан до сад не може да се покрене (изврши), јер нема "main" методу, али може да се компајлира.

```
class Motocikl {  
  
    String markaIModel;  
    int kubikaza;  
  
}  
class Test {  
  
    public static void main(String[] args) {  
        Motocikl m1;  
        Motocikl m2;  
  
        m1 = new Motocikl();  
        m2 = new Motocikl();  
    }  
  
}
```

Објекат је појављивање класе које има конкретне вредности атрибута. Да би се вредности атрибута промениле или прочитале, потребно им је приступити на одређен начин. Приступ атрибутима објекта се врши преко назива објекта и назива атрибута међусобно одвојених тачком:

```
nazivobjekta.nazivAtributa
```

Пример 10

Искористити класу *Мотоцикл* из претходног примера и преправити класу *Тест* која садржи *майн* методу тако да прави два објекта класе *Мотоцикл*. Први би требало да буде "Suzuki GS" од 500 кубика (доделити атрибутима првог објекта ове вредности) а други "Yamaha R6" од 600 кубика (доделити атрибутима другог објекта ове вредности).

```
//Kod za klasu Motocikl ostaje isti  
  
class Test {  
  
    public static void main(String[] args) {  
        Motocikl m1;  
        Motocikl m2;  
  
        m1 = new Motocikl();  
        m2 = new Motocikl();  
  
        m1.markaIModel = "Suzuki GS";  
        m1.kubikaza = 500;  
  
        m2.markaIModel = "Yamaha R6";  
        m2.kubikaza = 600;  
    }  
}
```

У Јави се команде извршавају у редоследу у којем су написане ("одозго на доле"). Ако се, на пример, у оквиру *майн* методе прво напишу команде за приступ атрибутима, а тек после следи команда за иницијализацију објекта, Јава ће да пријави грешку.

Претходна два примера садрже main методу па могу да се изврше. Међутим, резултат њиховог извршавања неће да буде ништа видљиво - објекти ће да се иницијализују, доделиће се вредности атрибутима, али ништа неће да се испише на екрану. Команда за исписивање на екрану (**у Јави се то зове “стандардни излаз”**) је следећа:

```
System.out.println( ...neki tekst i/ili vrednost... );
```

Резултат извршавања ове команде је испис (на екрану) садржаја који се налази између заграда и прелазак у нови ред. Тада садржај може да буде неки текст, вредност неког атрибута (или променљиве) или комбинација ова два. У следећем примеру је приказано шта ова команда тачно ради и како се користи.

Пример 11

Следи неколико примера употребе команде “`println`” за испис на екрану и резултата који ће бити приказани:

```
System.out.println("Lep dan danas");
//Ispisace se na ekranu:
//Lep dan danas

int broj;
broj = 12;
System.out.println(broj);
//Ispisace se na ekranu:
//12

System.out.println("Ova recenica ide u prvi red");
System.out.println("Ova recenica ide u drugi red");
//Ispisace se na ekranu:
//Ova recenica ide u prvi red
//Ova recenica ide u drugi red

int temperatuta;
temperatuta = 21;
System.out.println("Temperatura je: "+temperatuta);
//Uloga znaka plus (+) je da omoguci nadovezivanje vrednosti
//temperature na poruku
//Ispisace se na ekranu:
//Temperatura je: 21
```

Пример 12

Искористити класу Motocikl из претходних примера и класу Test која садржи main методу и прави два објекта класе Motocikl - “Suzuki GS” од 500 кубика и “Yamaha R6” од 600 кубика. Преправити main методу класе Test тако да исписује вредности атрибута оба објекта на екрану.

```
//Kod za klasu Motocikl ostaje isti

class Test {

    public static void main(String[] args) {
        Motocikl m1;
```

```
Motocikl m2;

m1 = new Motocikl();
m2 = new Motocikl();

m1.markaIModel = "Suzuki GS";
m1.kubikaza = 500;

m2.markaIModel = "Yamaha R6";
m2.kubikaza = 600;

System.out.println(m1.markaIModel);
System.out.println(m1.kubikaza);

System.out.println(m2.markaIModel);
System.out.println(m2.kubikaza);
//Ispisace se na ekranu:
//Suzuki GS
//500
//Yamaha R6
//600
}

}
```

Једна од варијанти наредбе за испис на екрану је и “print” наредба. Једина разлика у односу на “println” (“print-line”) наредбу је та што се после исписа текста не прелази у следећи ред већ се и даље исписује у постојећем реду.

```
System.out.print( ...neki tekst i/ili vrednost... );
```

Пример 13

Следи неколико примера употребе команде “print” за испис на екрану и резултата који ће бити приказани:

```
System.out.print("Rec1");
System.out.print("Rec2");
//Ispisace se na ekranu:
//Rec1Rec2

int temperatura;
temperatura = 21;
System.out.print("Temperatura je: "+temperatura+" stepen ");
System.out.print("u Beogradu");
//Ispisace se na ekranu:
//Temperatura je 21 stepen u Beogradu
```

2.4 Методе

Већ је речено да класе могу имати и одређена понашања и да се та понашања изражавају у форми **метода**. На пример, методе класе Аутомобил, могу бити: упали, угаси, крени, стани итд.

Дефиниција метода у Јави се врши у оквиру тела класе и то на следећи начин:

```
tip_povratne_vrednosti nazivMetode( ...parametri... ) {  
    //Telo metode  
}
```

Тип повратне вредности методе, њен назив и параметри чине **заглавље методе (потпис методе)**, док сваки код написан између витичастих заграда чини **тело методе**. Тело методе почиње отвореном витичастом заградом а завршава се затвореном витичастом заградом.

Методама се дефинише понашање класе. Поред тога што нешто ради, метода може и да **враћа неку вредност као резултат извршавања (повратна вредност методе)**. На пример, метода "сабери" класе "Дигитрон", може да врати број који представља резултат сабирања. Са друге стране, неке методе немају повратну вредност. Пример једне такве методе би била метода "испишиИме" класе "Особа" која на екрану само исписује име особе. **Када метода не враћа никакву повратну вредност, тип повратне вредности је "void", а када враћа тип повратне вредности може бити било који прост или сложен тип података (int, double, char, boolean, String...).** Метода може да има **само једну повратну вредност**.

Према неписаним правилу, **називи метода се пишу на исти начин као називи атрибута (прва реч почиње малим словом а, ако има више речи, остале почињу великим словом)**: "сабери", "одузми", "испишиИме", "конвертујВалуту"...

Конечно, методе могу да имају и неке улазне вредности - **параметре**. Параметри представљају оне вредности које је потребно проследити методи да би она могла да се правилно изврши. На пример, метода "сабери" класе "Дигитрон" може да, као параметре, има два броја која је потребно сабрати. Ако метода нема параметре, простор између заграда се оставља празан (пише се само "()"). Параметри ће детаљно бити објашњени у даљем тексту.

Пример 14

Направити класу Телевизор. Ова класа би требало да има:

- Атрибут *јачинаТона* који је цео број и означава тренутну јачину тона на телевизору. Почетна вредност овог атрибута је 0 (тон је утишан до краја).
- Атрибут *тренутниПрограм* који означава број програма који је тренутно на телевизору (нпр. укључен је програм 5). Почетна вредност овог атрибута је 1.
- Атрибут *укључен* који означава да ли је телевизор укључен или није (ако је укључен има вредност *TRUE*, иначе има вредност *FALSE*). Сматра се да је на почетку телевизор искључен.
- Методу *укључи* која укључује телевизор (поставља вредност атрибута укључен на *TRUE*).
- Методу *искључи* која искључује телевизор (поставља вредност атрибута укључен на *FALSE*).

```
class Televizor {  
  
    int jacijaTona = 0;  
    int trenutniProgram = 1;  
    boolean ukljucen = false;  
  
    void ukljuci() {  
        ukljucen = true;
```

```
}

void iskljuci() {
    ukljucen = false;
}
}
```

У овом примеру, класа *Телевизор* има само две методе - укључи и искључи. Метода укључи нема повратну вредност јер само мења вредност атрибута укључен. Према томе, тип повратне вредности ове методе је *void*. Ове методе нема параметре јер нису потребни (увек се вредност атрибута поставља на *true*) па је простор између обичних заграда празан. Тело методе укључи садржи само једну команду којом се атрибуту укључен додељује вредност *true*. Све наведено важи и за методу искључи.

Из претходног примера се може приметити и једно правило: **атрибутима класе се из тела било које методе те исте класе приступа само преко назива**. Другим речима, није потребно наводити назив објекта па тачка па назив атрибута као када се приступа из *main* методе.

Пример 15

Додати у класу *Телевизор* и следеће методе:

- Методу *појачајТон* која повећава вредност атрибута *јачинаТона* за један.
- Методу *смањиТон* која смањује вредност атрибута *јачинаТона* за један.
- Методу *искључиТон* која потпуно утишава тон (смањује вредност *јачине тона* на 0).

После измене, код класе изгледа овако:

```
class Televizor {

    int jacijaTona = 0;
    int trenutniProgram = 1;
    boolean ukljucen = false;

    void ukljuci() {
        ukljucen = true;
    }

    void iskljuci() {
        ukljucen = false;
    }

    void pojacajTon() {
        jacijaTona = jacijaTona + 1;
    }

    void smanjiTon() {
        jacijaTona = jacijaTona - 1;
    }

    void iskljuciTon() {
        jacijaTona = 0;
    }
}
```

Методе *појачајТон*, *смањиТон* и *искључиТон* немају повратну вредност (ништа не враћају као резултат), па је тип повратне вредности *void*. Ове методе немају параметре.

Методе које су наведене у досадашњим примерима не враћају никакву вредност као резултат. Међутим, постоје ситуације у којима је то потребно урадити - некада је потребно вратити тренутну вредност атрибута неког објекта или вратити резултат неке рачунске операције. У тим случајевима, се као тип повратне вредности уместо void пише тип вредности податка који ће бити враћен - ако метода враћа цео број - пише се int, ако метода враћа реалан број - пише се double, ако враћа низ слова - пише се String итд. Поред тога, у оквиру самог тела методе **се мора написати команда “return”** да означи вредност коју је потребно вратити. Овде је потребно дати и једну напомену: **у тренутку када се команда “return” изврши, извршавање методе се прекида**. Другим речима, било која команда написана тако да се безусловно извршава након команде “return” се никад неће извршити.

Пример 16

Додати у класу Телевизор и следеће методе:

- Методу промениПрограмНавише која повећава вредност атрибута тренутниПрограм за један.
- Методу промениПрограмНаниже која смањује вредност атрибута тренутниПрограм за један.
- Методу вратиТренутниПрограм која враћа вредност атрибута тренутниПрограм.
- Методу вратиЈачинуТона која враћа тренутну вредност атрибута јачинаТона.
- Методу далиЈеУкључен која враћа тренутну вредност атрибута укључен.
- Методу испишиПараметре која исписује на екрану тренутне вредности свих атрибута телевизора уз одговарајућу поруку.

После свих измена, програмски код класе изгледа овако:

```
class Televizor {

    int jacijaTona = 0;
    int trenutniProgram = 1;
    boolean ukljucen = false;

    void ukljuci() {
        ukljucen = true;
    }

    void iskljuci() {
        ukljucen = false;
    }

    void pojacaJTon() {
        jacijaTona = jacijaTona + 1;
    }

    void smanjiTon() {
        jacijaTona = jacijaTona - 1;
    }

    void iskljuciTon() {
        jacijaTona = 0;
    }

    void promeniProgramNavise() {
        trenutniProgram = trenutniProgram + 1;
    }
}
```

```
void promeniProgramNanize() {
    trenutniProgram = trenutniProgram - 1;
}

int vratiTrenutniProgram() {
    return trenutniProgram;
}

int vratiJacinuTona() {
    return jaciaTona;
}

boolean daLiJeUkljucen() {
    return ukljucen;
}

void ispisiParametre() {
    System.out.println("Jacija tona je "+jaciaTona);
    System.out.println("Trenutni program je "+trenutniProgram);
    System.out.println("Televizor je ukljucen "+ukljucen);
}

}
```

Методе `promeniProgramNaviše` и `promeniProgramNaniže` немају никакве повратне вредности. Међутим, методе `vratiTrenutniProgram`, `vratiJacinuTona` и `daLiJeUkljucen` имају. Метода `vratiTrenutniProgram` би требало да врати тренутну вредност атрибута програм. Практично гледано, то би значило да врати број програма који се тренутно приказује на телевизору (на пример број 5). Пошто је у питању цео број (атрибут `trenutniProgram` је типа `int`), и повратна вредност методе ће бити `int`. Тело методе `vratiTrenutniProgram` садржи само команду `return` којом се означава вредност коју је потребно вратити.

Примера ради, рецимо да је у оквиру тела ове методе написана команда:

```
return 2;
```

Метода би у овом случају увек враћала број 2. Метода `vratiJacinuTona` је веома слична претходној методи и доволно је рећи да и она враћа `int` вредност. Али, метода `daLiJeUkljucen` враћа вредност атрибута укључен, па је тип повратне вредности `boolean`.

Конечно, метода `ispisiParametre` само врши испис вредности свих атрибута на екрану уз одговарајућу поруку. Потребно је напоменути да исписивање на екрану није исто што и враћање повратних вредности. Вредности које су исписане на екрану се не могу користити касније (за рачунање или неке друге операције), па се сматра да метода `ispisiParametre` заправо не враћа никакву вредност. Тип повратне вредности ове методе је, због тога, `void`.

Поставља се питање, а како се методе позивају? Одговор је сличан као и за атрибуте. Да би методе могле да се позову, потребно је направити објекат, па тек онда позвати неку његову методу преко назива класе и назива методе. Разлика у односу на позив атрибута је у томе што се при позиву методе после назива методе обавезно морају написати и заграде (чак иако метода нема параметре):

```
nazivObjekta.nazivMetode();
```

Ако метода има повратну вредност, прво је потребно декларисати променљиву која ће да прими повратну вредност методе, па тек онда позвати методу. **Тип променљиве мора да буде исти као тип повратне вредности методе:**

```
tip_promenljive naziv_promenljive;
naziv_promenljive = nazivobjekta.nazivMetode();
```

Пре него што се пређе на конкретан пример, потребно је разјаснити шта је променљива и које врсте променљивих постоје. **Променљива** представља неки идентификатор (име, слово, израз исл.) који је повезан са неком вредношћу, при чему се та вредност може мењати. Променљиве су физички репрезентоване као место у меморији рачунара у које се може сместити нека вредност. Тип променљиве одређује тип вредности која се може унети у променљиву. На пример, променљива типа int може да садржи неку целобројну вредност а физички је представљена као део меморије рачунара у који се може сместити нека целобројна вредност. У зависности од тога где се декларишу у коду и како се могу позвати, постоје следеће врсте променљивих²:

- **Локалне променљиве (или само променљиве)** - променљиве декларисане у оквиру тела методе.
- **Атрибути** - променљиве које директно припадају објектима неке класе. Дефинишу се у оквиру тела те класе.
- **Глобалне променљиве** - променљиве које не припадају појединачним објектима неке класе већ се користе на нивоу целог програма. Дефинишу се на сличан начин као и атрибути³.
- **Параметри** - у суштини, и параметри метода су променљиве. Декларишу се у заглављу методе.

Пример 17

Искористити класу Телевизор из претходног примера. Направити класу TestТелевизор која креира један објекат класе Телевизор и позива неке од његових метода. После сваког позива методе, позвати методу испишиПараметре и уочити промене у вредностима атрибута.

```
class TestTelevizor {

    public static void main (String[] args) {
        Televizor t = new Televizor();
        boolean uklj;
        int prog;

        t.ispisiParametre();
        //Ispisace se na ekranu
        //Jacina tona je 0
        //Trenutni program je 1
        //Televizor je ukljucen false

        t.ukljeni();
        t.ispisiParametre();
```

2 Појам "видљивости" променљивих који је уско повезан са овом поделом се обрађује у наредних пар страна.

3 О глобалним променљивим и глобалним методама ће бити више речи у једном од наредних подпоглавља.

```
//Ispisace se na ekranu
//Jicina tona je 0
//Trenutni program je 1
//Televizor je ukljucen true

t.pojacajTon();
t.ispisiParametre();
//Ispisace se na ekranu
//Jicina tona je 1
//Trenutni program je 1
//Televizor je ukljucen true

t.promeniProgramNavise();
t.ispisiParametre();
//Ispisace se na ekranu
//Jicina tona je 1
//Trenutni program je 2
//Televizor je ukljucen true

uklj = t.daLiJeUkljucen();
System.out.println("Televizor je ukljucen: "+uklj);
//Ispisace se na ekranu
//Televizor je ukljucen: true

prog = t.vratiTrenutniProgram();
System.out.println("Na televizoru ide program "+prog);
//Ispisace se na ekranu
//Na televizoru ide program 2

t.ispisiParametre();
//Ispisace se na ekranu
//Jicina tona je 1
//Trenutni program je 2
//Televizor je ukljucen true
}

}
```

Када се позива нека метода која нема повратну вредност (на пример метода `испишиПараметре`), позив је сличан као за атрибуте објекта. Једина разлика је у томе што се после назива методе пишу и заграде у којима се наводе конкретне вредности за параметре. Ако метода нема параметре, простор између заграда остаје празан ("()"):

```
t.ispisiParametre();
```

Позив методе која има неку повратну вредност је мало другачији. То је случај са методама `даЛиЈеУкључен` и `вратитренутниПрограм`. У примеру се види да је декларисана променљива "укљ" која је типа `boolean` (исто као тип повратне вредности методе `даЛиЈеУкључен`) и која прима повратну вредност методе:

```
uklj = t.daLiJeUkljucen();
```

На овај начин, повратна вредност методе се не губи (сачувана је у променљивој "укљ"), и може се користити у даљем извршењу програма. То илуструје следећа линија кода из примера. Променљива "укљ" се позива у оквиру команде за испис на екрану у циљу исписивања податка о томе да ли је телевизор тренутно укључен.

У многим досадашњим примерима и задацима је коришћен један оператор. У питању је **оператор за доделу вредности**. Формални опис и пар примера у вези са овим оператором су дати у следећој табели.

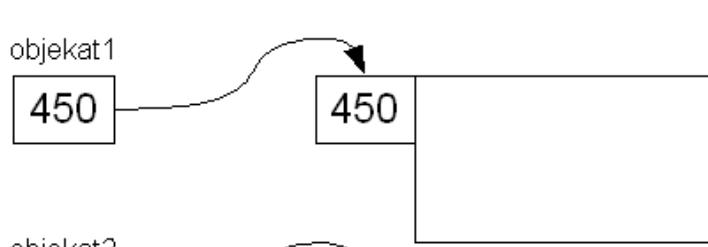
| <i>Оператор за доделу вредности</i> | | |
|-------------------------------------|------------------------------|---|
| <i>Оператор</i> | <i>Опис</i> | <i>Примери</i> |
| = | Оператор за доделу вредности | a = 2; m1.kubikaza = 750; osoba1.ime = "Pera"; ind = obj.metoda1(); objekat1 = obekat2; |

У суштини, овај оператор омогућава да се некој променљивој, атрибуту или објекту додели нека вредност. Увек се ономе што се налази на левој страни знака једнакости додељује оно што се налази на десној страни знака једнакости. На десној страни знака једнакости може да буде нека конкретна вредност, променљива, атрибут, неки израз, позив методе или објекат. Једино је битно да **оно што је са леве стране знака једнакости буде истог типа као и оно са десне стране** (нпр. није могуће доделити вредност 23.44 променљивој int типа).

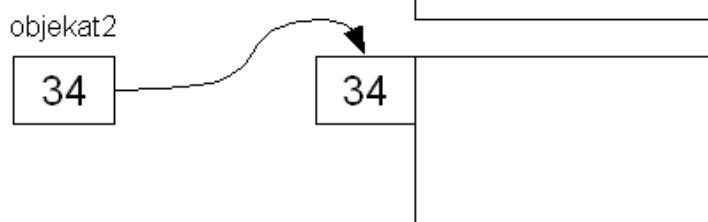
Када се оператор за додељивање вредности користи у комбинацији са атрибутима, променљивим и методама који садржи или враћају вредности простог типа, ствар је релативно једноставна. Левој страни се увек додељује вредност са десне стране. Међутим, ствари су мало компликованије ако су операнди показивачи на објекте (Слика 7). Нека је први операнд показивач на објекат (променљива "објекат1") и нека је други операнд такође показивач на неки објекат (променљива "објекат2"). Оператор додељивања додељује садржај променљиве са десне стране променљивој на левој страни. У овом случају то значи да ће се **адреса коју садржи променљива "објекат2" уписати у променљиву "објекат1" па ће оба показивача показивати на исту меморијску локацију тј. исти објекат**. Меморијска локација на коју више не показује ниједан показивач ће убрзо да буде ослобођења преко "garbage collection" механизма.

Додатни ефекат који се у овом случају појављује је тај да се сада преко оба показивача може приступити истој меморијској локацији (истом објекту). Ово се може видети и на једном практичном примеру.

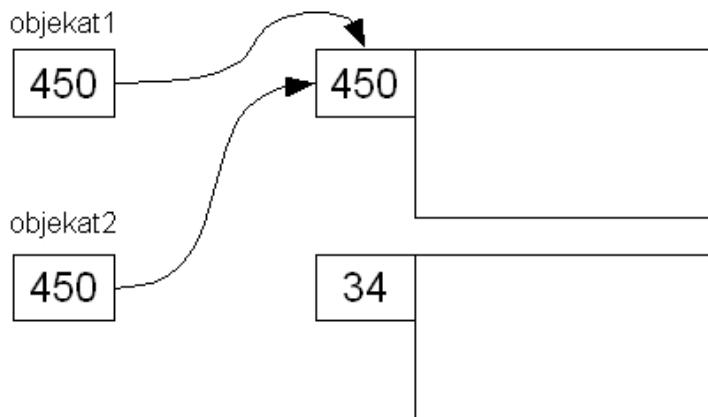
```
objekat1 = new ImeKlase();
```



```
objekat2 = new ImeKlase();
```



```
objekat2 = objekat1;
```



Слика 7: Додељивање вредности објеката

Пример 18

Написати класу Производ која има само један атрибут - назив и једну методу "испиши" која исписује вредност овог атрибута на екрану. Направити класу ТестПроизвод која креира два објекта класе Производ са називима "автомобил" и "трактор". Исписати вредности атрибута оба производа. Доделити другом објекту адресу првог објекта. Поново исписати вредности атрибута оба производа. Изменити назив другог производа у "комбајн" и још једном исписати вредности атрибута оба производа.

```
class Proizvod {  
    String naziv;  
    void ispisiSve(){  
        System.out.println("Naziv je: "+naziv);  
    }  
}  
  
class TestProizvod {
```

```

public static void main(String[] args) {
    Proizvod p1 = new Proizvod();
    Proizvod p2 = new Proizvod();

    p1.naziv = "automobil";
    p2.naziv = "traktor";

    p1.ispisIsve();
    p2.ispisIsve();
    //Ispisace se na ekranu
    //Naziv je: automobil
    //Naziv je: traktor

    p2=p1;

    p1.ispisIsve();
    p2.ispisIsve();
    //Ispisace se na ekranu
    //Naziv je: automobil
    //Naziv je: automobil

    p2.naziv="kombajn";

    p1.ispisIsve();
    p2.ispisIsve();
    //Ispisace se na ekranu
    //Naziv je: kombajn
    //Naziv je: kombajn
}
}

```

У тренутку када се показивачу на објектам `p2` додели садржај показивача `p1`, оба показивача ће да се референцирају на исти део меморије тј. на исти објект. Од тада, све промене које се ураде над објектом на који показује `p2` ће бити видљиве и код објекта на који показује `p1` и обрнуто.

Параметри методе представљају неке вредности које је потребно проследити методи да би она могла правилно да се изврши. Метода може да има нула, један или више параметара, а параметри се декларишу у оквиру заглавља методе на сличан начин као и променљиве (прво тип вредности, па назив параметра). Овако декларисани параметар се назива **формални параметар (или само параметар)**. Ако метода има више параметара, они се одвајају зарезом:

```

tip_povratne_vrednosti nazivMetode(tip_parametra parametar) {

    //Telo metode

}

```

Позивање метода које имају параметре се врши на тај начин што се између заграда наведу неке конкретне вредности или променљиве. Када се при позиву методе као параметри проследе ове конкретне вредности или променљиве, оне се називају **аргументи (реални тј. стварни параметри)**. У оквиру позива овакве методе мора да се испоштује број параметара, њихов тип и редослед. Другим речима, ако метода прима два параметра неког типа, сваки позив те методе мора да буде такав да се методи заиста и проследе тачно два аргумента који одговарају

овим параметрима по типу и редоследу.

```
nazivMetode(argument);
```

Пример 19

Направити класу АутоматНовца. Ова класа би требало да има:

- атрибут стање који представља тренутни износ новца у аутомату (реалан број). Почетна вредност овог атрибута је 5200.0 динара.
- Методу подигниИзнос која прима као параметар износ новца који корисник жели да подигне (реалан број нпр. 550.5) и смањује вредност атрибута стање за тај износ.
- Методу уложиИзнос која прима као параметар износ новца који корисник жели да уложи (реалан број) и повећава вредност атрибута стање за тај износ.
- Методу вратиСтање која враћа тренутни износ новца у аутомату (вредност атрибута стање).
- Методу испишиСтање која на екрану исписује која је тренутна количина новца у аутомату (вредност атрибута стање).

Направити класу ПровераАутоматаНовца која креира два објекта класе АутоматНовца. У први аутомат новца је потребно уложити 1002.03 динара и исписати стање пре и после улагања. Потребно је и подићи 234.55 динара из другог аутомата и исписати стање аутомата пре и после улагања.

```
class AutomatNovca {  
  
    double stanje=5200.0;  
  
    // "iznos" je (formalni) parametar  
    void podigniIznos (double iznos){  
        stanje = stanje - iznos;  
    }  
  
    // I ovde je "iznos" parametar  
    void uloziIznos (double iznos){  
        stanje = stanje + iznos;  
    }  
  
    double vratiStanje (){  
        return stanje;  
    }  
  
    void ispisiStanje (){  
        System.out.println("Trenutni iznos u automatu je: "+stanje);  
    }  
}  
  
class ProveraAutomataNovca {  
  
    public static void main (String[] args){  
        AutomatNovca a1 = new AutomatNovca();  
        AutomatNovca a2 = new AutomatNovca();  
  
        a1.ispisiStanje();  
        // Vrednost 1002.03 je argument (realni parametar)  
        // koji metoda dobija pri pozivu.  
        a1.ulozIznos(1002.03);
```

```

a1.ispisStanje();
//Ispisace se na ekranu
//Trenutni iznos u automatu je: 5200.0
//Trenutni iznos u automatu je: 6202.03

a2.ispisStanje();
//Vrednost 234.55 je argument (realni parametar)
//koji metoda dobija pri pozivu.
a2.podigniIznos(234.55);
a2.ispisStanje();
//Ispisace se na ekranu
//Trenutni iznos u automatu je: 5200.0
//Trenutni iznos u automatu je: 4965.45
}
}

```

Методе подигниИзнос и уложиИзнос имају по један параметар. У оба случаја, у питању су реални бројеви, па је тип параметра доубле. Ово су формални параметри јер се дефинишу у оквиру заглавља методе. Назив параметра је увек произволjan, само је битно обратити пажњу на то да се тај назив користи у оквиру тела методе када се параметар жели употребити. Ове две методе немају повратну вредност јер само мењају вредност атрибута стање.

У оквиру “main” методе класе ПровераАутоматаНовца се заправо види разлика између параметара (формалних параметара) и аргументата (реалних параметара). Формални параметар методе уложиИзнос је “износ” који је типа “double”, док је аргумент конкретан број или променљива који се прослеђују методи при pozиву (у овом случају број 1002.03).

У програмском језику Јава је могуће писати и разне **аритметичке изразе**. Неки од претходних примера већ садрже примере коришћења оператора за сабирање и одузимање, али се комплетна листа аритметичких оператора налази у следећој табели.

| Аритметички оператори | | |
|-----------------------|---|---|
| Оператор | Опис | Примери |
| + | Сабирање | $a + b$ $12 + 5$ <code>m1.kubikaza + 300</code> |
| - | Одузимање | $a - b$ $12 - 5$ <code>m1.kubikaza - 300</code> |
| * | Множење | $a * b$ $12 * 5$ <code>m1.kubikaza * 300</code> |
| / (цели бројеви) | Целобројно дељење (DIV) | $2 / 12$ a / b |
| / (realni brojevi) | Дељење | $a / 12.0$ $5.0 / 123.44$ |
| % | Остатак при целобројном дељењу (MOD) | $5 \% 2$ $a \% 10$ |
| ++ | Увећање за 1 (инкрементација) | <code>a++</code> <code>m1.kubikaza++</code> |
| -- | Умањење за 1 (декрементација) | <code>a--</code> <code>m1.kubikaza--</code> |
| += | Увећање вредности променљиве за неку вредност | <code>a += 3</code> |
| -= | Умањење вредности променљиве за неку вредност | <code>a -= 3</code> |
| *= | Множење вредности променљиве неком вредношћу | <code>a *= 3</code> |
| /= | Дељење вредности променљиве неком вредношћу | <code>a /= 3</code> |
| %= | Остатак при дељењу вредности променљиве неком вредношћу | <code>a %= 3</code> |

Оператори за сабирање, множење и одузимање се користе на сличан начин као што би се користили у математици. Ови оператори могу да се користе и са целобројним и реалним вредностима и променљивима. Јава омогућава и коришћење заграда па су уобичајени и изрази као:

```
b * 2 * ((234.55 + 20) - (a * 3.4))
```

Оператори за инкрементацију и декрементацију увећавају тј. смањују вредност неке целобројне променљиве за један. Они пружају опцију скраћеног записа одређених аритметичких израза, па су следећа два пара израза потпуно еквивалентни:

```
a = a + 1;  
b = b - 1;
```

```
//Je isto sto i...
a++;
b--;
```

И оператори “`+ =`”, “`- =`”, “`* =`”, “`/ =`” и “`% =`” су уведени ради могућности краћег записа оних израза у којима се нова вредност променљиве израчунава на основу њене постојеће вредности. Према томе, следећа два скупа израза су еквивалентна:

```
a += 3;
a -= 4;
a *= 5;
a /= 6;
a %= 7;

//Je isto sto i...

a = a + 3;
a = a - 4;
a = a * 5;
a = a / 6;
a = a % 7;
```

Оператори дељења су у Јави мало сложенији него што се на први поглед може учинити. Прво, и оператор за дељење реалних бројева и оператор за целобројно дељење се означавају истим знаком - косом цртом (“`/`”). Дељење реалних бројева је “нормално дељење” (на пример $10.0 / 3 = 3.33333$). Целобројно дељење, са друге стране, представља дељење без остатка ($10 / 3 = 3$). При овом дељењу се одсеца децимални део (део који се налази иза зареза) и као резултат се враћа целобројни део. Са обзиром на то да се оба оператора исто пишу, поставља се питање како Јава компајлер разазнаје када је потребно употребити један, а када други оператор. Одговор је веома једноставан - врши се провера да ли су операнди (бројеви тј. променљиве који се деле) цели или реални и примењује се одговарајући оператор. Ако су оба броја цела, примењује се целобројно дељење. Ако је макар један од бројева реалан, примењује се дељење реалних бројева.

Оператор за израчунавање остатка при целобројном дељењу се може користити искључиво са целобројним вредностима. Резултат који се добије представља цео број који остаје као вишак при дељењу два цела броја.

Операторе дељења и остатка при дељењу је најбоље објаснити преко неколико једноставних примера.

Пример 20

Следећи изрази ће у Јави бити протумачени као изрази дељења реалних бројева:

```
double x = 3.5;
double y = 7;
double z;
z = y / x;
//Rezultat je 2.0
//I x i y su realni brojevi.
```

```
z = y / 7;  
//Rezultat je 1.0  
//Dovoljno je da je jedan od brojeva realan - y  
  
z = 35.0 / 7;  
//Rezultat je 5.0  
//Dovoljno je da je jedan od brojeva realan - 35.0  
  
z = 35 / 7.0;  
//Rezultat je 5.0  
//Dovoljno je da je jedan od brojeva realan - 7.0
```

Међутим, ови изрази ће бити протумачени као целобројно дељење без остатка:

```
int a = 2;  
int b = 12;  
int c;  
double r;  
  
c = b / a;  
//Rezultat je 6 (ne 6.0 nego само 6)  
//I a i b su celobrojne promenljive  
  
c = b / 4;  
//Rezultat je 3  
//b je celobrojna promenljiva  
  
c = 123 / 12;  
//Rezultat je 10 (odseca se deo iza zareza)  
//I 123 i 12 su celi brojevi  
  
r = 123 / 12;  
//Rezultat je 10.0 (rezultat 10 se pretvara u realan broj 10.0)  
//I 123 i 12 su celi brojevi pa je pozvano celobrojno deljenje
```

Како што се из последњег примера види, могућа је ситуација и да се због непажње, при дељењу добију неки неочекивани резултати. Узрок је увек то што се, због погрешно написаног израза, позове оператор за дељење целих бројева уместо оператора за дељење реалних бројева. Да би претходни израз био протумачен као дељење реалних бројева, потребно је написати га на један од следећих начина:

```
r = 123.0 / 12;  
  
r = 123.0 / 12.0;  
  
r = 123 / 12.0;  
  
//Rezultat je 10.25  
//Bitno je da je makar jedan od brojeva realan
```

Ево и неколико примера коришћења оператора за израчунавање остатка при целобројном дељењу:

```
c = 5 % 2;  
//Rezultat je 1 (jer je 5 = 2*2 + 1)
```

```
c = 10 % 2;
//Резултат је 0 (јер је 10 = 5*2 и нema остатка)
c = 14 % 4;
//Резултат је 2 (јер је 14 = 3*4 + 2)

c = 55 % 5;
//Резултат је 0 (јер је 55 = 11*5 и нema остатка)
```

Овај оператор може да се користи да се провери да ли је неки број паран јер ће остатак при дељењу бројем два увек бити нула ако јесте паран, а један ако није паран. Овако се може проверити и дељивост са неким другим бројевима (остатак је нула ако је број дељив а има неку другу вредност ако није).

Да би се у потпуности схватио начин функционисања метода, потребно је увести и објаснити појмове **видљивости променљивих** и **начина преноса параметара методе**.

Сви параметри методе и променљиве које се декларишу у оквиру тела методе (локалне променљиве) су “видљиви” (могу се позвати) искључиво у оквиру тела те методе и никде друге. Сви атрибути класе су увек видљиви у оквиру тела методе те класе.

Пример 21

Нека класа Калкулатор има две методе - сабери и обимКруга. Прва метода добија два цела броја као параметре и враћа резултат сабирања, док друга прима полуупречник круга као параметар и враћа обим круга као резултат. Класа има и атрибут π који представља вредност одговарајуће математичке константе.

```
class Kalkulator {

    double pi = 3.14;

    int saberi(int a, int b) {
        int rezultat;
        rezultat = a + b;
        return rezultat;
    }

    double obimKruga(double poluprecnik) {
        double rezultat;
        rezultat = 2 * poluprecnik * pi;
        return rezultat;
    }
}
```

Пи је атрибут класе Калкулатор, па се без проблема може позвати и користити у оквиру тела било које методе класе Калкулатор (користи се у оквиру тела методе обимКруга). Променљиве a и b су, са друге стране, параметри прве методе. То значи да се они могу користити и позивати искључиво у оквиру тела ове методе. Ако би се покушало са позивањем променљивих a и b из тела методе обимКруга, јавила би се грешка. Исто важи и за параметар полуупречник - “видљив” је само у оквиру тела методе обимКруга и никде више.

Површиним прегледањем кода би се могло закључити да у коду постоји конфликт јер се декларишу две променљиве са називом резултат - једна у оквиру тела прве а друга у оквиру тела друге методе. То, међутим, није тако. У Јави се ове две променљиве посматрају као потпуно различите јер променљива резултат из методе сабери има видљивост само у

оквиру тела методе сабери, док променљива резултат из методе обимКруга има видљивост само у оквиру тела методе обимКруга.

Конфлікт у “видљивости” параметара и атрибута се може јавити ако се у оквиру тела методе користи параметар или променљива који имају исти назив као и неки атрибут класе. У том случају се користи **резервисана реч “this”** да издвоји атрибут у односу на друге променљиве. Ова резервисана реч је у ствари показивач објекта на самог себе.

Пример 22

Нека класа Особа има атрибут име и методу унесиИме која као параметар прима неко име и атрибуту додељује ту вредност. Нека се параметар ове методе такође зове “име”

```
class Osoba {  
  
    String ime;  
  
    void unesiIme(String ime) {  
        ime = ime;  
    }  
  
}
```

Прво што се може приметити је да је команда додељивања вредности атрибуту име веома нејасна. Шта је атрибут, а шта параметар? И атрибут се назива “име” а и параметар. Према томе, који је ефекат извршавања наредбе:

```
ime = ime;
```

Одговор је у томе да ће Јава да протумачи обе стране као параметар, тако да се додељивање вредности атрибуту име уопште неће ни десити. Да би метода радила оно што је потребно, користи се резервисана реч “this” да означи атрибут. После преправки, метода би требало да изгледа овако:

```
void unesiIme(String ime) {  
    this.ime = ime;  
}
```

Израз са леве стране знака једнакости се односи на атрибут име, док се израз са десне стране знака једнакости односи на параметар име.

Параметри метода могу бити прости типови података (int, double, boolean, char) али и сложени типови (класе). Другим речима, метода као параметар може имати и неки објекат. Поред тога, метода може као резултат да врати неки објекат.

Када се, при позиву, методи прослеђују аргументи простог типа, врши се **пренос параметра преко вредности**. То значи да се вредност прослеђеног аргумента копира. Сваки пут када се у оквиру тела методе позове параметар, користи се заправо копија унетог аргумента. Због тога се **промене извршене над параметром у току извршавања методе неће запамтити (јер су промене вршене над копијом, а не над оригиналом)**. Када се метода заврши, копија се брише.

Ако се методи при позиву проследи објекат као аргумент, врши се **пренос параметра преко адресе**. Прослеђивање објекта као аргумента се врши навођењем одговарајућег показивача на

објекат. И у овом случају се садржај аргумента копира па се, са обзиром на то да је аргумент показивач, прави копија адресе објекта у меморији. Када се у оквиру тела методе позове параметар, приступа се објекту на адреси која је претходно прекопирана, а то је заправо оригинални објекат. Последица оваквог приступа је да се промене над објектом унетим као аргумент извршене у оквиру тела методе памте и после краја извршавања методе (јер се промене врше над оригиналном).

Пример 23

Нека је дата класа *Бројеви* која има атрибуте *a* и *b* који су цели бројеви. Нека је дата и класа *ЗаменаБројева* која им две методе. Прва метода прима као параметре два цела броја и покушава да им замени места. Друга метода као параметар прима објекат класе *Бројеви* и мења места вредностима атрибута *a* и *b*. Класа *ТестЗаменаБројева* позива обе методе класе *ЗаменаБројева* и исписује резултате на екрану.

```
class Brojevi {
    int a;
    int b;
}

class Zamenabrojeva {
    void zameniMesta(int a, int b) {
        int pomocna;
        pomocna = a;
        a = b;
        b = pomocna;
    }

    void zameniMesta2(Brojevi br) {
        int pomocna;
        pomocna = br.a;
        br.a = br.b;
        br.b = pomocna;
    }
}

class TestZamenabrojeva {
    public static void main(String[] args) {
        int a = 10;
        int b = 12;

        Brojevi br = new Brojevi();
        br.a = 44;
        br.b = 100;

        Zamenabrojeva zb = new Zamenabrojeva();

        System.out.println ("Pre promene a = "+a+" b = "+b);
        zb.zameniMesta(a, b);
        System.out.println ("Posle promene a = "+a+" b = "+b);
        //Ispisace se na ekranu
        //Pre promene a = 10 b = 12
    }
}
```

```
//Posle promene a = 10 b = 12

System.out.println ("Pre promene br.a = "+br.a+" br.b =
"+br.b);
    zameniMesta2(br);
    System.out.println ("Posle promene br.a = "+br.a+" br.b =
"+br.b);
        //Ispisace se na ekranu
        //Pre promene br.a = 44 br.b = 100
        //Posle promene br.a = 100 br.b = 44
    }

}
```

Када се изврши прва метода, бројеви неће бити замењени јер су унети као прости параметри и пренети преко вредности. Међутим, позивом методе замениМеста2, замена вредности атрибута a и b ће да се изврши и остаће упамћена јер је цео објекат пренет као параметар преко адресе.

Називи атрибута у оквиру једне класе морају да буду јединствени - не могу се појавити два атрибута са истим називом. Са методама, то није случај. Наиме, могуће је написати две или више метода са истим називом у оквиру једне класе. Али, да би се разликовале, свака од њих мора имати другачију листу параметара - било да је у питању број параметара или њихов тип. При позиву методе, позваће се она метода чији параметри тачно одговарају унетим вредностима. Ова појава се назива **преклапање метода** ("overloading").

Пример 24

Направити класу Калкулатор која има две методе са истим називом - сабери. Прва метода прима два цела броја као параметре и враћа њихов збир (цео број). Друга метода прима два реална броја као параметре и враћа њихов збир (реалан број). Написати и класу ТестКалкулатор која позива обе методе.

```
class Kalkulator {

    int saberi (int x, int y){
        int rezultat = x+y;
        return rezultat;
    }

    double saberi (double x, double y){
        double rezultat = x+y;
        return rezultat;
    }
}

class TestKalkulator {

    public static void main(String[] args) {
        Kalkulator k = new Kalkulator();

        double r = k.saberi(12.0, 15.0);
        System.out.println(r);
        //Ispisace se na ekranu
        //27.0
    }
}
```

```

int r2 = k.saberi(12, 15);
System.out.println(r2);
//Ispisace se na ekranu
//27
}
}

```

Први позив методе `saberi` ће да активира методу која врши сабирање реалних бројева и враћа реалан број као резултат. Други позив методе `saberi` ће да активира методу која врши сабирање целих бројева и враћа цео број као резултат. У првом случају су унети реални бројеви као параметри а у другом цели бројеви.

2.5 Конструктори

Сваки пут када је потребно иницијализовати неки објекат користи се команда “`new`”. После ове резервисане речи увек следи назив класе па отворена и затворена заграда. Већ је објашњено да се при иницијализацији резервише простор у меморији рачунара у који ће бити смештен нови објекат. Међутим, после резервисања меморијског простора, дешава се још једна ствар: врши се позив конструктора. **Конструктор** представља посебну методу помоћу које се може извршити додељивање почетних вредности атрибутима објекта приликом његове иницијализације. Значи, конструктор класе се **автоматски позива сваки пут када се позове команда “new”**. Дефиниција конструктора је скоро иста као и дефиниција методе, само уз два ограничења:

- **Назив конструктора је увек исти као и назив класе** (ако се назове другачије, Јава ту методу неће сматрати за конструктор).
- Конструктори **никада немају повратну вредност**, па се у оквиру заглавља не пише чак ни “`void`”.

Поред тога, **класа може имати и више конструктора**, али се онда они морају разликовати по параметрима (број и/или типови параметара). Ово је преклапање конструктора и слично је преклапању метода.

У досадашњим примерима се нису експлицитно писали конструктори класе. Када нека класа у Јави **нема експлицитно написан конструктор, Јава јој аутоматски додељује подразумевани (“default”) конструктор који не мења вредности атрибута нити врши било какве друге видљиве промене.**

Али, у неким ситуацијама је погодно написати конструктор јер се одмах, при иницијализацији класе, могу доделити неке почетне вредности атрибутима, може се нешто исписати на екрану итд.

Конструктори се деле на две врсте у зависности од тога да ли имају параметре или не. Прву врсту чине **параметризовани** конструктори, а другу **подразумевани (“default”)** конструктори. Један подразумевани конструктор је приказан у следећем примеру.

Пример 25

Направити класу `ПрехрамбениАртикал` која има два атрибута: назив и калоријска вредност.

Првом атрибуту додељити почетну вредност "непознат" а другом 0.0.

Направити класу `PrehrambeniArtikal2` која има два атрибута: назив и калоријска вредност. Направити конструктор који првом атрибуту додељује почетну вредност "непознат" а другом 0.0.

```
class PrehrambeniArtikal {  
  
    String naziv = "nepoznat";  
    double kalorijskaVrednost = 0.0;  
  
}  
  
class PrehrambeniArtikal2 {  
  
    String naziv;  
    double kalorijskaVrednost;  
  
    PrehrambeniArtikal2(){  
        naziv = "nepoznat";  
        kalorijskaVrednost = 0.0;  
    }  
  
}
```

Класа `PrehrambeniArtikal` нема конструктор, па се почетна вредност атрибутима додељује одмах при дефинисању. Супротно томе, класа `PrehrambeniArtikal2` има подразумевани конструктор (нема параметре) који служи да се атрибутима додели почетна вредност. Прво, потребно је приметити да конструктор, за разлику од обичне методе, нема повратну вредност - не пише се чак ни "void" него одмах назив. Друго, назив конструктора је потпуно исти као и назив класе (`PrehrambeniArtikal2` - прво слово сваке речи је велико). На крају, види се да је ефекат извршења конструктора потпуно исти као да се атрибутима доделила почетна вредност одмах при дефинисању.

Све ово значи да се при иницијализацији објекта п класе `PrehrambeniArtikal` позива подразумевани конструктор који је Јава аутоматски доделила класи (овај конструктор не ради ништа), а атрибутима се додељује вредност преко дефиниције атрибута.

```
PrehrambeniArtikal p = new PrehrambeniArtikal();
```

При иницијализацији објекта п2 класе `PrehrambeniArtikal2` се позива експлицитно написани подразумевани конструктор који додељује вредности атрибутима.

```
PrehrambeniArtikal2 p2 = new PrehrambeniArtikal2();
```

Сврха конструктора је да, при креирању, доведе објекат у почетно стање тј. да га иницијализује. Међутим, у неким случајевима је потребно да конструктор уради и нешто више од самог додељивања почетних вредности атрибутима. То илуструје следећи пример.

Пример 26

Направити класу `PrehrambeniArtikal3` која има два атрибута: назив и калоријска вредност. Направити конструктор који првом атрибуту додељује почетну вредност "непознат" а другом 0.0 и исписује поруку о почетним вредностима атрибута на екрану.

```
class PrehrambeniArtikal3 {

    String naziv;
    double kalorijskaVrednost;

    PrehrambeniArtikal3() {
        naziv = "nepoznat";
        kalorijskaVrednost = 0.0;
        System.out.println("Naziv: "+naziv);
        System.out.println("Kalorijska vrednost: "+kalorijskaVrednost);

    }

}
```

Конструктор класе *ПрехрамбениАртикал3* врши и исписивање неких порука на екрану. Другим речима, сваки пут када се неки објекат ове класе иницијализује на екрану ће се исписати почетне вредности његових атрибута:

```
PrehrambeniArtikal3 p3 = new PrehrambeniArtikal3();
//Ispisace se na ekranu
//Naziv: nepoznat
//Kalorijska vrednost: 0.0
```

Подразумевани конструктори увек постављају атрибуте на исте почетне вредности. Параметризовани конструктори, са друге стране, могу да поставе атрибуте на вредности које су унете као параметри приликом креирања објекта. На овај начин се преко само једног позива конструктора могу истовремено доделити вредности једном или више атрибута. То се може видети у следећем примеру.

Пример 27

Направити класу *Особа* која има два атрибута: име и презиме. Направити конструктор који има два параметра од којих први садржи вредност за име, а други вредност за презиме. Овај конструктор би требало да додељује унете вредности атрибутима. Направити и класу *TestOsoba* у којој се креира објекат класе *Особа* и одмах му се, преко конструктора, додељује име “Пера” и презиме “Перић”.

```
class Osoba {

    String ime;
    String prezime;

    Osoba(String imel, String prezimel) {
        ime = imel;
        prezime = prezimel;
    }

}

class TestOsoba {

    public static void main(String[] args) {
        //Ova komanda nece biti prihvacena jer
        //klasa Osoba ima jedan konstruktor a on
        //ima parametre
    }
}
```

```
//Osoba o1 = new Osoba();  
//Ovako se poziva parametrizovani konstruktor klase Osoba  
Osoba o1 = new Osoba("Pera", "Peric");  
}  
}
```

2.6 Глобалне променљиве и глобалне методе (резервисана реч STATIC)

Објекти имају атрибуте и методе који се могу позвати и користити тек када се објекат иницијализује. Такође, сваки објекат има своје, засебне вредности атрибута. У неким ситуацијама је, међутим, корисно да више објекта (из истих или различитих класа) дели једну променљиву. Такве променљиве су видљиве на нивоу целог програма, па се зову **глобалне променљиве**. У Јави се глобалне променљиве **означавају резервисаном речи "static"** и пишу се у оквиру тела класе као атрибути класе:

```
static tip_vrednosti nazivPromenljive;
```

Због тога што глобална променљива није везана ни за један објекат, коришћење ове променљиве не подразумева иницијализацију објекта класе којој припада већ јој се приступа директно преко назива класе у оквиру које је дефинисана:

```
NazivKlase.nazivPromenljive
```

Пример 28

Направити класу Хотел и у оквиру ње дефинисати глобалну променљиву преосталиБројСоба (почетна вредност је 100). Направити класу ТестХотел која мења вредност ове глобалне променљиве.

```
class Hotel {  
  
    static int preostaliBrojSoba = 100;  
  
}  
  
class TestHotel {  
  
    public static void main(String[] args) {  
        //Nije potrebna inicijalizacija objekta klase  
        //Hotel da bi se koristila globalna promenljiva  
        //vec se poziv vrsti preko naziva klase.  
        Hotel.preostaliBrojSoba = 5;  
  
        System.out.println("Kapacitet je: "  
                           +Hotel.preostaliBrojSoba);  
    }  
}
```

Понекад постоји и потреба за методама које пружају неку општу функционалност која није стриктно везана за неку класу. Овакве методе су често добри кандидати да постану **глобалне**

методе. Глобалне методе се у Јави означавају резервисаном речи “static” па се често зову и **статичке методе.** Ове методе се дефинишу у оквиру тела класе на исти начин као и било која друга метода:

```
static tip_vrednosti nazivMetode(...parametri...) {
    //Telo metode
}
```

Разлика у односу на обичну методу је та што се не мора инцијализовати објекат те класе да би се метода користила, већ се позив врши преко назива класе:

```
NazivKlase.nazivMetode(argumenti);
```

У ствари, ако се мало боље погледа, може се видети да је и наредба “System.out.println” позив статичке методе јер није потребно направити објекат класе System да би се позвала принтлн метода.

Main метода је такође статичка метода. Ово је неопходно јер се позива на почетку рада програма пре него што је било који објекат направљен.

Пример 29

Направити класу Калкулатор и у оквиру ње дефинисати две глобалне методе: сабери и одузми. Обе методе примају два цела броја као параметре, а враћају резултат операције сабирања тј. одузимања. Направити класу ТестКалкулатор која позива ове две методе.

```
class Kalkulator {
    static int saberi(int a, int b) {
        return a+b;
    }

    static int oduzmi(int a, int b) {
        return a-b;
    }
}

class TestKalkulator {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;

        int rezultat1 = Kalkulator.saberi(a, b);
        int rezultat2 = Kalkulator.oduzmi(a, b);

        System.out.println(rezultat1);
        System.out.println(rezultat2);
    }
}
```

При коришћењу статичких метода треба имати на уму и то да статичке методе **не могу позивати не-статичке елементе класе у којој су дефинисане** (обичне атрибуте и методе).

2.7 Константе (резервисана реч FINAL)

Уколико је потребно дефинисати неку константу у оквиру Јава класе, користи се **резервисана реч “final”**. Константе се дефинишу као атрибути, са тим што се испред типа атрибута стави реч “final”:

```
final tip_vrednosti NAZIV_CONSTANTE = vrednost;
```

Према неписаним правилу, **називи константи се пишу свим великим словима** (“ПИ”, “КАМАТА”). Ако назив константе има више речи, речи се одвајају знаком доње црте (“МИН_КАМАТА”). Константама се **морају доделити вредности одмах при дефиницији** јер то касније није могуће.

Позивање константе се врши на исти начин као и позивање атрибута класе. Једина разлика је у томе што константама **никада не може да се додели нова вредност**.

За неке константе је пожељно да буду глобално видљиве па се, поред речи “final”, означавају и резервисаном речи “static”. Позивање ових константи је исто као и позивање глобалних променљивих.

```
static final tip_vrednosti NAZIV_CONSTANTE = vrednost;
```

Резервисана реч “final” може да се користи и уз дефиниције класа и метода, али у тим случајевима има потпуно другачије значење. Ова врста употребе је ускo везана за наслеђивање класа, па ће бити детаљно објашњена у поглављу о наслеђивању.

2.8 Релације

Предмети и појаве из реалног света често формирају и неке односе. Класе, као упрошћене слике реалних предмета и појава, такође могу да формирају међусобне односе који рефлектују реалне односе. Односи између класа се називају **релацијама**. Као што објекти преузимају и користе атрибуте и методе класе којој припадају, тако преузимају и релације. Постоје две основне врсте релација:

1. **Асоцијација** и њени специјализовани облици:
 - (a) **Композиција - декомпозиција** (“HAS-A” релација, “PART-OF” релација)
 - (b) **Генерализација - специјализација** (“IS-A” релација, “наслеђивање”)
2. **Релација коришћења** (“USING” релација)

Асоцијација и њени специјализовани облици (композиција-декомпозиција и генерализација-специјализација) се **односе на структуру** јер успостављају структурне односе између класа, док се релација коришћења **односи на понашање**.

Асоцијација представља најопштији облик релације у коме објекти једне класе имају неку структурну везу или однос са објектима друге класе. Свака асоцијација се дефинише преко три

елемента:

- **Кардиналност**
- **Навигација (смер)**
- **Назив (улога)**

Кардиналност је број који означава колико објеката једне класе може да буде у релацији са једним објектом друге класе. Она одређује ограничења као: „једна поруџбина мора имати тачно једног купца“ или „фирма мора имати макар једног запосленог“. Кардиналност не мора да буде само један број већ може да буде распон, па према томе постоји **доња и горња граница кардиналности**. Најчешће коришћене кардиналности и њихово значење су дати у следећој табели.

| Кардиналност | Пример |
|----------------------|---|
| 1..1 (или само 1) | Једна поруџбина мора имати тачно једног купца (доња граница је 1 и горња граница је 1) |
| 0..1 | Купац на берзи може имати свог ексклузивног брокера, али не мора (доња граница 0, горња граница 1) |
| 0..* (или само *) | Осoba може бити власник једног или више аутомобила, али може и да не поседује аутомобил уопште (доња граница 0, горња граница „више“) |
| 1..* | Фирма мора да има макар једног запосленог, али их може имати и више (доња граница 1, горња граница „више“) |

Навигација (смер) асоцијације говори о томе како је асоцијација усмерена: од које класе ка којој класи. На пример, асоцијација „вози“ је усмерена од класе Особа ка класи Аутомобил. Према овом критеријуму, асоцијације се деле на **једносмерне и двосмерне** при чему су двосмерне асоцијације такве да могу да се посматрају у оба смера.

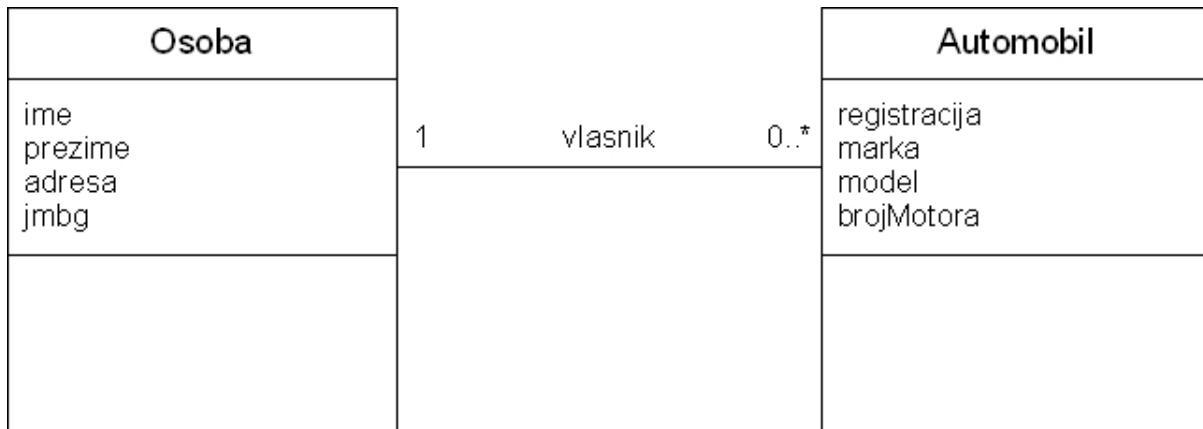
Свака асоцијација може да има и **назив (улогу)** који описује однос који асоцијација представља. Једносмерне асоцијације могу да имају само један назив, док двосмерне асоцијације могу имати два назива у односу на то у ком смеру се посматрају. На пример, асоцијација која повезује класу Фирма и класу Радник има назив „запошљава“ ако се посматра у смеру Фирма-Радник, и назив „је запослен у“ ако се посматра у смеру Радник-Фирма.

У Јави, **асоцијације се најчешће имплементирају као обични атрибути класе, са тим што је тип атрибута сама класа** - наводи се назив класе. У општем случају, имплементација зависи од кардиналности, па се тако асоцијације „више на према више“ могу имплементирати увођењем нове класе и сл. Асоцијације се графички представљају пуном линијом између две класе на којој може бити написана кардиналност и назив асоцијације. Ако је асоцијација једносмерна, ова линија може на крају имати стрелицу која упућује на смер асоцијације, док се код двосмерних асоцијација навигационе стрелице најчешће изостављају.

Пример 30

Пример асоцијације може да буде релација између класа Аутомобил и Особа у којој је особа власник возила (Слика 9). Прво, потребно је приметити да је ово двосмерна асоцијација (нема стрелица за навигацију). Друго, неопходно је обратити пажњу на кардиналност ове асоцијације. Одмах поред класе Аутомобил, на слици се налази ознака „0..*“. То значи да је доња граница кардиналности асоцијације у овом смеру нула („0“) а горња више („*“) - једна особа може бити власник више аутомобила (горња граница кардиналности), а не мора бити

власник ниједног аутомобила (доња граница кардиналности). Кардиналност у другом смеру асоцијације је 1 и значи да тачно једна особа може да буде власник једног аутомобила (и доња и горња граница кардиналности је 1). Асоцијација има само један назив ("власник"), али би се могла написати и са два назива од којих би сваки одговарао смеру у којем се асоцијација посматра: "је власник" (смер од класе Особа ка класи Аутомобил) и "има власника" (смер од класе Аутомобил ка класи Особа).



Slika 1: Primer asociјације

Имплементација ове асоцијације у Јави би изгледала овако:

```
class Osoba {  
    String ime;  
    String prezime;  
    String adresa;  
    String jmbg;  
}  
  
class Automobil {  
    String marka;  
    String model;  
    String registacija;  
    int brojMotora;  
  
    //Asocijacije mogu da se implementiraju kao atributi klase.  
    //Tip podatka atributa je druga klasa iz asocijacije.  
    Osoba vlasnik;  
}
```

Битно је приметити да се асоцијације имплементирају као атрибуты класе, али се као тип вредности наводи назив друге класе.

Позивање објекта преко асоцијације се врши на сличан начин као и позивање обичног атрибута са тим што је објекат потребно инцијализовати пре првог коришћења.

Пример 31

Написати класу *ТестАутомобил* која креира један објекат класе *Аутомобил* (из претходног примера) марке “Форд”, модел “Фокус” регистрације “БГ123-456” и број мотора “123456”. Поставити да власник овог аутомобила буде *Пера Перић* ЈМБГ: 2112980710018 који живи у Ресавској 40.

```
class TestAutomobil {

    public static void main(String[] args) {
        Automobil a = new Automobil();

        a.marka = "Ford";
        a.model = "Focus";
        a.registacija = "BG123-456";
        a.brojMotora = 123456;

        //Atribut vlasnik je objekat klase Osoba
        //pa prvo mora da se inicializuje
        a.vlasnik = new Osoba();

        //Atributima osobe koja je vlasnik se
        //pristupa preko naziva objekta klase
        //Automobil i preko naziva atributa
        a.vlasnik.ime = "Pera";
        a.vlasnik.prezime = "Peric";
        a.vlasnik.jmbg = "2112980710018";
        a.vlasnik.adresa = "Resavska 40";
    }

}
```

Атрибутима објекта класе *Особа* који је власник аутомобила се може приступити посредно - прео позивањем објекта класе *Аутомобил*, па позивањем атрибута *власник*. Између свака два елемента у оквиру позива је тачка⁴.

```
a.vlasnik.ime = "Pera";
```

Композиција - декомпозиција представља посебну врсту асоцијације у којој објекат једне класе, као саставни део, садржи један или више објеката друге класе. Постоје две врсте релација композиција-декомпозиција.

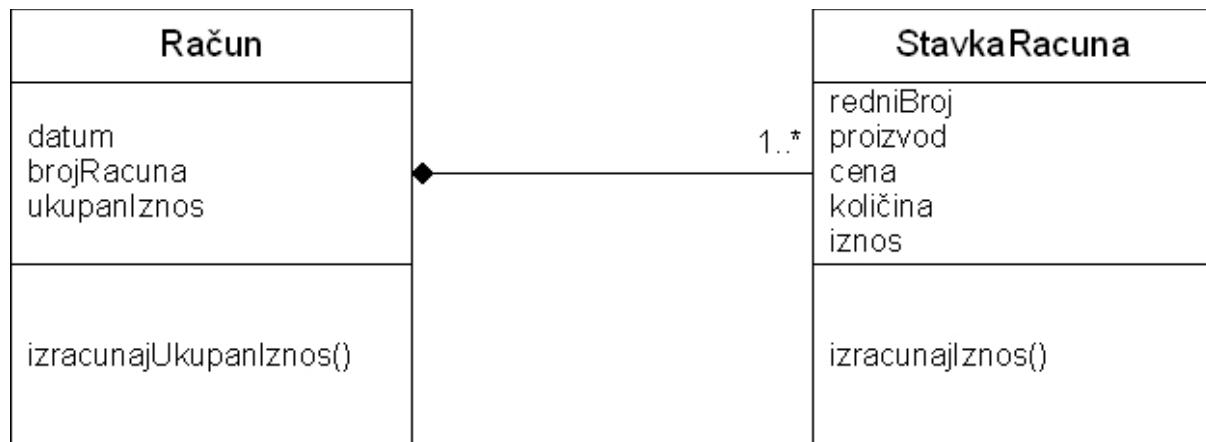
Ако објекат (или више њих) који је садржан не може да постоји независно од објекта који га садржи, у питању је **композиција**, а у супротном је у питању **агрегација**. Код композиције, када се обрише објекат који садржи друге објекте, бришу се и ти објекти. Код агрегације то није случај.

Релација композиције-декомпозиције се графички представља линијом која на једном крају има ромб. Ако је ромб испуњен у питању је композиција, а ако је празан онда је агрегација. Композиција-декомпозиција може да има назив, а кардиналност се пише само са једне стране јер је са друге стране увек 1.

⁴ У реалним ситуацијама се избегава директно приступање атрибутима класа јер се тиме нарушава принцип учаурења података (енкапсулација). О учаурењу ће бити више речи у следећим поглављима.

Пример 32

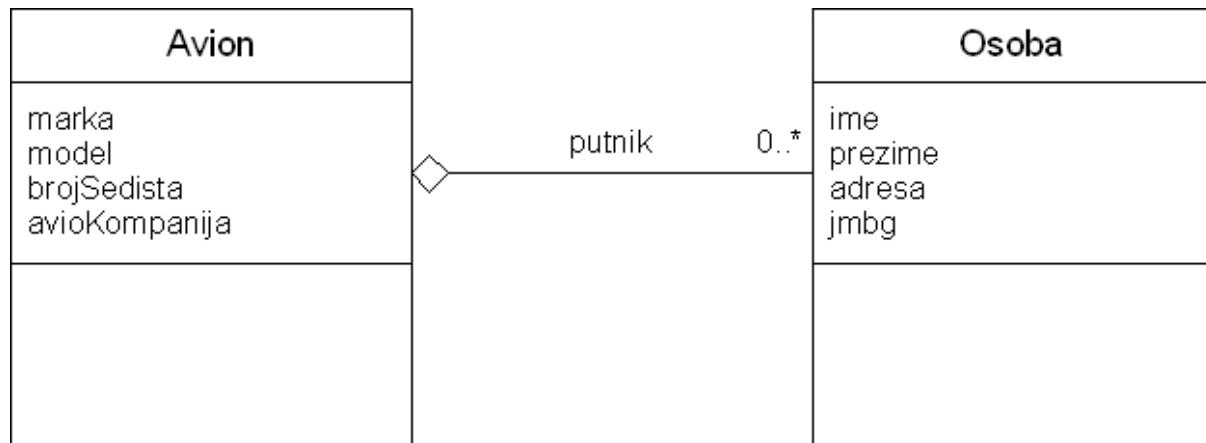
Пример који се често наводи да опише релацију композиције је однос између класа “Рачун” и “СтавкаРачуна” (Слика 11). Сваки рачун садржи једну или више ставки (доња граница кардиналности је 1, а горња граница је више), а свака ставка припада тачно једном рачуну (и доња и горња граница је увек 1 па се ни не пише). Не би имало никаквог смисла ако би постојале ставке рачуна које не припадају ниједном рачуну или ставке које припадају неколико рачуна одједном. Такође, када се обрише рачун, бришу се и све његове ставке.



Слика 11: Пример композиције

Пример 33

Релација агрегације може да се демонстрира преко односа класа Авион и Особа у којима су особе путници у авиону (Слика 12). За разлику од композиције, агрегација подразумева да агрегирани објекти могу да постоје и без објекта који их садржи. У овом случају, објекти класе Особа могу да постоје и као засебни ентитети - потпuno невезано за контекст Авиона и његових путника. На пример, објекат класе Особа може да формира асоцијацију “власник” са објектима класе Аутомобил.



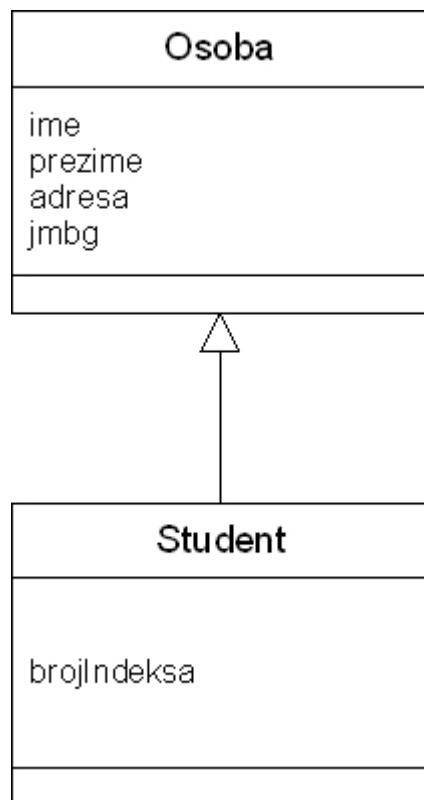
Слика 12: Пример агрегације

Композиција-декомпозиција се имплементира на исти начин као и обична асоцијација - преко атрибута класе. Једина разлика је у начину на који се тумачи асоцијација.

Генерализација - специјализација је врста релације у којој једна класа “наслеђује” другу класу, па се зато често назива и **наслеђивање**. Наслеђивање подразумева преузимање свих атрибута и јавних метода⁵ **надкласе** (класа која је наслеђена) и њихово преношење у **подкласу** (класа која наслеђује). Крајњи ефекат је тај да подкласа поседује све карактеристике и (јавно доступна) понашања надкласе, али такође може да садржи и додатне карактеристике и понашања. Наслеђивање ће детаљно бити објашњено у једном од наредних поглавља. За сада, довољно је рећи да се графичко представљање релације наслеђивања врши усмереном линијом чији врх представља празну троугласту стрелицу и да се уз ову релацију не наводи кардиналност нити назив.

Пример 34

Нека класа **Особа** има атрибуте **име**, **презиме**, **адреса** и **јмбг**. Нека је потребно направити класу **Студент** која има исте ове атрибуте, али и број индекса. У овом случају, најједноставније решење је да се ове класе направе тако да класа **Студент** наслеђује класу **Особа**. **Студент** ће да наследи атрибуте **име**, **презиме**, **адреса** и **јмбг** из класе **Особа**, па ће бити потребно само додати атрибут **број индекса** (Слика 13).



Слика 13: Пример наслеђивања

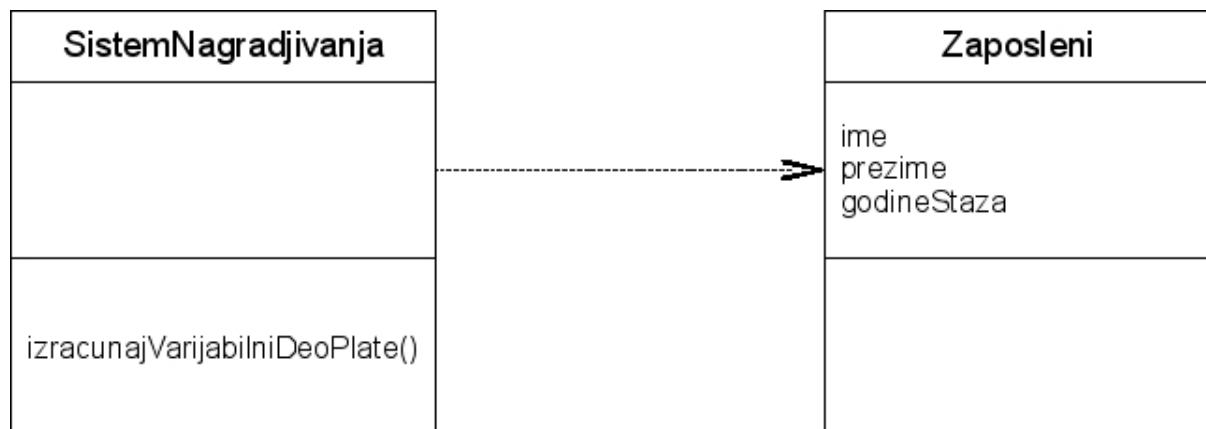
Релација коришћења се јавља када једна класа користи објекат друге класе у оквиру неке своје методе. Није битно да ли је објекат те друге класе параметар методе, повратна вредност методе или се само користи у оквиру тела методе. Релација коришћења најчешће нема експлицитно означену кардиналност нити име, а графички се представља испрекиданом линијом усмереном ка класи која се користи.

5 О јавним методама и ниовима приступа уопште, ће бити више речи у наредним поглављима.

Пример 35

Написати класу Запослени која има атрибуте: име, презиме и годинеСтаза. Направити класу СистемНаградјивања која има методу израчунајВаријабилниДеоПлате. Ова метода као параметар прима објекат класе Запослени и израчунава и враћа проценат који представља варијабилни део плате по формулама: годинеСтажа * 1.2.

На слици (Слика 14) се може видети како се релација коришћења графички приказује. Као и друге релације и ова је представљена линијом између класа, али је овај пут линија испрекидана и усмерена - класа СистемНаградјивања користи класу Запослени. Код релације коришћења се обично не наводи назив нити кардиналност.



Слика 14: Пример релације коришћења

```
class Zaposleni {
    String ime;
    String prezime;
    int godineStaza;
}

class SistemNagradjivanja {
    double izracunajVarijabilniDeoPlate(Zaposleni z) {
        double var;
        var = z.godineStaza * 1.2;
        return var;
    }
}
```

У овом примеру, класа СистемНаградјивања користи објекат класе Запослени као параметар једне своје методе. Методе такође могу и да враћају објекте као резултат.

3 Наредбе за контролу тока извршавања програма

Алгоритам се може дефинисати као процедура са јасно дефинисаним корацима који воде ка решењу неког проблема. Сваки низ Јава команди који се напише у оквиру тела неке методе заправо представља неки алгоритам (тј. његову реализацију у Јави). Јава команде из досадашњих примера се извршавају искључиво у редоследу у којем су написане - “одозго на доле”. Међутим, у неким ситуацијама је пожељно извршити неке команде само ако је неки услов испуњен. Друге ситуације захтевају да се одређена команда понови већи број пута. У сваком од ова три случаја се говори о различитим **алгоритамским структурима**:

- **Линијска (секвенцијална)** - свака команда се извршава само једнпут и то у редоследу у којем је написана, безусловно и без икаквих понављања.
- **Разграната (селекција)** - на почетку се проверава одређени услов, па ако је услов испуњен извршава се један скуп наредби, а ако није, извршава се неки други скуп наредби.
- **Циклична (итерација)** - одређена наредба (или скуп наредби) се извршава више пута заредом.

Реализација линијске алгоритамске структуре у Јави је једноставна - довољно је написати наредбе “једну испод друге” и оне ће се извршавати секвенцијално (“одозго на доле”). Реализација разгранате и цикличне алгоритамске структуре у Јави се постиже коришћењем посебних наредби. То су **наредбе за контролу тока извршавања програма** и деле се у две групе у зависности од тога коју алгоритамску структуру реализују:

- Наредбе за условно гранање (реализују разгранату алгоритамску структуру)
 - IF наредба
 - SWITCH наредба
- Наредбе за циклично понављање (реализују цикличну алгоритамску структуру)
 - FOR наредба
 - WHILE наредба
 - DO-WHILE наредба

Уобичајена пракса је да алгоритми у себи садрже комбинације више алгоритамских структура па се, према томе, и наредбе за контролу тока програма по потреби могу међусобно комбиновати.

3.1 IF наредба

Програмери се често сусрећу са ситуацијама у којима је потребно проверити неки услов, па у зависности од његове испуњености, урадити једну ствар или другу. На пример, потребно је проверити да ли је унета висина у оквиру граница 80 - 240 цм, па ако је висина ван граница исписати поруку о грешци. Ово се може постићи коришћењем IF наредбе. Декларација IF наредбе се врши на следећи начин:

```
if ( ...uslov... )      komanda_1;
```

Декларација почиње резервисаном речи “if” (мала слова). После тога, пише се логички услов који се проверава. После логичког условия се наводи **команда која се извршава ако је услов**

испуњен (команда IF). Услов који се пише између заграда у оквиру IF наредбе представља неки логички израз који се може проверити и свести на тачан (true) или нетачан (false). У најједноставнијем случају, тај израз може да представља проверу да ли је неки број већи, мањи или једнак неком другом броју. Оператори за поређење вредности су дати у следећој табели.

| Оператори за поређење вредности | | |
|---------------------------------|---|------------------------------|
| Оператор | Опис | Примери |
| > | Веће од | $a > b$ $x > 2$ |
| < | Мање од | $a < b$ $x < 2$ |
| \geq | Веће или једнако | $a \geq b$ $x \geq 135.5$ |
| \leq | Мање или једнако | $a \leq b$ $x \leq 13$ |
| $=$ | Једнако (пише се као два знака једнакости један поред другог) | $a == b$ $x == 45.22$ |
| \neq | Различито | $a != b$ $x != 1$ |

Оператори за поређење вредности се **могу користити искључиво са простим типовима података**, јер не функционишу у оним ситуацијама када је потребно упоредити два објекта. String, Calendar и сви други објекти се пореде на други начин, а не преко оператора за поређење вредности. Ако би се покушало поређење два објекта преко оператора “==”, **поредили би се садржаји показивача на те објекте (адресе објекта у меморији), а не садржај атрибута објекта.**

Потребно је напоменути и да **оператори “=” и “==” нису исти**. У првом случају, у питању је оператор додељивања вредности (левој страни се додељује вредност са десне стране), док је у другом случају у питању оператор поређења једнакости (пореди се да ли су лева и десна страна једнаке). Ако се уместо једног напише други, Јава јавља грешку. Нови програмери често греше и мешају ова два оператора, па је потребно обратити пажњу при писању логичких услова.

Пример 36

Направити класу ProveraЦелихБројева која има:

- *Методу провериЗнак која проверава да ли је број, који се прослеђује као параметар позитиван, негативан или нула и исписује поруку о томе на екрану у форми “Број __ је __”.*
- *Методу провериВеће/Мање/Једнако која прима два броја A и B као параметре и проверава да ли је $A > B$, $A < B$ или $A = B$ и исписује одговарајућу поруку о томе на екрану.*

```
class ProveraCelihBrojeva {
```

```

void proveriZnak (int a){
    if (a == 0)
        System.out.println("Broj "+a+" je jednak nuli");

    if (a > 0)
        System.out.println("Broj "+a+" je veci od nule");

    if (a < 0)
        System.out.println("Broj "+a+" je manji od nule");
}

void proveriVeceManjeJednako (int a, int b){
    if (a > b)
        System.out.println("Broj "+a+" je veci od broja "+b);

    if (a == b)
        System.out.println("Broj "+a+" je jednak broju "+b);

    if (a < b)
        System.out.println("Broj "+a+" je manji od broja "+b);
}
}

```

Метода ПровераЦелихБројева као параметар добија број, проверава да ли је једнак нули, мањи од нуле или већи од нуле и исписује поруку о томе на екрану. Провера се врши уз помоћ три независне IF наредбе. Услов који се проверава у оквиру прве наредбе је да ли је број једнак нули. Ако јесте, извршиће се команда која се налази у продужетку тј. исписаће се на екрану порука да је број једнак нули. Ако услов није испуњен. тј ако број није једнак нули, неће се извршити испис на екрану. Слично важи и за наредне две IF наредбе, само што оне проверавају да ли је број већи од нуле или мањи од нуле.

Метода провериВећеМањеЈеднако исто има три IF наредбе, али се у оквиру њих проверава да ли је параметар A већи, једнак или мањи од параметра B.

У претходном примеру је приказан и принцип да се Јава команде могу писати у више редова. Другим речима, IF команда се може написати и овако:

```

if ( ...uslov... )
    komanda_1;

```

Ово је корисно у ситуацијама када је услов или команда која га следи прилично дугачка па не може да стане у један ред на екрану, или не може да се види у целости без померања текста.

IF наредба **може да има и “else” део (али не мора)**. После резервисане речи “else” се пише команда која се извршава **само ако услов није испуњен (команда_2)**. Декларација IF наредбе која има и “else” гласи:

```

if ( ...uslov... )      komanda_1;
    else                  komanda_2;

```

Пример 37

Допунити класу ПровераЦелихБројева тако да има и:

- Методу различито која прима два броја као параметре и враћа TRUE ако су бројеви различити, а у супротном враћа FALSE.

- Методу `proveriParnost` која прима цео број као параметар и проверава да ли је паран или непаран. Метода враћа `TRUE` ако је паран, а `FALSE` ако је непаран.

Направити класу `TestProveraCelihBrojeva` која креира један објекат класе `ProveraCelihBrojeva` и користи њене методе.

Код ове две методе је следећи:

```
boolean razlicito (int a, int b){  
    if (a != b) return true;  
        else return false;  
}  
  
boolean proveriParnost (int a){  
    if ((a%2) == 0) return true;  
        else return false;  
}
```

Код класе `TestProveraCelihBrojeva` је следећи:

```
class TestProveraCelihBrojeva {  
  
    public static void main(String[] args) {  
        ProveraCelihBrojeva p = new ProveraCelihBrojeva();  
  
        p.proveriZnak(-12);  
  
        p.proveriVecemanjeJednako(123, 456);  
  
        boolean razliciti = p.razlicito(12, 13);  
        if (razliciti)  
            System.out.println("Brojevi su razliciti");  
        else  
            System.out.println("Brojevi su jednaki");  
  
        boolean paran = p.proveriParnost(33);  
        if (paran)  
            System.out.println("Broj 33 je paran");  
        else  
            System.out.println("Broj 33 je neparan");  
    }  
}
```

У методи `proveraParnosti` је приказано како се проверава да ли је број паран или непаран. Ако је остатак при дељењу бројем два једнак нули, број је паран, иначе је непаран.

Досадашњи примери обухватају оне случајеве у којима је после провере услова IF наредбе потребно извршити само једну команду. У случају када је потребно извршити више наредби после провере услова, наредбе се морају уоквирити у **блок наредби** витичастим заградама:

```
if ( ...uslov... ) {  
    komanda_1;  
    komanda_2;  
    ...  
    komanda_n;  
}
```

```
else {
    komanda_1_1;
    komanda_1_2;
    ...
    komanda_1_m;
}
```

Пример 38

Направити класу *ПровераРеалнихБројева* тако да има:

- Методу *мањиОдPi* која прима реалан број као параметар и враћа *TRUE* ако је број мањи од 3.141592, а у супротном враћа *FALSE*. У оба случаја, требало би исписати поруку на екрану о томе да ли је број мањи од пи или није.

```
class ProveraRealnihBrojeva {

    boolean manjiOdPi(double x) {

        if (x < 3.141592) {
            System.out.println("Broj "+x+" je manji od pi");
            return true;
        }

        else {
            System.out.println("Broj "+x+" je veci ili jednak pi");
            return false;
        }
    }
}
```

Метода *мањиОдPi* садржи IF наредбу која покреће две наредбе ако је услов испуњен и друге две ако није. Ове наредбе су морале да буду уоквирене у блок наредби коришћењем витичастих заграда.

Претходни пример указује и на још једну појаву. Поред тога што се *return* наредбом враћа нека вредност, врши се и моментално прекидање даљег извршавања методе. Другим речима, сваки пут када се ретурн команда изврши, метода се прекида и све команде које су написане после (“испод”) ове команде се не извршавају. Због тога, требало би обратити пажњу на то да је *return* команда последња у блоку наредби.

Услов који се проверава у оквиру IF команде може да буде и сложен тј. да се састоји из више простијих услова. Формирање сложенијих услова се врши коришћењем логичких оператора И (AND), ИЛИ (OR) и НЕ (NOT) чија спецификација је дата у следећој табели.

| Логички оператори | | |
|-------------------|----------|--|
| Оператор | Опис | Примери |
| && | И (AND) | ((a > 2) && (a < 5)) а мора да буде веће од 2 и мање од 5 да би цео услов био тачан (true) |
| | ИЛИ (OR) | ((x < 0) (x > 33.3)) х мора да буде мање од 0 или веће од 33.3 да би цео услов био тачан |
| ! | НЕ (NOT) | (!(a > b)) а не сме да буде веће од б да би цео услов био тачан |

Пример 39

Допунити класу *ПровераРеалнихБројева* тако да има и:

- Методу која проверава да ли је параметар A (реалан број) у распону од 100 до 200 укључујући и те вредности. Ако јесте, метода враћа TRUE а иначе FALSE.
- Методу која проверава да ли је параметар A (реалан број) мањи од нуле или већи од 33.3. Ако важи било који од ова два услова, метода враћа TRUE а иначе враћа FALSE. У оба случаја је потребно исписати и одговарајуће обавештење на екрану.

Код тражених метода је:

```
boolean proveraRaspona1 (double a) {
    if ((a >=100) && (a <=200)) return true;
    else return false;
}

boolean proveraRaspona2 (double a) {
    if ((a < 0) || (a > 33.3)) {
        System.out.println("Broj "+a+
                           " manji od nule ili veci od 33.3");
        return true;
    }

    else {
        System.out.println("Broj "+a+
                           " je u rasponu 0 - 33.3");
        return false;
    }
}
```

У досадашњим примерима су приказане неке методе које садрже више **редно повезаних** IF наредби. Овако написане, IF наредбе се посматрају као међусобно независне (одвојене) наредбе. Међутим, у неким ситуацијама се тражи да се после провере једног услова провере и још неки услови или само ако је први услов задовољен. Решење оваквих проблема подразумева писање тзв. **паралелно повезаних (угњеждених) наредби**, конкретно - угњеждених IF наредби:

```
if ( ...uslov 1... )    if ( ...uslov 2... )    komanda_1_1;
                           else    komanda_2_1;
```

```
else      if ( ...uslov 3... )   komanda_1_2;
            else komanda_2_2;
```

Из декларације се може видети да се уместо обичних команда које следе услов, налазе нове IF наредбе. Тако ће да се провери “услов 2” тек ако је задовољен “услов 1”. Ако и “услов 1” и “услов 2” важе, извршиће се “команда_1_1”, а ако важи “услов 1” а не важи “услов 2” извршиће се “команда_2_1”. Слично томе, ако не важи “услов 1”, провериће се “услов 3” па, ако он буде важио, извршиће се “команда_1_2”, а у супротном “команда_2_2”. IF наредбе се могу угњеждавати и стављати у блок наредби заједно са другим наредбама тако да је и следећа ситуација честа:

```
if ( ...uslov... )      {
    komanda_1;
    komanda_2;
    if ( ...uslov 2... )   komanda_1_1;
                           else komanda_2_1;
    ...
    komanda_n;
}
else {
    if ( ...uslov 3... )   komanda_1_2;
                           else komanda_2_2;
    komanda_3_1;
    komanda_3_2;
    ...
    komanda_3_m;
}
```

Пример 40

Направити класу ВисинскеИСтароснеГрупе тако да има:

- Методу **превериVisину** која, као параметар прима висину неке особе у сантиметрима (реалан број) и исписује на екрану да ли та особа припада ниским особама (мање од или једнако 158цм), средње високим особама (више од 158 цм, мање или једнако 179цм) или високим особама (више од 179 цм). Ако је унета висина ван граница (120-240цм), на екрану је потребно исписати само поруку о грешци.
- Методу **превериСтаросноДоба** која као параметар добија старост особе изражену у годинама. Метода прво проверава да ли је унета старост у границама 0 - 120 година. Ако није, исписује се порука о грешци на екрану. Ако јесте, потребно је проверити да ли је особа млада (0-30 година), средњег доба (31-55 година) или стара (56 година и више) и исписати поруку о томе на екрану.

Направити класу ТестВисинскеИСтароснеГрупе која креира један објекат класе ВисинскеИСтароснеГрупе и позива његове методе.

```
class VisinskeIStarosneGrupe {

    void proveriVisinu(double visina) {
        if ((visina < 120) || (visina > 240))
            System.out.println("Visina je van granica");
        else{
            if (visina <= 158)
                System.out.println("Osoba je niska");
            if ( (visina > 158) && (visina <= 179))
                System.out.println("Osoba je srednje visine");
```

```
        if (visina > 179)
            System.out.println("Osoba je visoka");
    }

void proveriStarosnoDoba (int starost){
    if ((starost < 0) || (starost > 120))
        System.out.println("Uneta starost je van granica");
    else{
        if (starost <= 30)
            System.out.println("Osoba je mlada");
        if ( (starost > 30) && (starost <= 55))
            System.out.println("Osoba je srednjeg doba");
        if (starost > 55)
            System.out.println("Osoba je stara");
    }
}

class TestVisinskeIStarosneGrupe {

    public static void main(String[] args) {
        VisinskeIStarosneGrupe vsg = new VisinskeIStarosneGrupe();

        vsg.proveriStarosnoDoba(5);
        vsg.proveriStarosnoDoba(35);
        vsg.proveriStarosnoDoba(56);

        vsg.proveriVisinu(145);
        vsg.proveriVisinu(185);
        vsg.proveriVisinu(175);
    }
}
```

IF наредба се често користи када је потребно проверити да ли је нека улазна вредност у оквиру задатих граница за неки атрибут или променљиву (нпр. да ли је висина у оквиру граница 120-240cm). Овај поступак провере улазних вредности се често зове **логичка контрола улазних података**.

3.2 SWITCH наредба

IF наредба омогућава условно гранање тј. извршавање једне групе команди ако је неки логички услов испуњен, а друге групе команди ако није. Ово је **“једноструко” гранање** јер су понуђене максимално две алтернативне путање (“гране”) од којих се бира и извршава само једна. Ако је потребно извршити проверу више услова, могуће је написати више IF наредби од којих свака проверава по један услов. Међутим, исти ефекат се може постићи коришћењем SWITCH наредбе.

SWITCH наредба омогућава проверавање више услова одједном и представља начин за изражавање **“вишеструког гранања”** у Јави. Вишеструког гранања је такво гранање у којем је понуђено више алтернативних путања тј. грана од којих се, у зависности од испуњености услова, може извршити једна или и више. Декларација SWITCH наредбе се врши на следећи начин:

```
switch ( ...селектор... ) {

    case vrednost_1: komanda_1;
                    break;

    case vrednost_2: komanda_2;
                    break;

    ...
    default:           komanda_d;
}
```

Декларација SWITCH наредбе почиње резервисаном речи “switch” после које следи **селектор** који се пише у загради. Селектор може бити **било која променљива која је целобројног или char типа**. Променљиве које су било ког другог типа (String, boolean...) не могу бити селектори. Селектор може да буде и **неки израз чији резултат је целобројног или char типа** нпр. “A%2”. SWITCH наредба има и тело у којем су дефинисане све могуће гране и команде које би требало извршити у оквиру гране. Грана се дефинише резервисаном речи “case” иза које следи нека вредност. При покретању SWITCH наредбе, **пореди се тренутна вредност селектора са вредностима које су написане иза речи “case”**. Када се нађе грана која има вредност која је једнака вредности селектора, извршавају се команде написане у продужетку (нпр. команда_1). **Ако се не нађе ниједна одговарајућа грана, извршава се она грана која је означена са “default”(команда_d)**. SWITCH наредба **не мора да има “default” грану**.

Овако како је написана, ова SWITCH наредба представља вишеструког гранање при којем се бира и извршава само једна грана од више понуђених. Управо то је смисао **“break” наредбе која је написана као последња команда сваке гране**. Када се изврше све команде неке гране, наилази се на “break” команду чиме се прекида SWITCH наредба. Да је “break” наредба изостављена, **извршила би се одабрана грана, али и све остале које су написане “испод” ње (укључујући и “default” грану)**. Тако написана SWITCH наредба би представљала вишеструког гранање при којем се бира и извршава више грана.

Такође, потребно је приметити да команде које су написане у оквиру једне гране нису уоквирене витичастим заградама па је и следећи код потпуно синтаксно исправан:

```
switch ( ...selektor... ) {  
  
    case vrednost_1: komanda_1_1;  
                      komanda_1_2;  
                      komanda_1_3;  
                      break;  
  
    case vrednost_2: komanda_2_1;  
                      komanda_2_2;  
                      komanda_2_3;  
                      break;  
  
    ...  
  
    default:          komanda_d_1;  
                      komanda_d_2;  
                      komanda_d_3;  
  
}
```

Пример 41

Направити класу *ДешифровањеДНК* која има:

- Статичку методу *десифрујДНК* која као параметар добија неки знак који представља почетно слово неке од нуклеотида од којих се састоје карике ДНК ланца. Ако знак има вредност 'A', 'Ц', 'Г' или 'Т', потребно је на екрану исписати назив одговарајуће нуклеотиде која почиње тим словом: Аденин, Цитозин, Гуанин и Тимин. Ако унети знак нема ниједну од тих вредности исписати на екрану поруку о грешци.

Направити класу *ТестДешифровањеДНК* која позива методу класе *ДешифровањеДНК*.

```
class DesifrovanjeDNK {  
  
    static void desifrujDNK(char karika) {  
        switch (karika) {  
            case 'A':  
                System.out.println("Adenin");  
                break;  
            case 'C':  
                System.out.println("Citozin");  
                break;  
            case 'G':  
                System.out.println("Guanin");  
                break;  
            case 'T':  
                System.out.println("Timin");  
                break;  
            default:  
                System.out.println("Greska!");  
        }  
    }  
  
    class TestDesifrovanjeDNK {
```

```
public static void main(String[] args) {
    DesifrovanjeDNK.desifrujDNK('C');
}

}
```

SWITCH наредба, нажалост, не омогућава формирање логичких услова као код IF наредбе, па је зато њена употребљивост ограничена. Ако је потребно проверити неке сложеније услове, или ако се не може наћи одговарајући селектор, једино решење које остаје је да се напише више IF наредби.

3.3 FOR наредба

Контролисање тока извршавања програма подразумева и решавање оних ситуација у којима је потребно једну или више команди поновити већи број пута. У овим случајевима, користе се **наредбе за циклично понављање**. Ове наредбе се називају и **циклуси (петље)**. Једна врста петље је и FOR петља. Декларација FOR петље се врши на следећи начин:

```
for (команда_з1; услов_з; команда_з2)      команда_п;
```

Декларација почиње резервисаном речи “for”, после које следи заграда са две команде и условом. Прва команда у загради (команда_з1) се **извршава само једанпут и то на почетку**. Услов (услов_з) представља **логички израз који се проверава пре извршавања сваке итерације (круга)**. Ако је услов задовољен, извршиће се још једна итерација. Ако није, петља се прекида. Последња команда у загради (команда_з2) се **извршава на крају сваке итерације**. После заграде се налази команда коју је потребно извршити више пута (команда_п). Када се петља покрене, редослед извршавања команди и провере услова је следећи:

ПОЧЕТАК ПЕТЉЕ

- Извршава се команда_з1

1. ИТЕРАЦИЈА

- Проверава се услов_з (услов важи)
- извршава се команда_п
- извршава се команда_з2

2. ИТЕРАЦИЈА

- Проверава се услов_з (услов важи)
- извршава се команда_п
- извршава се команда_з2

3. ИТЕРАЦИЈА

- Проверава се услов_з (услов важи)
- извршава се команда_п
- извршава се команда_з2

...

ПОСЛЕДЊА ИТЕРАЦИЈА

- Проверава се услов_з (услов НЕ важи - петља се прекида)

КРАЈ ПЕТЉЕ

Ако је потребно циклично понављање више наредби, потребно их је уоквирити у блок наредби уз помоћ витичастих заграда:

```
for (komanda_z1; uslov_z; komanda_z2) {
    komanda_p_1;
    komanda_p_2;
    ...
    komanda_p_n;
}
```

Пример 42

Направити класу Исписивац која има:

- Статичку методу испишиПорукуПетПута која на екрану пет пута исписује поруку “Добар дан”.

```
class Ispisivac {

    static void ispisiPorukuPetPuta() {
        for (int i=1; i<=5; i++)
            System.out.println("Dobar dan");
    }
}
```

Метода испишиПорукуПетПута је могла да се реализује и тако што би се написала “System.out.println” команда пет пута заредом, али то не би уопште било практично. Прво, број “System.out.println” команди би морао да се промени сваки пут када би се променио број тражених исписа. Ако би се број потребних исписа повећао на сто или хиљаду пута, овај приступ не би био практично изводљив. Друго, шта ако је потребно поновити више наредби? То би значило да се цео блок наредби напише више пута у оквиру тела методе што сам код чини гломазним, непрегледним и јако тешким за одржавање. Због тога је коришћена FOR petlja. У овом случају, направљена је променљива “i” и додата јој је почетна вредност 1. Пре сваке итерације се проверава да ли је “i” мање или једнако 5. Ако јесте, извршава се итерација - исписује се “Добар дан” на екрану и “i” се увећава за 1 (i++). Ово се ради све док “i” не добије вредност 6 у ком случају се прекида петља. Следи кратак опис тога шта се дешава када се покрене петља:

ПОЧЕТАК ПЕТЉЕ

- int i = 1; (декларише се променљива “i” и добија вредност 1)

1. ИТЕРАЦИЈА ($i = 1$)

- $i \leq 5 ? (1 \leq 5 - \text{услов је испуњен})$
- $\text{System.out.println("Dobar Dan");}$
- $i++; (i = 2 - \text{променљива је увећана за 1})$

2. ИТЕРАЦИЈА ($i = 2$)

- $i \leq 5 ? (2 \leq 5 - \text{услов је испуњен})$
- $\text{System.out.println("Dobar Dan");}$
- $i++; (i = 3 - \text{променљива је увећана за 1})$

3. ИТЕРАЦИЈА ($i = 3$)

- $i \leq 5 ? (3 \leq 5 - \text{услов је испуњен})$
- $\text{System.out.println("Dobar Dan");}$

- $i++;$ ($i = 4$ - променљива је увећана за 1)

4. ИТЕРАЦИЈА ($i = 4$)

- $i \leq 5 ? (4 \leq 5$ - услов је испуњен)
- `System.out.println("Dobar Dan");`
- $i++;$ ($i = 5$ - променљива је увећана за 1)

5. ИТЕРАЦИЈА ($i = 5$)

- $i \leq 5 ? (5 \leq 5$ - услов је испуњен)
- `System.out.println("Dobar Dan");`
- $i++;$ ($i = 6$ - променљива је увећана за 1)

6. ИТЕРАЦИЈА ($i = 6$)

- $i \leq 5 ? (6 \leq 5$ - услов НИЈЕ испуњен, петља се прекида)

КРАЈ ПЕТЉЕ

Променљива “и” из претходног примера се користи да би се обезбедило да се изврши тачно 5 итерација FOR петље. Ова променљива се назива **бројач циклуса**. Уобичајена је пракса да се бројач циклуса и декларише и да му се додели почетна вредност у оквиру прве наредбе у оквиру заграде FOR наредбе. Услов за излазак из петље је да бројач достигне неку вредност, а трећа команда у оквиру заграде мења тренутну вредност бројача (у претходном примеру, увећава га за један). Са обзиром на то да се ова трећа команда извршава на крају текуће итерације, бројач добија нову вредност која се преноси у следећу итерацију. Када се бројач циклуса декларише у оквиру прве команде у загради FOR наредбе, он је **“видљив” само у оквиру FOR петље**⁶ што значи да се не може се користити и позивати ван ње.

Пример 43

Преправити класу *Исписивач* тако да има и:

- Статичку методу `ispisiPorukuNPuta` која као параметар прима једну поруку (Стринг) и један позитиван цео број N . Ова метода на екрану исписује N пута унету поруку.
- Статичку методу `ispisiOd0Do30` која на екрану исписује бројеве од 0 до 30.
- Статичку методу `ispisiOd30Do0` која на екрану исписује бројеве од 0 до 30 али у обрнутом редоследу (30, 29, 28, 27, ..., 2, 1, 0).

Направити класу *ТестИсписивач* која позива све методе класе *Исписивач*.

После допуна, код класе *Исписивач* изгледа овако:

```
class Ispisivac {
    static void ispisiPorukuPetPuta() {
        for (int i=1; i<=5; i++)
            System.out.println("Dobar dan");
    }

    static void ispisiPorukuNPuta(String poruka, int n) {
        for (int i=1; i<=n; i++)
            System.out.println(poruka);
    }

    static void ispisiOd0Do30() {
```

6 Концепт видљивости променљивих је обрађен у другом поглављу.

```
        for (int i=0; i<=30; i++)
            System.out.println(i);
    }

    static void ispisiOd30Do0(){
        for (int i=30; i>=0; i--)
            System.out.println(i);
    }

}

class TestIspisivac {

    public static void main(String[] args) {
        //Sve metode klase Ispisivac su staticke
        //pa se ne mora kreirati objekat klase
        //Ispisivac da bi se pozvale
        Ispisivac.ispisiPorukuPetPut();
        Ispisivac.ispisiPorukuNPut("Zdravo", 2);
        Ispisivac.ispisiOd0Do30();
        Ispisivac.ispisiOd30Do0();
    }
}
```

У методи “испишиПорукуНПута” се параметар *N* користи у оквиру услова за излазак из петље. Ако се, на пример, унесе број 5 као вредност *N*, петља ће извршити 5 итерација (порука ће се исписати 5 пута на екрану) а ако се унесе број 10 извршиће се 10 итерација.

Метода “испишиОд0До30” исписује на екрану све целе бројеве из обог распона. Најједноставније решење је да се подеси тако да бројач петље узима вредности од 0 до 30 и да се у свакој итерацији испише тренутна вредност бројача за ту итерацију. Бројач као почетну вредност добија нулу. У првој итерацији се на екрану исписује тренутна вредност бројача (нула) а бројач се увећава за 1. У другој итерацији се на екрану опет исписује тренутна вредност бројача (сада износи један) а бројач се увећава за 1. У трећој итерацији се поново исписује његова тренутна вредност (два), бројач се увећава за један итд. У последњој итерацији, бројач има вредност 30, која се исписује на екрану (јер је $30 \leq 30$ па услов важи). Када се вредност бројача увећа за 1 и постане 31, услов престаје да важи ($31 > 30$) и циклус се прекида.

У методи “испишиОд30До0” је приказано да се бројач петље може и умањивати у сваком кораку ако је то потребно. То онда, наравно, захтева да и се почетна вредност бројача као и услов за излазак из петље прилагоде ситуацији.

FOR наредба се може комбиновати и са свим другим наредбама за контролу тока програма. Тако, може да се деси да у неком блоку наредби FOR петље налази и нека IF наредба и обратно. Ово угњежђавање наредби може да буде и тако да FOR наредба садржи и другу FOR наредбу или неки други циклус.

Пример 44

Направити класу СложениИсписивач која има:

- Статичку методу испишиПарнеБројевеОд1До25 која на екрану исписује све бројеве у

- распону од 1 до 25 који су парни.
- Статичку методу испшиМатрицу која на екрану исписује матрицу димензија 6x4 чији су сви елементи једнаки 1 и то тако да се сви елементи из истог реда матрице напишу у једном реду на екрану:

```

1     1     1     1
1     1     1     1
1     1     1     1
1     1     1     1
1     1     1     1
1     1     1     1

```

Написати класу ТестСложениИсписивач која позива методе класе СложениИсписивач.

```

class SlozeniIspisivac {

    static void ispisiParneBrojeveOd1Do25() {
        //FOR нaredba u sebi sadrzi IF нaredbu
        //која у свакој iterацији проверава да
        //ли је trenutna vrednost brojaca paran
        //broj i исписује га на екрану ако јесте
        for(int i = 1; i<=25; i++)
            if (i%2 == 0) System.out.println(i);
    }

    static void ispisiMatricu(){
        //FOR нaredba u sebi sadrzi drugu FOR
        //нaredbu. Unutrasnji FOR циклус исписује
        //све елементе једног реда на екрану док
        //се спољашњим FOR циклусом обезбедује
        //да се то уради по једном за сваки red.
        for (int i = 1; i<=6; i++){
            for(int j = 1; j<=4; j++) System.out.print(1+" ");
            //Ova naredba ne pripada unutrasnjem
            //FOR ciklusu
            System.out.println();
        }
    }
}

class TestSlozeniIspisivac {

    public static void main(String[] args) {
        SlozeniIspisivac.ispisiParneBrojeveOd1Do25();
        SlozeniIspisivac.ispisiMatricu();
    }
}

```

Метода “испишиПарнеБројевеОд1До25” садржи FOR петљу која има угњеждену IF наредбу. У свакој итерацији се проверава да ли је тренутна вредност бројача паран број па се, ако јесте, исписује на екрану. Са обзиром на то да бројач узима вредности од 1 до 25, ефекат је тај да ће се исписати сви парни бројеви у том распону - што се и тражило.

Метода “испишиМатрицу” садржи FOR петљу која има угњеждену другу FOR петљу. Прво, потребно је приметити да свака петља има различит бројач. Спољашња петља има бројач “i” док унутрашња има бројач “j”. Унутрашња петља исписује елементе из једног реда

матрице. Са обзиром на то да је потребно исписати ове елементе у једном реду на екрану, користи се "print" а не "println" наредба. "System.out.println" наредба која следи унутрашњу петљу не припада овој петљи а користи се да би се после исписивања елемената из једног реда матрице прешло у следећи ред на екрану. Спољашња петља служи томе да се обезбеди исписивање елемената за свих шест редова.

Уобичајен начин за излазак из FOR петље је да услов који се проверава на почетку сваке итерације престане да важи. Међутим, могуће је прекинути FOR петљу и на други начин, коришћењем **"break"** наредбе. Значи, ако се у некој итерацији FOR петље изврши команда "break", **петља се истог тренутка прекида**. Трећи начин за прекидање FOR петље произилази из карактеристика "return" наредбе. Ако метода у којој се користи FOR петља враћа неку вредност, **петља се може прекинути позивањем "return" наредбе**. Споредни ефекат је тај да ће се на овај начин прекинути и извршавање целе методе. И **WHILE и DO-WHILE петље се могу прекинути коришћењем ове две команде**.

У Јави постоји и наредба чијим извршавањем се обезбеђује да се прескочи тренутна итерација и да се настави са следећом итерацијом петље. То је **"continue"** наредба. Када се изврши, све наредбе које би требало да се изврше у тој итерацији, а које су написане "испод" ове наредбе се прескачу, и прелази се у следећу итерацију. Једина команда која се регуларно извршава је трећа команда у загради FOR петље којом се повећава тј. смањује вредност бројача.

Пример 45

Допунити класу СложениИсписивач тако да има и:

- Статичку методу испишиМинДељивСа12и15и9 која на екрану исписује најмањи број у распону од 10 до 1000 који је дељив са 12, 15 и 9 истовремено.
- Статичку методу вратиМинДељивСа12и15и9 која враћа најмањи број у распону од 10 до 1000 који је дељив са 12, 15 и 9 истовремено.

Код тражене две методе је следећи:

```
static void ispisiMinDeljivSa12i15i9() {  
  
    for(int i = 10; i<=1000;i++)  
        if ((i%12 == 0)&&(i%15 == 0)&&(i%9==0)) {  
            System.out.println(i);  
            break;  
        }  
  
    }  
  
static int vratiMinDeljivSa12i15i9(){  
  
    for(int i = 10; i<=1000;i++)  
        if ((i%12 == 0)&&(i%15 == 0)&&(i%9==0))  
            return i;  
  
    return 0;  
}
```

У првој методи, пролази се кроз бројеве од 10 до 1000 коришћењем FOR петље. Ова петља има угњеждену IF наредбу тако да се у сваком кораку проверава да ли је тренутна вредност бројача дељива са 12, 15 и 9 истовремено. Ако није, прелази се у следећу итерацију. Међутим, када се нађе број који јесте дељив са 12, 15 и 9, исписује се на екрану и извршава се "break" наредба која прекида FOR петљу. Да није написана ова "break" наредба, FOR

петља би се наставила и исписала би све бројеве у распону од 10 до 1000 који су дељиви са 12, 15 и 9 истовремено а не само најмањи.

Друга метода је скоро иста као и прва са тим што се, чим се нађе број који је дељив са 12, 15 и 9, тај број враћа коришћењем “return” наредбе. Ова наредба прекида извршавање FOR петље али и целе методе.

3.4 WHILE наредба

Већ је речено да се наредбе за циклично понављање користе у оним ситуацијама када је потребно извршити неку команду (или више њих) више пута. FOR петља је најкориснија у оним ситуацијама када се на почетку циклуса, пре његовог извршења, зна колико ће бити итерација. На пример, када је потребно исписати бројеве од 1 до 100 на екрану, зна се да се то може урадити у 100 итерација тако што ће се у свакој итерацији исписати по један број. Чак иако број итерација зависи од неког параметра или променљиве, он ће бити познат у тренутку када петља почне. Међутим, постоје и оне ситуације у којима није унапред познато колико ће се итерација извршити. У таквим ситуацијама нема смисла користити бројач петље јер услов за излазак из петље не зависи од вредности бројача већ од неког другог услова. WHILE наредба је настала управо да би могла бити примењена у оваквим случајевима. Декларација WHILE петље се врши на следећи начин:

```
while (...uslov...)      komanda_p;
```

Декларација почиње резервисаном речи “while” иза које се у загради пише неки логички услов. После логичког услова пише се команда (или више команди) коју је потребно поновити више пута (команда_p). WHILE петља **понавља извршавање команде (команда_p) све док услов из заграде важи**. За разлику од FOR петље, WHILE петља **нема бројач**. Излазак из петље не зависи од вредности бројача, већ се дешава кад услов престане да важи. Овај **услов је обичан логички израз**, и формира се на потпуно исти начин као било који услов из IF наредбе (логички оператори, оператори поређења и њихове комбинације). Када се петља покрене, редослед извршавања команди и провере услова је следећи:

ПОЧЕТАК ПЕТЉЕ

1. ИТЕРАЦИЈА

- Проверава се услов (услов важи)
- извршава се команда_p

2. ИТЕРАЦИЈА

- Проверава се услов (услов важи)
- извршава се команда_p

3. ИТЕРАЦИЈА

- Проверава се услов (услов важи)
- извршава се команда_p

...

ПОСЛЕДЊА ИТЕРАЦИЈА

- Проверава се услов (услов НЕ важи - петља се прекида)

КРАЈ ПЕТЉЕ

Може се десити да услов не важи на почетку петље. У том случају се **неће извршити ниједна итерација**. Међутим, исто тако је могуће да се, услед грешке, напише услов који увек важи без обзира на све. Ефекат ће бити тај да ће се WHILE петља непрестано понављати и постаће тзв. **“бесконачна петља”**.

Као и код FOR петље, ако је потребно циклично понављање више наредби, морају се уоквирити у блок наредби уз помоћ витичастих заграда:

```
while (...uslov...) {  
    komanda_p_1;  
    komanda_p_2;  
    ...  
    komanda_p_n;  
}
```

Наредбе **“break”** и **“continue”** функционишу и код WHILE циклуса. Овај циклус се може прекинути и коришћењем “return” наредбе али се у том случају прекида и метода у којој се циклус налази.

Пример 46

Направити класу Увећање која има:

- Статичку методу која као параметар добија неки позитиван цео број A и множи га самим собом све док не постане већи од броја 1000. Ова метода у ствари израчунава најмање A^n које је веће од 1000. Овако умножен број је потребно исписати на екрану.

Направити класу TestУвећање која проверава који је најмањи степен броја 2 који је већи од 1000.

```
class Uvecanje {  
  
    void veciOd1000(int a) {  
        int rez = 1;  
  
        while (rez < 1000) rez = rez * a;  
  
        System.out.println(rez);  
    }  
}  
  
class TestUvecanje {  
  
    public static void main(String[] args) {  
        //Ispisace se 1024 (dva na deseti)  
        Uvecanje.veciOd1000(2);  
    }  
}
```

WHILE петља из методе “већиОд1000” је постављена тако да се множење броја самим собом врши све док је резултат мањи од 1000. У тренутку када резултат постане већи од 1000, петља се прекида и резултат се исписује на екрану.

У истом примеру се види и то да је потребна опрезност при писању услова за излазак из петље. Ако се методи као аргумент проследи број један, WHILE петља ће да постане бесконачна петља. То је због тога што ће резултат множења јединицом увек бити исти и неће се увећавати ($1^n = 1$) - биће увек мањи од 1000 без обзира на то колико се итерација изврши.

3.5 DO-WHILE наредба

Ако услов WHILE петље није испуњен на почетку, пре прве итерације, ниједна итерација се неће извршити и петља ће да се прекине. Разлог је тај што се услов проверава на почетку сваке итерације, пре него што се изврши иједна команда. Међутим, у неким ситуацијама је потребно обезбедити да се увек изврши макар једна итерација петље. Начин на који се то постиже је коришћењем DO-WHILE петље:

```
do
    komanda_p;
while (...uslov...);
```

DO-WHILE петља има скоро исту синтаксу као WHILE петља. Обе петље се извршавају све док услов (логички израз у загради) важи. Разлика између ове две петље је у томе што се у DO-WHILE петљи **услов проверава на крају итерације**, а не на почетку. Ефекат је тај да ће се **увек извршити макар једна итерација** чак иако услов не важи. Када се петља покрене, редослед извршавања команди и провере услова је:

ПОЧЕТАК ПЕТЉЕ

1. ИТЕРАЦИЈА

- извршава се команда_п
- Проверава се услов (услов важи)

2. ИТЕРАЦИЈА

- извршава се команда_п
- Проверава се услов (услов важи)

3. ИТЕРАЦИЈА

- извршава се команда_п
- Проверава се услов (услов важи)

...

ПОСЛЕДЊА ИТЕРАЦИЈА

- извршава се команда_п
- Проверава се услов (услов НЕ важи - петља се прекида)

КРАЈ ПЕТЉЕ

Као и код осталих петљи, ако је потребно циклично понављање више наредби, морају се уоквирити у блок наредби уз помоћ витичастих заграда:

```
do {
    komanda_p_1;
    komanda_p_2;
```

```
    ...
    komanda_p_n;
} while (...uslov...);
```

Пример 47

Направити класу *Сигураниспис* која има:

- Статичку методу која као параметар добија неку поруку и цео број A. Метода би требало да A пута испише поруку на екрану. Потребно је поруку исписати макар једанпут, па чак и у ситуацијама када је унети број A нула или мањи од нуле.

Написати класу *ТестСигураниспис* која позива методу класе *Сигураниспис* да испише поруку "Добар дан" 5 пута и "Лаку ноћ" 5 пута.

```
class SiguranIspis {

    static void ispisi(String poruka, int a) {
        int brojac = 0;

        //Izvrsice se uvek makar jedna iteracija
        //jer se uslov proverava posle ispisa na
        //ekranu.
        do{
            System.out.println(poruka);
            brojac++;
        }
        while (brojac<a);
    }

}

class TestSiguranIspis {

    public static void main(String[] args) {
        //Poruka ce se ispisati 5 puta
        SiguranIspis.ispisi("Dobar dan", 5);

        //Poruka ce se ispisati jedanput
        SiguranIspis.ispisi("Laku noc", -5);
    }

}
```

4 Низови

Променљива представља неки идентификатор (име, слово, израз исл.) који је повезан са неком вредношћу, при чему се та вредност може мењати. У Јави, оне се декларишу навођењем типа променљиве (int, boolean, double...) па онда и назива променљиве. За сваку вредност коју је потребно негде ускладиштити, доволно је дефинисати по једну променљиву. Али, шта се дешава када је потребно ускладиштити више вредности - на пример 50 целих бројева. Једно решење је да се декларише 50 целобројних променљивих и да се свакој од њих додели по једна вредност. Ово решење, међутим, није практично (шта ако се ради о 1000 целих бројева) и не може се применити ако се унапред не зна потребан број променљивих.

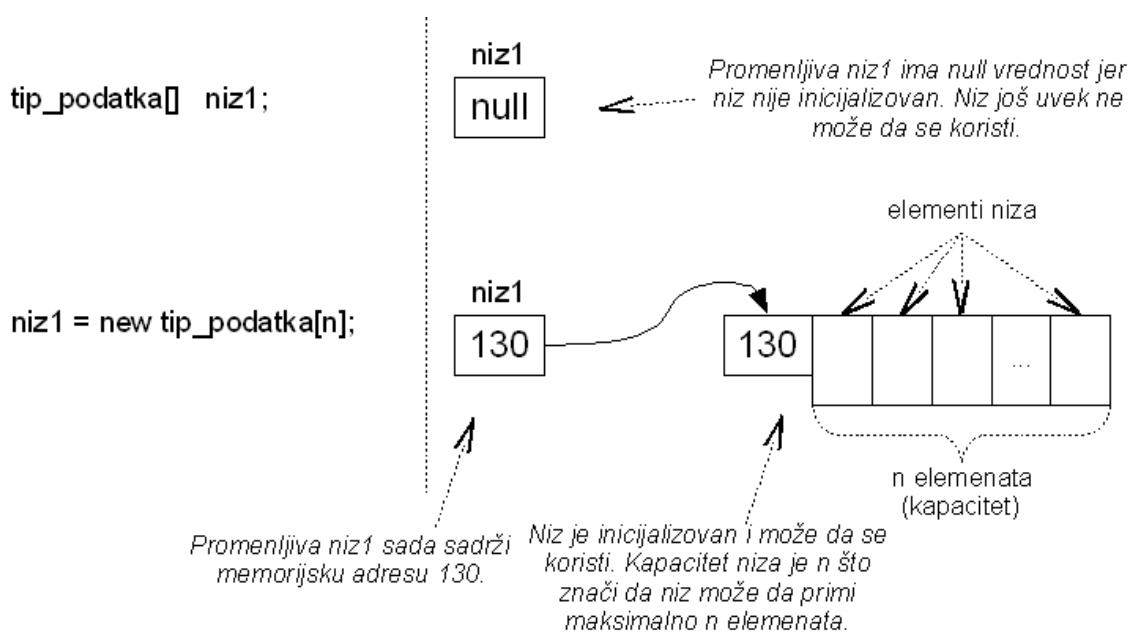
Право решење за овакве ситуације представљају **низови**. Низови су променљиве које **могу да ускладиште више вредности одједном**. Декларација низа се врши на следећи начин:

```
tip_podatka[] nazivPromenljive;
```

Декларација је скоро идентична као за обичну променљиву, једини додатак представљају угласте заграде које се наводе после типа податка. Као што је речено, низови могу да ускладиште више вредности одједном. Ове вредности се називају **елементи низа**. Сви елементи једног низа су **увек истог типа** (нпр. низ целих бројева, низ реалних бројева...). Да би низ могао да се користи он **прво мора да се иницијализује**. Иницијализација се врши на следећи начин:

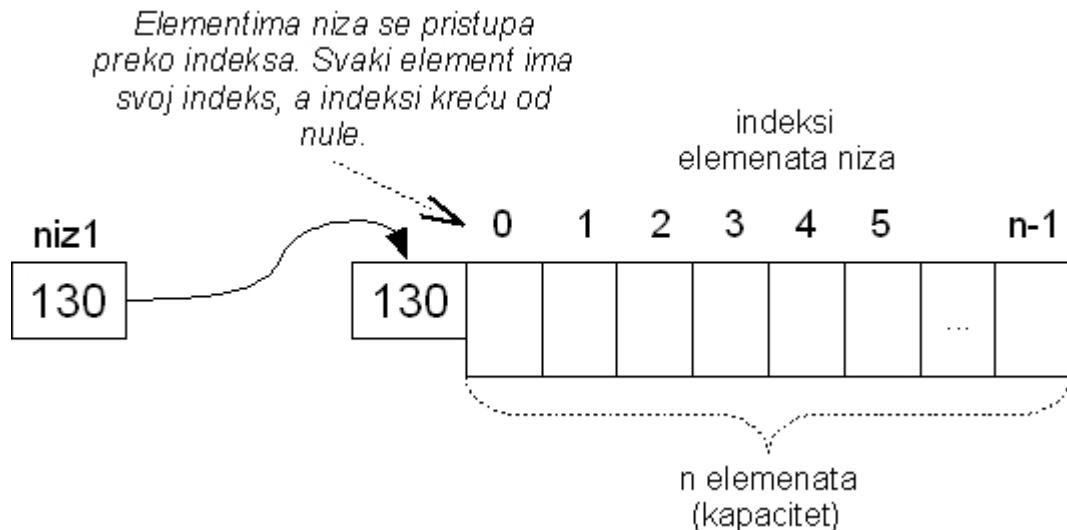
```
nazivPromenljive = new tip_podatka[ceo_broj];
```

Шта заправо иницијализација ради? Променљива која представља низ у ствари није сам низ већ **показивач на низ**. Иницијализацијом се алоцира меморија за низ и тек тада се низ може користити. Број који се налази у загради мора да буде неки ненегативан цео број. Он представља **капацитет низа** тј. колико ће елемената низ моћи максимално да садржи. На пример, низ целих бројева капацитета 100 може да прими 100 целих бројева. Једном када се одреди, **капацитет низа остаје фиксиран** и не може се смањити нити повећати (Слика 14). Низ се може поново иницијализовати (ако је потребно да се повећа или смањи капацитет), али се тада бришу елементи низа.



Слика 14: Иницијализација низа

Променљива која представља низ има само један назив, а односи се на више елемената, па се поставља питање како се приступа појединачним елементима низа. Сваки елемент низа има свој **индекс**. Индекс елемента је цео број који представља редни број елемента у низу. **Индекси елемената почињу од нуле** а не од један. Тако је, на пример, индекс првог елемента низа капацитета 10 увек нула а последњег 9 (Слика 15).



Слика 15: Индекси елемената низа

Позивање појединачних елемената низа уз помоћ индекса се врши на следећи начин:

```
nazivPromenljive[indeks]
```

Ако је потребно **добити вредност капацитета низа** (нпр. да би се видело да ли је низ довољно дугачак) то се ради на следећи начин:

```
nazivPromenljive.length
```

Пример 48

Нека је потребно декларисати низ реалних бројева. Одговарајућа команда и њен ефекат су дати на слици (Слика 16).

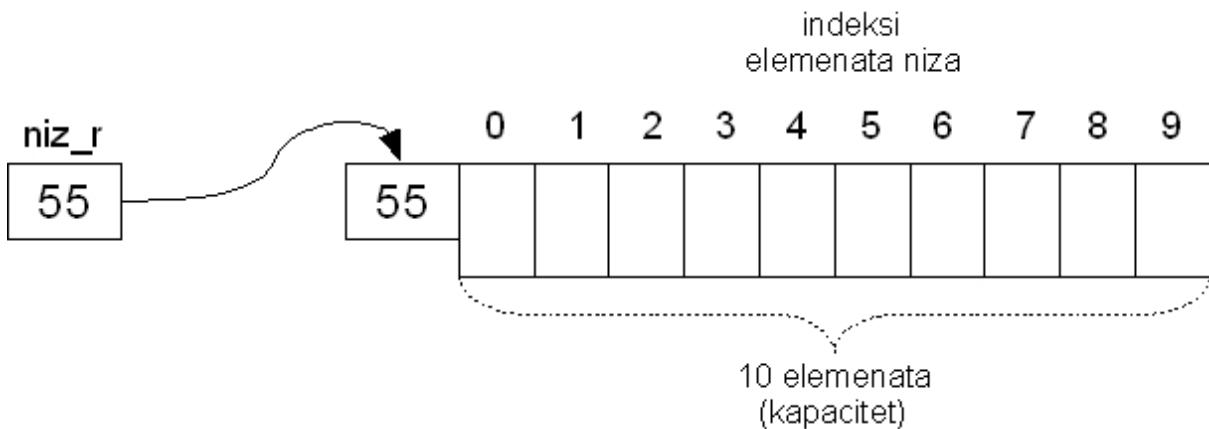
```
double [] niz_r;
```

| |
|-------|
| niz_r |
| null |

Слика 16: Декларација низа реалних бројева

Затим, нека је потребно иницијализовати низ тако да му максимални капацитет буде 10 елемената (Слика 17).

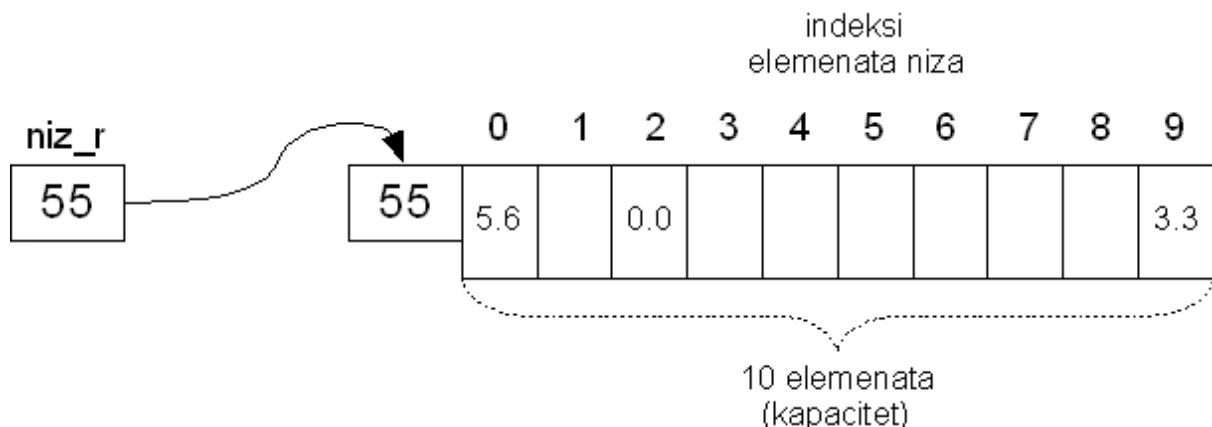
```
niz_r = new double [10];
```



Слика 17: Иницијализација низа реалних бројева на капацитет 10

На крају, нека је потребно првом елементу низа доделити вредност 5.6, трећем 0, а последњем 3.3 (Слика 18).

`niz_r[0] = 5.6;`
`niz_r[2] = 0;`
`niz_r[9] = 3.3;`



Слика 18: Додељивање вредности неким елементима низа

У пракси, низови се често користе у комбинацији са FOR петљом. Тада се бројач FOR петље користи да симулира индекс елемената низа. Поставља се да бројач петље има почетну вредност нула (јер индекси низа крећу од нуле), а петља се извршава све док бројач не достigne максималну вредност индекса.

Пример 49

Направити класу `МесечниПрофити` која има:

- Атрибут профити који представља низ од 12 реалних бројева. Сваки елемент низа представља профит за одређени месец (јануар, фебруар, ..., децембар).
- Методу која као параметре прима реалан број који представља профит и цео број који представља редни број месеца на који се тај профит односи (1 - јануар, 2 - фебруар, ..., 12 - децембар). Метода уноси одговарајућу вредност профита за тај месец у низ.
- Методу која на екрану исписује профит за сваки месец.

Направити класу `ТестиМесечниПрофити` која креира један објекат класе `МесечниПрофити`, уноси профит за фебруар који износи 122.33 и исписује све месечне профите на екрану.

```
class MesecniProfiti {  
  
    double[] profiti = new double[12];  
  
    void unesiProfit(double profit, int mesec){  
        //Ovde стоји [mesec-1] а не mesec jer  
        //индекси низа крећу од нуле па је индекс  
        //за januar 0 иако је то први месец (1) а  
        //индекс за decembar 11 иако је то дванаести  
        //месец.  
        profiti[mesec-1] = profit;  
    }  
}
```

```

void ispisi(){
    for(int i=0; i<profiti.length;i++)
        System.out.println(profiti[i]);
}

class TestMesecniProfiti {

    public static void main(String[] args) {
        MesecniProfiti mp = new MesecniProfiti();

        mp.unesiProfit(122.33, 2);

        mp.ispisi();
    }

}

```

Атрибут профити представља низ реалних бројева који се иницијализује на одговарајући капацитет - 12.

Метода “унесиПрофит” као параметар прима износ профита и редни број месеца на који се профит односи.

Метода “испиши” користи FOR петљу да испише све елементе низа. Бројач петље “и” као почетну вредност добија 0 а не 1 јер индекси елемената низа почињу од нуле. Бројач ће се у свакој итерацији увећавати за један, а петља ће се извршавати све док је бројач мањи од капацитета. То свакако има смисла јер ће у последњој итерацији вредност бројача бити “капацитет-1” (“profiti.length-1”) а то је управо индекс последњег елемената низа. У свакој итерацији петље се исписује по један елемент низа и то онај елемент низа који има индекс који је једнак вредности бројача петље за ту итерацију (“profiti[i]”). Тако, у првој итерацији ће “и” имати вредност 0 и исписаће се “profiti[0]” а то је управо први елемент низа. У другој итерацији “и” ће имати вредност 1 па ће се исписати “profiti[1]” (други елемент низа) и тако даље. У последњој итерацији “и” ће имати вредност “капацитет-1” па ће се исписати последњи елемент низа.

Низ који је дат у претходном примеру се увек користи до максималног капацитета. Шта то значи? Низ има дванаест елемената и увек се користе сви елементи (сваки представља износ профита за неки месец у току године). Међутим, могуће су и ситуације у којима се не користи пун капацитет низа. Пример овакве ситуације је низ оцена на испитном року из неког предмета - зна се да је испит пријавио одређени број студената али реално увек на испит изађе мање студената. У тим случајевима, погодно је увести и још једну помоћну променљиву која ће да има функцију бројача елемената у низу. Овај бројач ће да садржи податак о томе колико стварно има елемената у низу, а не који је максимални капацитет низа. Наравно, да би бројач био ажуран, потребно је увећати га за један сваки пут када се дода неки елемент у низ и умањити га за један сваки пут када се избаци неки елемент из низа.

Пример 50

Направити класу ОценеНаИспитномРоку која има:

- Атрибут оцене који представља оцене студената на испитном року. Зна се да испит може максимално да положе 100 студената.
- Атрибут бројач који представља тренутни број елемената у низу. Почетна вредност је 0.

- *Методу која као параметар добија оцену на испиту и уноси је у низ и то на прво слободно место.*
- *Методу која на екрану исписује све оцене на испитном року.*

Направити класу `TestOceneNaIspitnomRoku` која креира један објекат класе `OceneNaIspitnomRoku`, уноси у њега оцене 5, 10, 10, 7 и 8 и исписује све оцене на екрану.

```
class OceneNaIspitnomRoku {  
  
    int[] ocene = new int[100];  
    int brojac = 0;  
  
    void unesiOcenu(int ocena) {  
        //Promenljiva brojac predstavlja  
        //trenutni broj elemenata u nizu  
        //ali i predstavlja indeks prvog  
        //slobodnog elementa niza.  
        ocene[brojac] = ocena;  
  
        //Novi element je dodat u niz  
        //pa je potrebno uvecati brojac  
        //za jedan.  
        brojac++;  
    }  
  
    void ispisi(){  
        //Brojac FOR petlje ide od nule  
        //do trenutnog broja elemenata niza  
        //(brojac) a ne do kapaciteta (length)  
        //jer niz nije pun.  
        for(int i=0;i<brojac;i++)  
            System.out.println(ocene[i]);  
    }  
}  
  
class TestOceneNaIspitnomRoku {  
  
    public static void main(String[] args) {  
        OceneNaIspitnomRoku oir = new OceneNaIspitnomRoku();  
  
        oir.unesiOcenu(5);  
        oir.unesiOcenu(10);  
        oir.unesiOcenu(10);  
        oir.unesiOcenu(7);  
        oir.unesiOcenu(8);  
        oir.ispisi();  
    }  
}
```

4.1 Вишедимензионални низови

Вишедимензионални подаци (нпр. матрице које су дводимензионални низови) могу да се представе у Јави коришћењем **вишедимензионалних низова**. Вишедимензионални низ је веома сличан једнодимензионалном низу у сваком погледу, а његова декларација се врши на

следећи начин (ставља се онолико угластих заграда колико подаци имају димензија):

```
tip_podatka[][] nazivPromenljive;
```

Ово је декларација дводимензионалног низа. Од свих вишедимензионалних низова се, у принципу, најчешће користе дводимензионални низови и то у оним случајевима када је потребно представити неку матрицу или табелу са подацима.

И овде је потребна иницијализација низа, са тим што се при иницијализацији **наводи капацитет низа по свакој димензији**:

```
nazivPromenljive = new tip_podatka[ceo_broj_1][ceo_broj_2];
```

Број у првој загради (“цео_број_1”) представља број редова матрице, а број у другој загради (“цео_број_2”) број колона. Приступ елементима дводимензионалног низа се такође врши преко индекса, али свака димензија има свој индекс па се, у случају дводимензионалног низа, **обавезно наводе оба индекса** (први представља ред матрице а други колону):

```
nazivPromenljive[indeks_1][indeks_2]
```

У раду са дводимензионалним низовима се најчешће користе две угњеждене FOR петље при чему бројач сваке петље глуми индекс једне димензије низа.

Пример 51

Направити класу *Матрица* која има:

- Атрибут *матрица* који представља дводимензионални низ целих бројева.
- Конструктор који као параметар прима број редова и број колона који матрица треба да има и иницијализује атрибут матрица на те капацитете.
- Методу која претвара матрицу у нула матрицу - матрицу чији су сви елементи једнаки нули.
- Методу која матрицу претвара у јединичну матрицу тј. подешава вредности елемената матрице тако да елементи који се налазе на главној дијагонали имају вредност 1 а остали 0. Метода пре проверава да ли је матрица квадратна (да ли има исти број редова и колона), па ако није исписује поруку о грешци на екрану.
- Методу која на екрану исписује вредности елемената матрице и то тако да се у једном реду на екрану испишу сви елементи који припадају истом реду матрице.

Направити класу *TestMatrica* која креира један објекат класе *Матрица* димензија 5x5 и позива његове методе.

```
class Matrica {
    int[][] matrica;

    Matrica(int brojRedova, int brojKolona) {
        matrica = new int[brojRedova][brojKolona];
    }

    void nulaMatrica() {
        // Broj redova se dobija pozivanjem komande
        // matrica.length dok se broj kolona dobija
        // kao duzina prvog reda tj. matrica[0].length
        for (int i = 0; i < matrica.length; i++)
    }
}
```

```
        for (int j = 0; j < matrica[0].length; j++)
            matrica[i][j] = 0;
    }

void jedinicnaMatrica() {
    if (matrica.length == matrica[0].length) {
        for (int i = 0; i < matrica.length; i++)
            for (int j = 0; j < matrica[0].length; j++)
                if (i == j)
                    matrica[i][j] = 1;
                else
                    matrica[i][j] = 0;
    }
    else
        System.out.println("Greska!");
}

void ispisi() {
    for (int i = 0; i < matrica.length; i++) {
        for (int j = 0; j < matrica[0].length; j++)
            System.out.print(matrica[i][j] + " ");
        System.out.println();
    }
}

class TestMatrica {

    public static void main(String[] args) {
        Matrica m = new Matrica(5,5);

        m.nulaMatrica();
        m.ispisi();

        m.jedinicnaMatrica();
        m.ispisi();
    }
}
```

5 Класа String

У досадашњим примерима су се низови знакова представљали коришћењем примитивног типа “char” и ознаке за низове (“char[]”). Ово, међутим, није најбољи начин за рад са низовима знакова. Често је потребно над одређеном речи или изразом урадити неке сложене операције - издвојити само један део речи или израза, проверити да ли се у изразу налази нека реч, заменити неко слово неким другим, спојити неке речи у реченице итд. Ако би се користио “char” низ, све ове операције би требало посебно написати у виду метода. Чак би се и поређење једнакости два низа слова по слово морало имплементирати на исти начин. У Јави постоји предефинисана класа за рад са низовима знакова која већ има имплементиране многе од ових функционалности. То је **класа String**.

Декларација String променљиве се врши на исти начин као и за било коју другу променљиву (String је класа па њен назив почиње великим словом):

```
String nazivPromenljive;
```

Са обзиром на то да је у питању класа, **свака променљива овог типа је објекат па се мора иницијализовати пре него што се може користити**. Иницијализација се може урадити на неколико начина. Прво, могуће је иницијализовати објекат класе String коришћењем конструктора.

```
String nazivPromenljive = new String("neki niz znakova");
```

Конструктор ове класе као параметар прима неку String вредност. Потребно је приметити да се **String вредности увек пишу под двоструким знацима навода** (нпр. “Реченица број 1”). Насупрот томе, “char” вредности се пишу под једноструким знацима навода тј. апострофима (нпр. 'A').

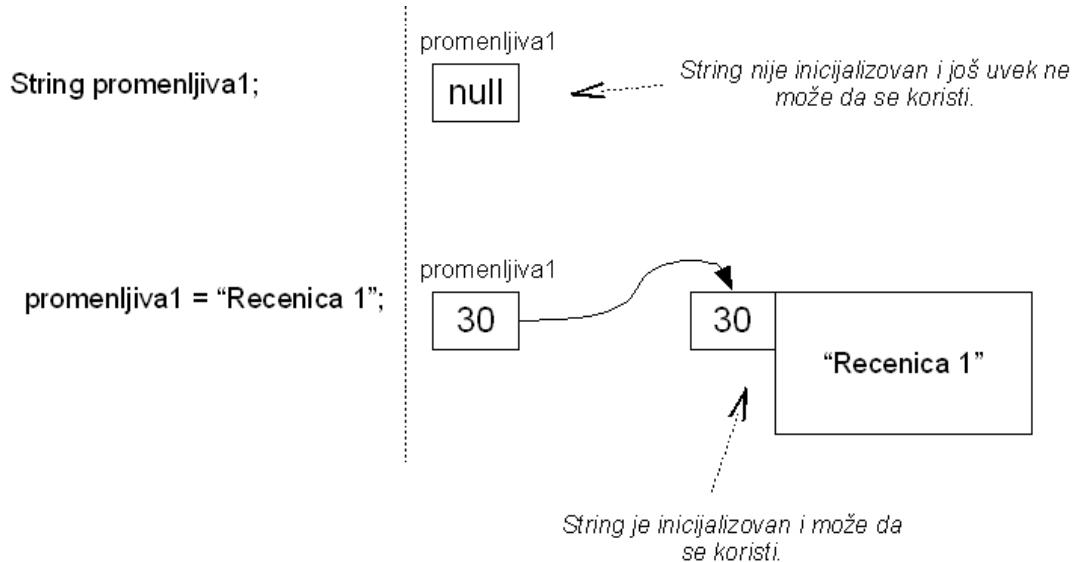
String променљиве се могу иницијализовати и на краћи начин. Ефекат претходне и наредне наредбе за иницијализацију је исти.

```
String nazivPromenljive = "neki niz znakova";
```

Овакав вид иницијализације је **могуће остварити само када су у питању String објекти и не важи за друге класе**. Могућ је и један облик иницијализације String променљиве који је користан у оним ситуацијама када је низ променљивих типа “char” потребно претворити у String објекат исте садржине. У овом случају, конструктор класе String као параметар добија низ променљивих типа “char”.

```
char[] niz;
...
String s = new String(niz);
```

Ефекат иницијализације String објекта је исти као и за објекте других класа. Пре иницијализације, променљива има “null” вредност. При иницијализацији се у меморији рачунара заузима простор одговарајуће величине за смештање конкретне String вредности (Слика 19).

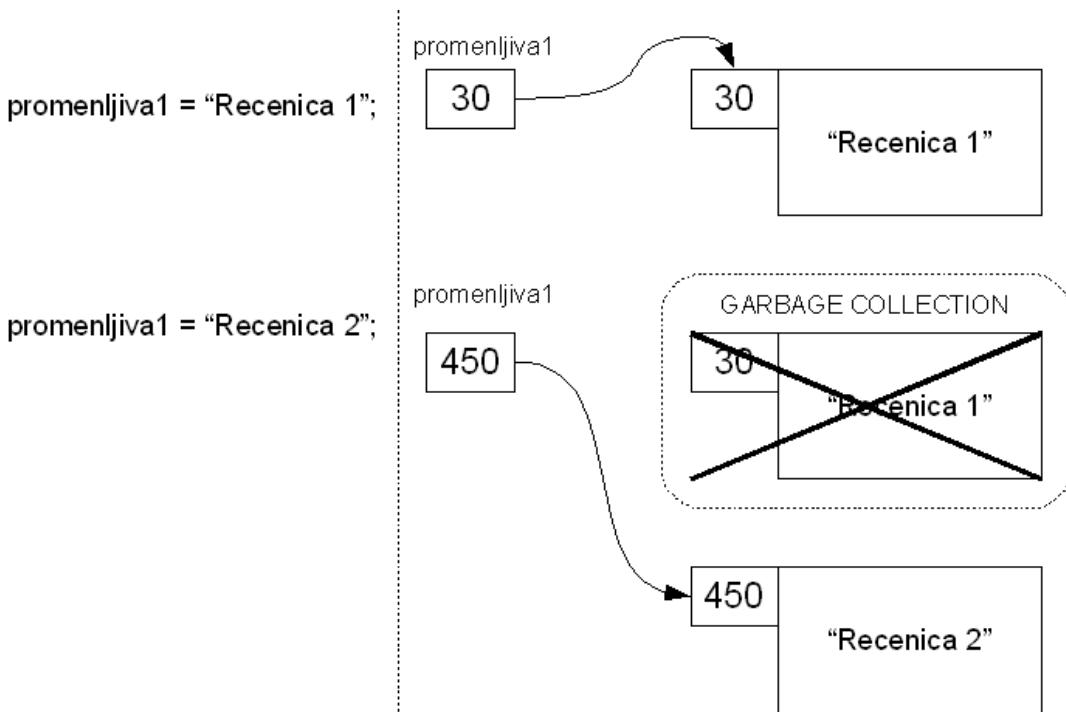


Слика 19: Иницијализација String променљиве

Поред "null" вредности, `String` променљива може да садржи и празан String. То је `String` вредност која не садржи ниједан знак (чак ни празна места) али је променљива ипак иницијализована.

```
String nazivPromenljive = "";
```

Потребно је скренути пажњу и на то шта се дешава када се `String` променљивој која већ има неку вредност додели нова вредност. Иако се чини логичним да ће се искористити меморијски простор који је већ заузет, то се не дешава. Уместо тога, иницијализује се нови String објекат и он прима нову вредност, а објекат који садржи стару вредност се ослобађа помоћу "garbage collection" механизма (Слика 20).



Слика 20: Додељивање нове вредности String променљивој

String вредности су, у ствари, низови знакова па се често дешава да је потребно спојити више вредности у један String. У питању је **надовезивање ("concatenation") String вредности** и врши се коришћењем оператора за надовезивање (који се пише исто као и знак за сабирање - `+`). Овај оператор се може користити искључиво са String вредностима.

```
nazivPromenljive = "neki niz znakova" + "drugi niz znakova";
```

Оператор за надовезивање String вредности је заправо већ коришћен у оквиру `println` команде да би се формирала нека сложенија излазна порука. Ту се може приметити још једна карактеристика овог оператора а то је да ће **све вредности које нису String типа бити претворене у одговарајући низ знакова и надовезане на постојећу String вредност**. Једини предуслов је да је **макар једна вредност у целом изразу String типа**.

Пример 52

Ево неколико примера наредби за надовезивање String вредности.

```
String s1 = "Danas";
String s2 = " je lep dan.";

//String s3 ce da dobije vrednost
///"Danas je lep dan."
String s3 = s1 + s2;

//Vrednost promenljive temperatura ce da
//se automatski pretvori u String vrednost
///"15" pa ce s4 da dobije vrednost
///"Napolju je 15 stepeni".
int temperatura = 15;
```

```
String s4 = "Napolju je "+temperatura+ " stepeni.";  
  
//Ovo ce da bude greska jer nijedan element  
//izraza nije String vrednost.  
//String s5 = 12 + 15;  
  
//Umesto toga, trebalo bi izraz napisati na jedan  
//od sledecih nacina da bi promenljiva s5  
//imala vrednost "1215"  
  
//Prvi element je String pa je sve u redu.  
String s5 = "12" + 15;  
  
//Drugi element je String pa je sve u redu.  
s5 = 12 + "15";  
  
//Podmetnut je prazan String koji nece  
//da utice na krajnju vrednost s5 ali ce  
//izraz biti ispravan.  
s5 = "" + 12 + 15;
```

Поређење једнакости за просте типове података (int, boolean, char, double) је релативно једноставно и врши се помоћу оператора за поређење једнакости (“==”). Са обзиром на то да је String класа, **String вредности се не могу поредити коришћењем оператора за поређење једнакости**. Ако би се то урадило, поредиле би се само адресе објекта у меморији а не и сам садржај. Уместо тога, потребно је користити **“equals” методу** која пореди две String вредности слово по слово и враћа “true” ако су једнаке а “false” ако нису. При томе, обраћа се пажња на велика и мала слова. Ова метода се користи на следећи начин:

```
promenljival.equals(promenljiva2)
```

Оператор за поређење једнакости се може користити једино када се жели проверити да ли String променљива има “null” вредност тј. да ли је иницијализована или не.

```
promenljival == null;
```

Пример 53

Овај пример приказује шта се дешава ако се две String вредности пореде на правilan и неправilan начин.

```
String s1 = "Recenica 1";  
String s2 = new String("Recenica 1");  
  
if (s1 == s2)  
    System.out.println("Jednaki su");  
else  
    System.out.println("Nisu jednaki");  
  
if (s1.equals(s2))  
    System.out.println("Jednaki su");  
else  
    System.out.println("Nisu jednaki");
```

У првој IF команди се вредности пореде на неправилан начин па ће да се испише да нису једнаке иако је садржај обе променљиве исти. У другој IF команди се позива equals метода која врши поређење слово по слово па ће се исписати да су једнаке вредности у питању.

Већ је речено да String класа омогућава рад са низовима знакова на једноставнији начин, пружајући додатне функционалности. Међутим, некада је потребно појединачно проћи кроз све знакове у String променљивој као да је у питању обичан низ знакова. Сваки знак String вредности има свој индекс а индекси крећу од нуле и ту се види сличност са обичним низовима. Међутим, знаковима се не приступа коришћењем угластих заграда већ позивањем **методе “charAt”**. Дужина String вредности се добија позивањем **методе “length”**.

Пример 54

Направити класу ПребројавањеЗнакова која има:

- Статичку методу преброј која као параметар прима String и враћа колико пута се у њему појављује слово 'a'.

```
class PrebrojavanjeZnakova {

    static int prebroj(String s) {
        int brojac = 0;

        for (int i=0; i<s.length(); i++)
            if(s.charAt(i) == 'a')
                brojac++;
        return brojac;
    }

}
```

Из примера се може видети да се кроз String пролази уз помоћ FOR петље на сличан начин као кроз обичан низ. Бројач петље се креће од нула (индекс првог знака) до “s.length()-1” (индекс последњег знака). Појединачном знаку се приступа позивањем “charAt” методе којој се прослеђује индекс жељеног знака.

Стринг класа има и низ других метода за рад са низовима знакова. Њихов назив и опис су дати у следећој табели.

| Назив методе | Опис |
|-------------------------------|---|
| charAt(int indeks) | Враћа знак из String променљиве који је на одређеној позицији |
| compareTo(String s) | Пореди две String вредности лексикографски (абецедно) и враћа позитиван број ако је прва вредност “после” друге вредности, негативан број ако је “пре” а нулу ако су једнаки. |
| compareToIgnoreCase(String s) | Ради исто што и претходна метода, али не обраћа пажњу на то да ли су у питању велика или мала слова. |
| endsWith(String s) | Проверава да ли се први String завршава низом знакова који се налази у Stringu с и враћа “true” ако је то тачно, а “false” ако није. |
| equals(Object o) | Проверава да ли је први String једнак унетом String-у и |

| | |
|---|---|
| | враћа “true” ако јесте а “false” ако није. Као параметар ова метода би требало да добије String без обзира на то што пише да је параметар типа Object. |
| equalsIgnoreCase(String s) | Ради исто што и претходна метода само се не обраћа пажња на велика и мала слова (на пример, “ДАН” и “дан” ће да буду протумачени као једнаке вредности). |
| indexOf(char c) | Пronалази и враћа индекс првог појављивања знака који је унет као параметар у String вредности. Ако се тај знак не може наћи, враћа -1. |
| indexOf(String s) | Ради исто што и претходна метода само тражи појављивање низа знакова. |
| indexOf(char c, int indeks) | Пronалази и враћа индекс првог појављивања знака који је унет као параметар у String вредности. Претраживање почиње од позиције у String-у која је унета у виду индекса. |
| indexOf(String s, int indeks) | Ради исто што и претходна метода само тражи појављивање низа знакова. |
| lastIndexOf(char c) | Пronалази и враћа индекс последњег појављивања знака који је унет као параметар у String вредности. Ако се тај знак не може наћи, враћа -1. |
| lastIndexOf(String s) | Ради исто што и претходна метода само тражи појављивање низа знакова. |
| lastIndexOf(char c, int indeks) | Пronалази и враћа индекс последњег појављивања знака који је унет као параметар у String вредности. Претраживање се врши уназад од последње позиције до позиције која је унета у виду индекса. |
| lastIndexOf(String s, int indeks) | Ради исто што и претходна метода само тражи појављивање низа знакова. |
| length() | Враћа дужину String вредности. |
| replace(char c1, char c2) | Пronалази сва појављивања знакова који су једнаки вредности променљиве c1 и мења их знаком из пром. c2. |
| split(String s) | Дели почетни String на више String вредности (враћа низ String вредности). Дељење се врши на оним местима где се нађе на израз који је дат као аргумент. Ако се нпр. као аргумент унесе празно место (“”), поделиће почетну вредност на појединачне речи. |
| startsWith(String s) | Проверава да ли први String почиње низом знакова који се налази у Stringu с и враћа “true” ако је то тачно, а “false” ако није. |
| substring(int pocetniIndeks) | Враћа део почетне String вредности почев од знака са индексом “почетниИндекс” па до краја. |
| substring(int pocetniIndeks, int krajnjiIndeks) | Враћа део почетне String вредности почев од знака са индексом “почетниИндекс” па до знака са индексом “крајниИндекс” или не укључујући тај знак. |
| toLowerCase() | Претвара сва велика слова String вредности у мала и враћа |

I ДЕО – ОСНОВЕ ПРОГРАМСКОГ ЈЕЗИКА ЈАВА

| | |
|---------------|--|
| | тако изменен String. |
| toUpperCase() | Претвара сва мала слова String вредности у велика и враћа тако изменен String. |
| trim() | “Одсеца” знакове који представљају празна места са почетка и краја String вредности и враћа тако изменен String. |

6 Наслеђивање, апстрактне класе, интерфејси и класа Object

У овом поглављу се детаљно обрађује појам наслеђивања. После уводних објашњења и примера, приказани су концепти апстрактних класа и интерфејса као механизама за одвајање спецификације програма од његове имплементације. Класа Object, надкласа свих класа у Јави, је објашњена на крају поглавља.

6.1 Наслеђивање

Наслеђивање (релација “IS-A”, генерализација-специјализација) представља један од основних концепата објектно-оријентисаног програмирања. Наслеђивање подразумева преузимање свих атрибута и метода **надкласе** (класа која је наслеђена) и њихово преношење у **подкласу** (класа која наслеђује). Крајњи ефекат је тај да подкласа поседује све карактеристике и понашања надкласе, али такође може да садржи и још неке карактеристике и понашање. Један од основних мотива наслеђивања је **избегавање редунданса у коду** (понављања кода). Код који је написан у надкласи не мора да се поново пише у подкласи већ се аутоматски преноси из надкласе. Ово води ка **поновном употребљавању већ написаног кода (“code reuse”)**.

Графичко представљање релације наслеђивања се врши усмереном линијом од подкласе ка надкласи. Врх ове стрелице представља празан троугао и налази се код надкласе, док се код ове врсте релације не наводи кардиналност јер је увек један.

Програмски језик Јава дозвољава само **једноструко наслеђивање** тј. подкласа може да има само једну надкласу. У неким другим објектно-оријентисаним програмским језицима (нпр. C++) је омогућено вишеструког наслеђивања, али се ту може јавити конфлікт ако две или више надкласа садрже истоимену методу или атрибут. Тада није јасно коју методу тј. атрибут ће наследити подкласа.

Наслеђивање се у Јави означава резервисаном речи **“extends”**. Ова реч се наводи при дефиницији класе одмах после назива класе, а у њеном продуктку следи назив класе која се наслеђује.

```
class NazivNadklase {  
    ...  
}  
  
class NazivPodklase extends NazivNadklase {  
    ...  
}
```

Пример 5

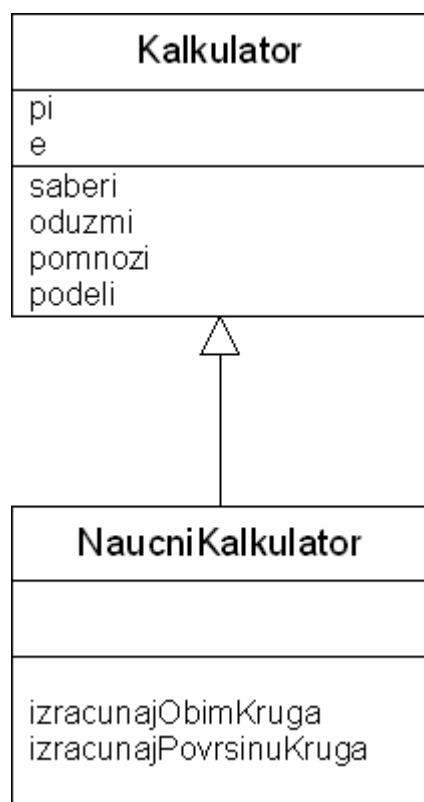
Направити класу Калкулатор. Ова класа би требало да има:

- Атрибут *pi* који има почетну вредност 3.14.
- Атрибут *e* који има почетну вредност 2.71.
- Методу *saberi* која сабира два реална броја и враћа резултат.
- Методу *oduzmi* која одузима два реална броја и враћа резултат
- Методу *pomnozi* која множи два реална броја и враћа резултат.
- Методу *podeli* која дели два реална броја и враћа резултат.

Направити класу НаучниКалкулатор која наслеђује класу Калкулатор и има:

- Методу израчунайОбимКруга која као параметар прима полу пречник круга и враћа обим круга.
- Методу израчунайПовршинуКруга која као параметар прима полу пречник круга и враћа површину круга.

Направити класу ТестКалкулатор која креира један објекат класе НаучниКалкулатор и позива све његове методе.



Слика 21: Пример наслеђивања

На слици (Слика 21) се може видети како се графички представља наслеђивање. У овом случају, класа НаучниКалкулатор (подкласа) наслеђује класу Калкулатор (надкласа). Класа Калкулатор има два атрибута и четири методе, док класа НаучниКалкулатор има две своје методе али и све атрибуте и методе из класе Калкулатор.

```

class Kalkulator {
    double pi = 3.14;
    double e = 2.71;
}
  
```

```
double saberi(double a, double b) {
    return a+b;
}

double oduzmi(double a, double b) {
    return a-b;
}

double pomnozi(double a, double b) {
    return a*b;
}

double podeli(double a, double b) {
    return a/b;
}

class NaucniKalkulator extends Kalkulator {

    //Atribut pi moze da se pozove jer je dobijen
    //nasledjivanjem od klase Kalkulator.
    double izracunajObimKruga(double poluprecnik) {
        return poluprecnik*2*pi;
    }

    double izracunajPovrsinuKruga(double poluprecnik) {
        return poluprecnik*poluprecnik*pi;
    }

}

class TestNaucniKalkulator {

    public static void main(String[] args) {
        NaucniKalkulator nk = new NaucniKalkulator();

        //Pozivanje metoda klase NaucniKalkulator koje su
        //napisane u toj klasi
        System.out.println("Obim: "+nk.izracunajObimKruga(5));
        System.out.println("Povrsina: "+
                           nk.izracunajPovrsinuKruga(5));

        //Pozivanje metoda klase NaucniKalkulator koje su
        //nasledjene od klase Kalkulator
        System.out.println("2+3="+nk.saberi(2,3));
        System.out.println("2-3="+nk.oduzmi(2,3));
        System.out.println("2*3="+nk.pomnozi(2,3));
        System.out.println("2/3="+nk.podeli(2,3));
    }

}
```

Класа *НаучниКалкулатор* наслеђује класу *Калкулатор*. Може се видети да се резервисана реч “extends” налази одмах после назива подкласе и да се после ње налази назив надкласе. У оквиру метода “израчунајОбимКруга” и “израчунајПовршинуКруга” се налазе позиви ка

атрибуту “пи”, што је дозвољено јер је тај атрибут наслеђен из класе Калкулатор. У оквиру “main” методе класе ТестКалкулатор се види да је класа НаучниКалкулатор наследила и методе “сабери”, “одузми”, “помножи” и “подели”, а да нису морале да буду експлицитно написане у оквиру класе. На тај начин не долази до понављања кода ових метода у обе класе.

Методе из надкласе се преносе у подкласу у оригиналном облику. Међутим, честа је ситуација да се у подкласи захтева да наслеђена метода има мало другачије понашање. У том случају, могуће је написати методу са истим заглављем (назив, повратна вредност и параметри) или која ће да има другачије понашање. Овај поступак се назива **редефинисање методе (“overriding”)**. Када се у оквиру подкласе буде позвала жељена метода, извршиће се редефинисана верзија методе, а не оригинална. Са обзиром на то да су, у суштини, конструктори специфична врста метода, могуће је редефинисати и конструктор у оквиру подкласе.

Ако редефинисана метода или конструктор само “проширују” функционалност оригиналне методе надкласе, постоји могућност да се у оквиру редефинисане методе позове оригинална метода, а онда да се само допише део кода који се односу на нову функционалност. Позивање методе или конструктора надкласе се врши коришћењем **резервисане речи “super”** на следећи начин:

```
//Pozivanje konstruktora nadklase
//koji nema parametre
super();

//Pozivanje konstruktora nadklase
//koji ima parametre
super(...argumenti...);

//Pozivanje metode nadklase
super.nazivMetode(...argumenti...);
```

Позивање метода или конструктора надкласе је опционо и не мора се радити сваки пут већ само када је неопходно. При томе, конструктор надкласе се може позвати само из конструктора подкласе а **наредба за позив мора да буде на првој линији кода конструктора подкласе**. Ако надкласа има експлицитно дефинисан параметризовани конструктор а нема експлицитно дефинисан подразумевани конструктор, онда **подкласа мора да има експлицитно дефинисан конструктор који позива неки конструктор надкласе**. Битно је знати и то да се, при иницијализацији објекта, увек прво позива конструктор надкласе (и њених надкласа), па тек онда конструктор подкласе.

Пример 56

Направити класу Особа која има:

- Атрибут име. Почетна вредност овог атрибута је “Н”.
- Атрибут презиме. Почетна вредност овог атрибута је “Н”.
- Атрибут јмбг (String) који представља матични број.
- Конструктор који као параметар прима име, презиме и јмбг и поставља вредности одговарајућих атрибута само аку су вредности сва три параметра различита од NULL. У супротном се исписује порука о грешци на екрану.
- Методу испиши која на екрану у три реда исписује податке о имену, презимену и јмбг особе.

Направити класу Ђак која наслеђује класу Особа и има:

- Атрибут просечнаОцена који представља просечну оцену у школи (нпр. 4,66).
- Конструктор који као параметар прима име, презиме, јмбг и просечну оцену ђака и

поставља вредности одговарајућих атрибута само ако су вредности прва три параметра различита од null и ако је просечна оцена у распону од 1 до 5. У супротном се исписује порука о грешци на екрану.

- Редефинисану методу испиши која на екрану у четири реда исписује податке о имену, презимену, јмбг и просечној оцени Ђака.

Направити класу Пензионер која наслеђује класу Особа из претходног примера и има:

- Атрибут пензија који представља износ који пензионер прима као пензију (нпр. 14200,5 динара).
- Конструктор који као параметар прима име, презиме, јмбг и износ пензије и поставља вредности одговарајућих атрибута само ако су вредности прва три параметра различита од нулл и ако је износ пензије већи од нуле. У супротном се исписује порука о грешци на екрану.
- Редефинисану методу испиши која на екрану у три реда исписује податке о имену, презимену и износу пензије. Није потребно исписати јмбг пензионера.

Направити класу ТестОсоба која креира по један објекат класе Особа, Ђак и Пензионер. Доделити особи име "Пера Перић" и матични број "1212007710567". Доделити Ђаку име "Мика Лазић", матични број "1010000715076" и просечну оцену 5.0. Доделити пензионеру име "Жика Жикић", матични број "0909944710078" и пензију 23400.0 динара. Исписати податке о свој тројици на екрану.

```
class Osoba {  
  
    String ime = "N";  
    String prezime = "N";  
    String jmbg;  
  
    Osoba(String ime, String prezime, String jmbg){  
        if(ime!=null && prezime!=null && jmbg!=null){  
            this.ime = ime;  
            this.prezime = prezime;  
            this.jmbg = jmbg;  
        }  
        else System.out.println("Greska");  
    }  
  
    void ispisi(){  
        System.out.println("Ime: "+ime);  
        System.out.println("Prezime: "+prezime);  
        System.out.println("JMBG: "+jmbg);  
    }  
}  
  
class Djak extends Osoba {  
  
    double prosecnaOcena;  
  
    Djak(String ime, String prezime, String jmbg, double prosecnaOcena){  
        //Poziva se konstruktor nadklase  
        //koriscenjem reci "super".  
        super(ime,prezime,jmbg);  
  
        if(prosecnaOcena<=5 && prosecnaOcena>=1)  
            this.prosecnaOcena = prosecnaOcena;  
        else System.out.println("Greska!");  
    }  
}
```

```

void ispisi(){
    //Pozivanje metode "ispisi" nadklase
    //(klase Osoba) koja ce da uradi ispis
    //prva tri podatka: ime, prezime, jmbg.
    //Na ovaj nacin, kod se ne ponavlja.
    super.ispisi();

    //Ispis prosecne ocene
    System.out.println("Prosecna ocena: "+prosecnaOcena);
}

}

class Penzioner extends Osoba {

    double penzija;

    Penzioner(String ime, String prezime, String jmbg, double penzija) {
        super(ime, prezime, jmbg);
        if (penzija>0) this.penzija = penzija;
        else System.out.println("Greska!");
    }

    void ispisi(){
        //Ne vrsti se poziv metodi "ispisi"
        //nadklase jer ona ispisuje i jmbg
        //a to u ovom slucaju nije potrebno.
        System.out.println("Ime: "+ime);
        System.out.println("Prezime: "+prezime);
        System.out.println("Penzija: "+penzija);
    }
}
class TestOsoba {

    public static void main(String[] args) {
        Osoba o = new Osoba("Pera","Peric","1212007710567");

        Djak dj = new Djak("Mika","Lazic","1010000715076",5.0);

        Penzioner p = new Penzioner("Zika","Zikic",
                                    "0909944710078",23400.0);

        //Poziva se metoda "ispisi" klase Osoba.
        o.ispisi();

        //Poziva se redefinisana metoda "ispisi"
        //klase Djak.
        dj.ispisi();

        //Poziva se redefinisana metoda "ispisi"
        //klase Penzioner.
        p.ispisi();
    }
}

```

Из примера се може уочити неколико ствари. Са обзиром на то да класа Ђак наслеђује класу Особа, и да класа Особа има експлицитно написан параметризовани конструктор, и класа Ђак мора да има експлицитно написан конструктор који позива конструктор надкласе. Овај позив се врши у првој линији кода конструктора класе Ђак коришћењем речи “super” (линије које садржи коментаре се не рачунају као код). Овим се промовише поновно коришћење кода јер оригинални конструктор врши постављање вредности за атрибуте “име”, “презиме” и “јмбг”. Према томе, у конструктор класе Ђак је било доволично додати код за проверу и унос просечне оцене.

У класи Ђак се редефинише метода “испиши”. Са обзиром на то да оригинална метода (метода “испиши” класе Особа) већ исписује на екрану име презиме и јмбг, редефинисана метода позива ову оригиналну методу а додатно је написан само онај део кода који се односи на испис просечне оцене.

Редефинисана метода “испиши” из класе Пензионер демонстрира случај у којем није потребно позвати оригиналну методу. Разлог је тај што се не тражи испис вредности свих атрибуута већ само имена, презимена и износа пензије.

Већ је наведено да подкласа путем наслеђивања добија све атрибууте и методе надкласе. То значи да подкласа има све елементе као и надкласа, са тим што може имати и додатне атрибууте и методе. Другим речима, подкласа и надкласа су донекле компатибилне. Због тога је **могуће променљивој типу надкласе доделити објекат подкласе**. Ово је **компатибилност класа**.

```
NazivNadklase objekat1 = new NazivPodklase();
```

Ако се променљивој типу надкласе додели објекат подкласе, преко те променљиве се **могуће позивати само методе надкласе иако је то, у суштини, објекат подкласе**. Ако се жели повратити пуна функционалност створеног објекта, потребно је извршити експлицитно конвертовање у подтип (“class cast”) на следећи начин:

```
NazivPodklase objekat2 = (NazivPodklase) (objekat1);
```

Потребно је напоменути и следећу ствар: позиви метода подкласе преко променљиве типа надкласе ће да **активирају редефинисане методе из подкласе (ако постоје), а не оригиналне**. На овај начин је у Јави омогућен **полиморфизам** - појава да се може приступати различитим типовима података (класама) преко заједничког интерфејса (заједничке надкласе).

Пример 57

Направити класу Возило која има:

- Атрибут назив чија је почетна вредност “Н”.
- Методу поставиНазив која као параметар прима назив возила и поставља вредност атрибута назив на ту вредност.
- Методу испиши која на екрану исписује назив возила.

Направити класу ТркачкоВозило која наслеђује класу возило и има:

- Атрибут максималнаБрзина чија почетна вредност је 0.0.
- Методу поставиМаксималнуБрзину која као параметар прима максималну брзину и поставља вредност атрибута максималнаБрзина на ту вредност.
- Редефинисану методу испиши која на екрану исписује назив возила и његову максималну брзину.

Направити класу ТестВозила која креира један објекат класе ТркачкоВозило, али тако да буде додељен променљивој типу Возило. Назив возила би требало да буде “Audi S8 GT”.

Позвати методу за испис и видети шта се дешава. Покушати са уносом максималне брзине од 350 км/х. Извршити конвертовање већ постојећег објекта у објекат класе TrkackoVozilo, и доделити максималну брзину па поново извршити испис.

```

class Vozilo {

    String naziv = "N";

    void postaviNaziv(String naziv) {
        this.naziv = naziv;
    }

    void ispisi(){
        System.out.println("Naziv vozila: "+naziv);
    }

}

class TrkackoVozilo extends Vozilo{

    double maksimalnaBrzina = 0.0;

    void postaviMaksimalnuBrzinu(double maksimalnaBrzina) {
        this.maksimalnaBrzina = maksimalnaBrzina;
    }

    void ispisi(){
        super.ispisi();
        System.out.println("Maksimalna brzina je: "+maksimalnaBrzina);
    }

}

class TestVozilo {

    public static void main(String[] args) {
        Vozilo v = new TrkackoVozilo();

        v.postaviNaziv("Audi S8 GT");

        //Ne moze se pozvati metoda
        //"postaviMaksimalnuBrzinu" jer je
        //to metoda podklase TrkackoVozilo
        //i ne postoji u nadklasi Vozilo.
        //v.postaviMaksimalnuBrzinu();
        //Pozvace se metoda "ispisi" klase
        //TrkackoVozilo, a ne metoda "ispisi"
        //klase Vozilo.
        v.ispisi();

        //Konvertovanje postojeceg objekta u
        //klasu TrkackoVozilo.
        TrkackoVozilo tv = (TrkackoVozilo) (v);

        //Tek se sada moze pozvati metoda
        //postaviMaksimalnuBrzinu.
        tv.postaviMaksimalnuBrzinu(350);
    }
}

```

```
        tv.ispisi();  
    }  
}
```

У оквиру “*main*” методе класе *ТестВозило* се креира један објекат класе *ТркачкоВозило* али се додељује променљивој типу *Возило*. Ове две класе су међусобно компатибилне, па је овакво додељивање дозвољено. Једино ограничење које се јавља је то да се сада објекат “гледа кроз фокус класе *Возило*” па се не могу позивати методе које не припадају овој класи. Према томе, позив методе “*поставиМаксималнуБрзину*” није могућ. Тек када се експлицитно изврши конвертована већ постојећег објекта у објекат класе *ТркачкоВозило*, могуће је позвати и ову методу.

Потребно је додати и то да се при сваком позиву методе “*испиши*” позива редефинисана метода из класе *ТркачкоВозило*, а не оригинална метода “*испиши*” класе *Возило*.

Резервисана реч “final” има своје специфично значење у контексту наслеђивања. Свака класа која се означи са овом речи **не може да се даље наслеђује тј. не може бити надкласа**.

```
final class NazivKlase {  
    ...  
}  
  
//Nije dozvoljeno nasleđivanje prethodne klase  
  
//class NazivKlase2 extends NazivKlase1 {  
//  
//    ...  
//  
// }
```

6.2 Апстрактне класе

Већ је речено да се метода састоји из заглавља (повратна вредност, назив, параметри) и тела (код који обезбеђује функционалност). У овом поглављу је наведено да се атрибути и методе преносе наслеђивањем и да је један од основних мотива наслеђивања избегавање поновног писања неког кода који је заједнички за више класа. Међутим, како поступити у оним ситуацијама када је унапред познато да ће класа морати да садржи одређене методе (са већ дефинисаним заглављем) али имплементација тих метода може да варира у зависности од неких фактора?

Јава садржи механизам апстрактних класа које омогућавају решавање оваквих проблема. **Апстрактне класе** представљају класе које поред обичних метода и атрибута имају и једну или више апстрактних метода. **Апстрактне методе** су оне методе које имају само заглавље али не и тело. **Класа која наслеђује апстрактну класу мора да имплементира све апстрактне методе те класе иначе и она постаје апстрактна**. Дефиниција апстрактне класе се врши коришћењем **резервисане речи “abstract”**. Ова резервисана реч се мора написати и на почетку заглавља сваке апстрактне методе у оквиру апстрактне класе, а уместо тела методе се пише знак тачка-зарез (“;”).

```
abstract class NazivApstraktneKlase {

    tip_atributA nazivAtributA;
    ...
    tip_atributN nazivAtributaN;

    tip_povratne_vrednosti nazivMetode1() {
        ...Telo metode 1...
    }

    ...

    tip_povratne_vrednosti nazivMetodeN() {
        ...Telo metode N...
    }

    abstract tip_povratne_vrednosti nazivApstraktneMetode1();

    ...

    abstract tip_povratne_vrednosti nazivApstraktneMetodeN();
}
```

Веома је важно напоменути да се **не може направити објекат (инстанца) апстрактне класе**. Разлог је тај што би позив апстрактне методе водио ка недефинисаном понашању. Следећа наредба није дозвољена, и била би пријављена као грешка.

```
NazivApstraktneKlase promenljival = new NazivApstraktneKlase();
```

Са друге стране, и овде важи компатибилност класа па је дозвољено променљивој која је типа апстрактне класе доделити објекат подкласе (обичне Јава класе).

```
NazivApstraktneKlase promenljival = new NazivKlase();
```

Свака апстрактна класа представља својеврstan уговор јер програмер који пише класу која наслеђује апстрактну класу мора да имплементира задате апстрактне методе. Подкласа, заузврат, наслеђивањем добија све атрибуте и методе које нису апстрактне.

Пример 58

Направити апстрактну класу Запослени која има:

- Атрибут плата који је реалан број.
- Методу *getPlata* која враћа тренутну вредност атрибута плата.
- Апстрактну методу израчунајПлату која не враћа ништа, али има параметар који представља број сати који је запослени радио.

Направити класу КанцеларијскиРадник која наслеђује класу Запослени и:

- Реализује методу израчунајПлату. Ова метода поставља вредност атрибута плата према формулама *ПЛАТА = 100 * БРОЈ_САТИ*.

Направити класу Менаджер која наслеђује класу Запослени и:

- Реализује методу израчунајПлату. Ова метода поставља вредност атрибута плата према формулама *ПЛАТА = 1000 * БРОЈ_САТИ*.

Направити класу `TestZaposleni` и креирати по један објекат класе КанцеларијскиРадник и Менаджер. Израцујати и исписати плате овојиће ако се зна да је менаджер радио 250 сати а канцеларијски радник 200 сати.

```
abstract class Zaposleni {  
  
    double plata;  
  
    double getPlata() {  
        return plata;  
    }  
  
    abstract void izracunajPlatu (int broj_sati);  
}  
  
class KancelarijskiRadnik extends Zaposleni {  
  
    void izracunajPlatu(int broj_sati) {  
        plata = 100 * broj_sati;  
    }  
}  
  
class Menadzer extends Zaposleni {  
  
    void izracunajPlatu(int broj_sati) {  
        plata = 1000 * broj_sati;  
    }  
}  
  
class TestZaposleni {  
  
    public static void main(String[] args) {  
        KancelarijskiRadnik k = new KancelarijskiRadnik();  
        Menadzer m = new Menadzer();  
  
        m.izracunajPlatu(250);  
        k.izracunajPlatu(200);  
  
        System.out.println("Plata menadzera je: "+m.getPlata());  
        System.out.println("Plata kancelarijskog radnika je: "+k.getPlata());  
    }  
}
```

Класа `Запослени` је апстрактна. Поред атрибута "плате" и методе "getPlata" она има и апстрактну методу "израчунајПлату". Са обзиром на то да класе КанцеларијскиРадник и Менаджер наслеђују класу `Запослени`, оне морају да имплементирају апстрактну методу "израчунајПлату". Плате менаджера и канцеларијског радника се израчунају на другачији начин, па су и имплементације конкретних метода "израчунајПлату" посебне за сваку класу.

6.3 Интерфејси

Интерфејси представљају корак даље у односу на апстрактне класе. Код интерфејса су све

методе апстрактне тј. немају тело већ само заглавље. Због тога се испред методе у интерфејсу чак **ни не наводи резервисана реч “abstract”**. Као и апстрактне класе, ни интерфејси **не могу да имају појављивања**. Поред тога што немају ниједну конкретну методу, интерфејси **не могу да имају ни атрибуте већ само константе**. Дефиниција интерфејса се врши коришћењем **резервисане речи “interface”**.

```
interface NazivInterfejsa {
    tip_povratne_vrednosti nazivMetode1();
    ...
    tip_povratne_vrednosti nazivMetodeN();
}
```

Интерфејс је концепт уз помоћу кога се **раздваја спецификација метода (назив, повратна вредност, параметри) од њихове имплементације**. Свака класа која имплементира интерфејс **обавезно мора да садржи имплементације свих његових метода**. Једна класа може истовремено **да имплементира више интерфејса**, и то се означава уз помоћ **резервисане речи “implements”**.

```
class NazivKlase implements NazivInterfejsa1, NazivInterfejsa2... {
    ...
}
```

Могуће су чак и ситуације у којима класа наслеђује неку другу класу и истовремено имплементира један или више интерфејса.

```
class NazivKlasa1 extends NazivKlase2 implements NazivInterfejsa1,
NazivInterfejsa2... {
    ...
}
```

Још је потребно напоменути и то да методе које имплементирају методе из интерфејса **морају да буду јавне тј. означене резервисаном речи “public”** (више о пакетима и нивоима приступа у наредном поглављу). Иако интерфејси нису класе у уобичајеном смислу, и код њих **важи компатибилност типова**. У контексту интерфејса то значи да се променљивој која је типа неког интерфејса увек може проследити објекат било које класе која наслеђује тај интерфејс. Наравно, у том случају се могу позивати само оне методе које се налазе у интерфејсу.

Пример 59

Направити интерфејс АутоматНовцаИнтерфејс који има:

- Методу уложиНовац која не враћа ништа, а има као параметар износ који се жели унети (нпр. 123,45 дин).
- Методу подигниНовац која не враћа ништа, а има као параметар износ који се жели подићи (нпр. 123,45 дин).

Направити класу АутоматНовца која имплементира интерфејс АутоматНовцаИнтерфејс и има:

- Атрибут стање који представља количину новца која се налази у аутомату. Почетно стање је 5000.0 дин.
- Имплементирану методу уложиНовац. Ова метода врши улагање тј. додавање унетог износа на тренутно стање само ако је унети износ већи од нуле. У

- *супротном, исписује се порука о грешци.*
- *Имплементирану методу подигниНовац. Ова метода врши подизање жељеног износа тј, смањивање стања само ако је унети износ већи од нуле и ако у аутомату има довољно новца. У супротном, исписује се порука о грешци.*
- *Методу испиши која исписује колико још новца има у аутомату.*

Направити класу НапредниАутоматНовца која имплементира интерфејс АутоматНовцаИнтерфејс и има:

- *Атрибут стање који представља количину новца која се налази у аутомату. Почетно стање је 20000.0 дин.*
- *Имплементирану методу уложиНовац. Ова метода врши улагање тј. додање унетог износа на тренутно стање. Максимални износ који се може уплатити је 20.000 динара. Ако се овај износ прекорачи, потребно је исписати поруку о томе да је максимални износ који се може уплатити 20.000 динара. Ако је унета вредност за износ мања од нуле исписује се порука о грешци.*
- *Имплементирану методу подигниНовац. Ова метода врши подизање жељеног износа тј, смањивање стања. Максимални износ који се може подићи је 10.000 динара. Ако се овај износ прекорачи, потребно је исписати поруку о томе да је максимални износ који се може подићи 10.000 динара. Ако је унета вредност за износ мања од нуле исписује се порука о грешци.*
- *Методу испиши која исписује колико још новца има у аутомату и који су максимални износи који се могу уплатити тј. подићи у једној трансакцији.*

Направити класу ТестАутоматНовца која прави по један објекат класе АутоматНовца и НапредниАутоматНовца и улаже у први 3000 дин, а из другог подиже 9.999 динара. Исписати стање сваког аутомата новца.

```
public interface AutomatNovcaInterfejs {  
  
    public void uloziNovac(double iznos);  
    public void podigniNovac (double iznos);  
  
}  
  
public class AutomatNovca implements AutomatNovcaInterfejs{  
  
    double stanje = 5000.0;  
  
    public void uloziNovac(double iznos){  
        if (iznos > 0) stanje += iznos;  
        else System.out.println("Greska");  
    }  
  
    public void podigniNovac(double iznos){  
        if (iznos > 0 && stanje >= iznos) stanje -= iznos;  
        else System.out.println("Greska");  
    }  
  
    void ispisi(){  
        System.out.println("U automatu ima jos "+stanje+" dinara");  
    }  
  
}  
  
public class NapredniAutomatNovca {  
  
    double stanje = 20000.0;
```

```

public void uloziNovac(double iznos) {
    if (iznos > 0) {
        if (iznos <= 20000) stanje += iznos;
        else System.out.println("Maksimalni iznos"+
            " koji se moze uplatiti je 20000 dinara");
    }
    else System.out.println("Greska");
}

public void podigniNovac(double iznos) {
    if (iznos > 0 && stanje >= iznos){
        if (iznos <= 10000) stanje -= iznos;
        else System.out.println("Maksimalni iznos"+
            " koji se moze podici je 10000 dinara");
    }
    else System.out.println("Greska");
}

void ispisi(){
    System.out.println(
        "U automatu ima jos "+stanje+" dinara");
    System.out.println(
        "Maksimalni iznos koji se moze uplatiti je 20000 dinara");
    System.out.println(
        "Maksimalni iznos koji se moze podici je 10000 dinara");
}

}

public class TestAutomatNovca {

    public static void main(String[] args) {
        AutomatNovca a = new AutomatNovca();
        a.ulozNovac(3000);

        NapredniAutomatNovca na = new NapredniAutomatNovca();
        na.podigniNovac(9999);

        a.ispisi();
        na.ispisi();
    }

}

```

Класе *АутоматНовца* и *НапредниАутоматНовца* имплементирају интерфејс *АутоматНовцаИнтерфејс*, па морају да садрже методе “уложиНовац” и “подигниНовац” које морају да буду означене као јавне (“public”).

6.4 Класа Object

Класа **Object** је, у Јави, надкласа свих класа. Иако се ни у једној Јава класи експлицитно не наводи ово наслеђивање, оно је увек присутно. Природна последица тога је да **променљива** типа **Object** може да прими инстанцу **било које друге Јаве класе** (компабилност типова). Ово може да буде изузетно корисно у оним ситуацијама када није унапред познато ког ће типа бити нека променљива или параметар (погледати “equals” методу даље у тексту).

Класа `Object` није ни апстрактна класа ни интерфејс већ обична класа са конкретним атрибутима и методама. У наредних неколико пасуса ће детаљно бити објашњене само две методе које се често користе: “`toString`” и “`equals`”.

Идеја методе “`toString`” је да врати вредности свих атрибута класе спојене у један `String`. На овај начин се лако и брзо може приказати унутрашње стање сваког објекта на екрану. У оригиналној форми, ова метода враћа само `String` са називом класе и меморијском адресом објекта који је позвао методу. Другим речима, да би метода “`toString`” била корисна, **потребно ју је редефинисати** тако да враћа `String` са вредностима одговарајућих атрибута конкретне класе у којој се налази. Заглавље ове методе је увек исто и гласи:

```
public String toString()
```

Метода “`equals`” је замишљена тако да пореди два објекта исте класе и враћа “`true`” ако су једнаки, а “`false`” ако нису. Једнакост два објекта исте класе зависи од тога о којој класи се ради: два аутомобила су “једнака” ако имају исту регистрацију, две особе су “једнаке” ако имају исти матични број, итд. У својој оригиналној форми, ова метода пореди меморијске локације оба објекта и сматра их једнаким ако се налазе на истој позицији у меморији. Слично као и са “`toString`” методом, и “`equals`” методу је потребно редефинисати да би вршила упоређивање на прави начин. Ова метода је већ коришћена у претходним примерима када је било потребно поредити две `String` вредности. Редефинисана варијанта “`equals`” методе из класе `String` пореди две `String` вредности слово по слово “`true`” ако су једнаки, а “`false`” ако нису. Заглавље “`equals`” методе гласи овако:

```
public boolean equals (Object o)
```

Може се видети да метода као параметар има објекат типа `Object`. Ово је и добро и лоше. Добро је због тога што се методи може проследити објекат било које класе као параметар (`Object` је надкласа свих класа) па је овакво заглавље одговарајуће за било коју класу. Лоше је због тога што се, пре било какве провере, унети објекат мора експлицитно конвертовати у одговарајући тип.

Када се говори о конвертовању типова, потребно је напоменути и да постоји оператор којим се може проверити које је класе неки објекат. У питању је “**instanceof**” оператор. Следећи израз враћа “`true`” ако је објекат заиста те класе, а “`false`” ако није.

```
објекат instanceof NazivKlase
```

Шта се заправо добија коришћењем ове две методе? Оне би требало да послуже као конвенција тј. оријентир при програмирању: ако се желе упоредити два објекта, потребно је позвати “`equals`” методу а ако се жели на екрану исписати унутрашње стање неког објекта, потребно је позвати “`toString`” методу.

Пример 60

Направити класу Књига која има:

- Атрибут **наслов**.
- Атрибут **автор** који садржи име и презиме аутора.
- Атрибут **ИСБН** који представља јединствени идентификатор књиге (`String`).
- Методе `getNaslov`, `getAutor` и `getISBN` које враћају вредности одговарајућих атрибута.
- Методе `setNaslov`, `setAutor` и `setISBN` које као параметре добијају нове вредности за

- одговарајуће атрибуте и постављају их.
- Редефинисану методу `toString` класе `Object` која враћа један `String`. Овај `String` садржи податке о књизи: наслов, аутор и ИСБН уз одговарајући текст.
- Редефинисану методу `equals` класе `Object` која као параметар прима објекат класе `Object`, али се сматра да ће се заиста уносити објекти класе Књига. Метода враћа `true` ако је вредност атрибута ИСБН једнака ИСБН књиге која је унета као параметар. У супротном, метода враћа `false`.

Направити класу `TestKnjiga` која креира два објекта класе Књига. Прва књига је “Историја лепоте”, њен аутор је Умберто Еко, а ИСБН је 0-234-567. Друга књига је “Историја ружноће”, њен аутор је такође Умберто Еко, а ИСБН је 0-567-890. Проверити да ли је у питању иста књига и исписати поруку о томе. Након тога, исписати податке о свакој књизи на екрану.

```
public class Knjiga {

    String naslov;
    String autor;
    String ISBN;

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        if (autor != null) this.autor = autor;
        else System.out.println("Greska");
    }

    public String getISBN() {
        return ISBN;
    }

    public void setISBN(String isbn) {
        if (isbn != null) ISBN = isbn;
        else System.out.println("Greska");
    }

    public String getNaslov() {
        return naslov;
    }

    public void setNaslov(String naslov) {
        if (naslov != null) this.naslov = naslov;
        else System.out.println("Greska");
    }

    public String toString(){
        return "Naslov: '" + naslov + "' Autor: " + autor + " ISBN " + ISBN;
    }

    // "equals" metoda poredi dva objekta iste klase i vraca true
    // ako su jednaki a false ako nisu.
    public boolean equals (Object o){
        //Ovako se vrši konvertovanje (cast) iz klase Object u
        //klasu Knjiga. Konvertovanje je moguce samo ako se kao
        //parametar zaista unese objekat kl. Knjiga. Ako
        //se unese objekat bilo koje druge klase, metoda ce da
        //prekine sa radom i pojave se greska.
        Knjiga k = (Knjiga) (o);
    }
}
```

```
        if (ISBN.equals(k.getISBN())) return true;
        else return false;
    }

}

public class TestKnjiga {

    public static void main(String[] args) {
        Knjiga k1 = new Knjiga();
        k1.setNaslov("Istorijska lepotica");
        k1.setAutor("Umberto Eko");
        k1.setISBN("0-234-567");

        Knjiga k2 = new Knjiga();
        k2.setNaslov("Istorijski ruznoce");
        k2.setAutor("Umberto Eko");
        k2.setISBN("0-567-890");

        if (k1.equals(k2)) System.out.println("Knjige su iste");
        else System.out.println("Knjige nisu iste");

        System.out.println(k1.toString());
        System.out.println(k2.toString());

        //Moglo je i ovako jer bi se metoda
        //toString automatski pozvala
        //System.out.println(k1);
        //System.out.println(k2);
    }

}
```

7 Нивои приступа, пакети и JavaBeans спецификација

Јава омогућава ограничавање приступа класама или њиховим елементима према више нивоа, па су Јавини модификатори приступа обрађени као прва тема у оквиру овог поглавља. У веома близкој вези са нивоима приступа су и пакети. Груписање класа по сличности или функцији у пакете омогућава лакшу организацију извornog кода, док пакети у комбинацији са модификаторима приступа још прецизније ограничавају приступ класама и њиховим елементима. На крају поглавља се обрађује JavaBeans спецификација, као један општи водич који би требало поштовати при писању кода.

7.1 Нивои приступа

Локалне променљиве и параметри су, као што је већ наведено, “видљиви” у оквиру методе у којој су декларисани, док се атрибути и методе класе могу позвати било где у оквиру класе, али и изван ње. Међутим, у неким ситуацијама је потребно ограничiti приступ неком атрибуту или методи тј. смањити им “видљивост” ван класе. У Јави је овај проблем решен увођењем **нивоа приступа** и одговарајућих резервисаних речи које представљају **модификаторе приступа** (“access modifiers”). Следи табела нивоа и модификатора приступа са њиховим описима.

| Ниво приступа | Модификатор приступа (резервисана реч) | Опис |
|--------------------------|--|---|
| Јавни | public | Класа, атрибут или метода који су означени на овај начин су видљиви свуда и јавно су доступни. |
| Приватни | private | Класа, атрибут или метода који су означени на овај начин су видљиви само у оквиру “java” фајла у којем су написани. |
| Подразумевани (пакетски) | (не пише се ништа) | Класа, атрибут или метода који су означени на овај начин су видљиви само у оквиру пакета у којем се налазе. |
| Заштићени | protected | Исто као и пакетски ниво приступа, само што се видљивост проширује и на све класе из других пакета које наслеђују класу која је заштићена или садржи заштићени елемент. |

Модификатори приступа се могу навести на почетку дефиниције класе, атрибута или методе и онда се односе на ту класу, атрибут тј. методу.

```
modifikator_pristupa1 class NazivKlase {
    modifikator_pristupa2 tip_atributa nazivAtributa;
    modifikator_pristupa3 tip_vrednosti nazivMetode(....) {
        ...
    }
}
```

```
}
```

Који су практични аспекти нивоа приступа и када се они заиста користе? Ниво приступа омогућавају спровођење **принципа учаурења података (енкапсулација, “encapsulation”)** у Јава програмима. Ово је принцип који се најчешће повезује са објектно оријентисаним програмирањем и подразумева да се **подацима не може приступати директно већ индиректно преко позива неке методе**. Имплементација учаурења се најчешће изводи тако што се **атрибутима класе додељују приватни ниво приступа па им се може приступити искључиво преко одговарајућих јавних метода**. Ове јавне методе најчешће садрже и код за логичку контролу, па било каква измена вредности атрибута мора да прође одређене провере тј. не могу се унети недозвољене вредности.

Пример 61

Направити јавну класу АутоматНовца која има :

- Приватни атрибут стање који представља количину новца која се налази у аутомату. Почетно стање је 5000.0 дин.
- Јавну методу уложиНовац која као параметар има износ новца који се жели уложити. Ова метода врши улагање тј. додајање унетог износа на тренутно стање само ако је унети износ већи од нуле. У супротном, исписује се порука о грешци.
- Јавну методу подигниНовац која као параметар има износ новца који се жели подићи. Ова метода врши подизање жељеног износа тј. смањивање стања само ако је унети износ већи од нуле и ако у аутомату има довољно новца. У супротном, исписује се порука о грешци.
- Јавну методу испишиСтанje која исписује колико још новца има у аутомату.

Направити јавну класу ТестАутоматНовца која прави објекат класе АутоматНовца и позива његове методе.

```
public class AutomatNovca {  
  
    private double stanje = 5000;  
  
    public void uloziNovac(double iznos){  
        if (iznos <= 0) System.out.println("Greska");  
        else stanje+=iznos;  
    }  
  
    public void podigniNovac(double iznos){  
        if (iznos <= 0) System.out.println("Greska");  
        else {  
            if (stanje >= iznos) stanje-=iznos;  
            else System.out.println("Zao mi je, nema dovoljno  
novca");  
        }  
    }  
  
    public void ispisiStanje(){  
        System.out.println("U automatu trenutno ima: "+stanje+"  
dinara");  
    }  
}  
  
public class TestAutomatNovca {
```

```

public static void main(String[] args) {
    AutomatNovca a = new AutomatNovca();

    //Ne moze se direktno pristupiti atributu stanje
    //jer je privatn. Zbog toga sledaca komanda nije
    //dozvoljena.
    //a.stanje = 5000;

    //Atributu se jedino moze pristupiti preko metoda
    //"uloziNovac" i "podigniNovac" koje vrse logicku
    //kontrolu pre nego sto promene atribut stanje.
    a.ulaziNovac(430);
    a.podigniNovac(222.34);
    a.ispisStanje();
}
}

```

У задатку се тражило да класа *АутоматНовца* буде јавна, па је испред дефиниције ове класе додат модификатор “*public*”. Атрибут “*станje*” је приватан (“*private*”), а методе “*улозиНовца*”, “*подигниНовца*” и “*испишиСтанје*” су јавне. Класа *АутоматНовца* поштује принцип учаурења јер се атрибуту “*станje*” не може приступити директно већ искључиво преко метода. То се види у коду “*main*” методе класе “*ТестАутоматНовца*” где се неуспешно покушава са директним позивом. Тиме се онемогућава директно уношење недозвољених вредности за овај атрибут (нпр. -1000 динара). Пре него што се *станје* заиста промени, методе “*улозиНовца*” и “*подигниНовца*” врше логичке провере (да ли је износ позитиван и да ли има довољно новца) чиме се фактички онемогућава да атрибут “*станје*” чак и индиректно добије неку недозвољену вредност.

7.2 JavaBeans спецификација

Са обзиром на то да учаурење подразумева сакривање података и то тако да им се може приступити једино преко позива неке методе, појављује се нови проблем. Како дати назив методи која нпр. враћа вредност неког атрибута? Неко би предложио да се метода назове “*врати*****”, други би је назвали “*узми*****” док би понеко предложио назив “*дај*****”. Проблем се састоји у томе што не постоји јединство при давању назива, па би се врло вероватно догодило да постоји више назива за методе које раде исто. Решење представља поштовање **JavaBeans спецификације**. Ова спецификација одговара на многа питања, али регулише и питање назива појединих метода.

Ако нека метода само враћа вредност одређеног атрибута, она би, према JavaBeans спецификацији, **требало да се назива “get****” при чему остатак назива чини назив атрибута**. Метода која, на пример, враћа вредност атрибута “*program*” би требало да се назива “*getProgram*”. Једини **изузетак су “get” методе за “boolean” атрибуте које би требало да се називају “is****”**.

Ако нека метода само поставља нову вредност одређеног атрибута, она би, према JavaBeans спецификацији, **требало да се назива “set****” при чему остатак назива чини назив атрибута**. Метода која, на пример, поставља нову вредност атрибута “*program*” би требало да се назива “*setProgram*”.

JavaBeans спецификација поштује принцип учаурења, па важи то да би сви атрибути требало да буду приватни, а све “*get*” и “*set*” методе јавне.

Пример 62

Направити јавну класу Особа која има :

- Приватни атрибут име.
- Приватни атрибут презиме.
- Приватни атрибут брачноСтање који може имати вредност TRUE ако је особа ожењена или удата а FALSE ако није.
- Одговарајуће јавне get и set методе за ове атрибуте.

```
public class Osoba {

    private String ime;
    private String prezime;
    private boolean bracnoStanje;

    public String getIme() {
        return ime;
    }

    public void setIme(String ime) {
        this.ime = ime;
    }

    public String getPrezime() {
        return prezime;
    }

    public void setPrezime(String prezime) {
        this.prezime = prezime;
    }

    public boolean isBracnoStanje() {
        return bracnoStanje;
    }

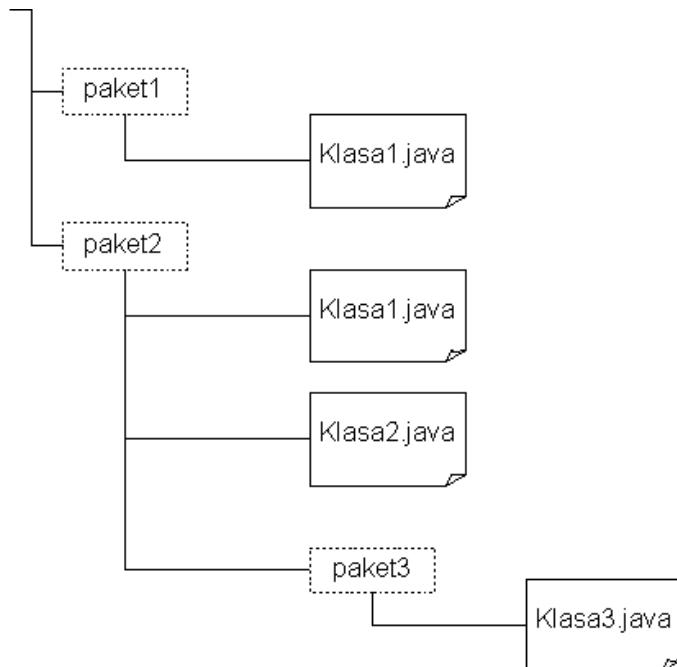
    public void setBracnoStanje(boolean bracnoStanje) {
        this.bracnoStanje = bracnoStanje;
    }
}
```

7.3 Пакети

Већ је објашњено да се у једној класи могу појавити две методе са истим називом, али да се морају разликовати по параметрима. Такође, при редефинисању методе се пише нова метода са истим заглављем, али другачијом имплементацијом. Међутим, да ли је могуће у оквиру истог програма написати две класе са истим називом?

Одговор на ово питање је потврдан. У Јави постоји концепт **пакета** који је умногоме сличан концепту директоријума (“folder”) на хард диску рачунара (пакети се, у суштини, и имплементирају као директоријуми). Сваки пакет представља **целину која може да садржи одговарајуће фајлове (.java”, “.class” или и друге) или и друге пакете (подпакете)**. Према

тome, пакети **могу да се организују у хијерархију**. У Јава програму се могу појавити две класе са истим називом али морају да буду у различитим пакетима (Слика 22).



Слика 22: Хијерархија пакета

Ово, наравно, није једини мотив за увођење пакета. Пакети су **веома погодни за груписање класа** према сличности или функцији коју обављају. На овај начин се програми који садрже велики број класа организују у више мањих, хомогених целина. Штавише, пакети могу да се користе у комбинацији са модификаторима приступа да би се ограничио приступ неким класама и њиховим елементима.

Према конвенцији, **називи пакета се пишу искључиво малим словима**. Раздавање између пакета и његових подпакета се врши знаком тачке (“.”) па би се пакет “људи” унутар пакета “антропологија” означавао као “антропологија.људи”.

У досадашњим примерима и задацима, класе нису припадале ниједном конкретном пакету, па их је Јава програмски језик аутоматски смештао у подразумевани пакет (“default package”). Ако класа припада неком конкретном пакету, онда се то мора назначити на првој линији кода у “java” фајлу у којем се налази класа и то пре дефиниције саме класе. У том случају се пише резервисана реч **“package”** после које следи назив пакета.

```

package nazivpaketa;

class NazivKlase {
    ...
}
  
```

Ако се у програму користи једна или више класа које припадају експлицитно дефинисаним пакетима, онда је пре њиховог коришћења потребно тзв. **увођење (“import”)**. У ту сврху се користи “import” наредба. Ова наредба се пише пре дефиниције класе или после декларације пакета (ако постоји). После резервисане речи “import” се наводи **пун назив класе** која се жели увести. Пун назив класе се састоји из назива пакета којем класа припада и назива класе. У једном “.java” фајлу **може бити нула, једна или више “import” наредби**.

```
package nazivpaketa1;

import nazivpaketa2.NazivKlase2;
...
import nazivpaketaN.NazivKlaseN;

class Nazivklasel {
    ...
}
```

Ако је истовремено потребно увести све класе које припадају једном пакету у програм, не морају се написати посебне “import” наредбе, **већ се може искористити знак звезда (“*”)**.

```
package nazivpaketa1;

import nazivpaketa2.*;

class Nazivklasel {
    ...
}
```

Пример 63

Направити јавну апстрактну класу Инструмент у пакету инструменти која има :

- Јавну апстрактну методу испишиВрсту која нема параметара и не враћа ништа.

Направити јавну класу Флаута у пакету инструменти.дувачкиинструменти која наслеђује апстрактну класу Инструмент и има :

- Имплементирану јавну методу испишиВрсту која на екрану исписује да је у питању дувачки инструмент.

Направити јавну класу Виолина у пакету инструменти.гудачкиинструменти која наслеђује апстрактну класу Инструмент и има :

- Имплементирану јавну методу испишиВрсту која на екрану исписује да је у питању гудачки инструмент.

Направити јавну класу ТестИнструменти у пакету тест која креира по један објекат класе Флаута и Виолина и позива методе за испис врсте инструмента.

```
package instrumenti;

public abstract class Instrument {

    public abstract void ispisiVrstu();

}

package instrumenti.duvackiinstrumenti;

import instrumenti.Instrument;

public class Flauta extends Instrument {

    public void ispisiVrstu() {
```

```

        System.out.println("U pitanju je duvacki instrument");
    }

}

package instrumenti.gudackiinstrumenti;

import instrumenti.Instrument;

public class Violina extends Instrument {

    public void ispisiVrstu() {
        System.out.println("U pitanju je gudacki instrument");
    }

}

package test;

import instrumenti.duvackiinstrumenti.Flauta;
import instrumenti.gudackiinstrumenti.Violina;

public class TestInstrumenti {

    public static void main(String[] args) {
        Flauta f = new Flauta();
        Violina v = new Violina();

        f.ispisiVrstu();
        v.ispisiVrstu();
    }

}

```

Класа **Инструмент** се налази у пакету “инструменти” што је и назначено преко команде “**package**”. Са обзиром на то да класе **Флаута** и **Виолина** наслеђују ову класу а налазе се у другим пакетима у односу на њу, обе морају прво да увезу ову класу (“**import instrumenti.Instrument;**”). Класа **Флаута** је у пакету “дувачкиинструменти” који се налази у оквиру пакета “инструменти” па је цео назив пакета “инструменти.дувачкиинструменти”. Слично важи и за класу **Виолина**. Класа **ТестИнструменти** користи класе **Флаута** и **Виолина** а не налази се у њиховом пакету (већ у пакету “тест”), па су наведене две “**import**” наредбе на почетку - за сваку класу по једна.

Сад се може објаснити значење и последња два нивоа приступа: подразумеваног (пакетског) и заштићеног.

Подразумевани ниво приступа (пакетски) дефинише да се класа или неки њен елемент који су овако означени, могу користити само у оквиру пакета у којем се налазе. Они нису видљиви ван тог пакета. Овај ниво приступа се добија када се испред дефиниције класе, методе или атрибута **не напише ниједан модификатор приступа**.

Заштићени ниво приступа (“protected”) подразумева да се класа или неки њен елемент који су овако означени, могу користити само у оквиру пакета у којем се налазе, али су видљиви и за оне класе ван пакета које наслеђују означену класу.

8 Низови објеката

У трећем поглављу су детаљно објашњени низови - шта представљају и чemu служе, како се формирају и користе итд. Ту је, међутим, прећутно уведено и једно ограничење - елементи низа су увек били простог типа (char, int, boolean, double). У Јави је могуће направити низ чији елементи нису простог типа, већ је **сваки елемент по један објекат**. Рад са низовима објеката је скоро потпуно исти као и рад са обичним низовима. Једине разлике се своде на то да се сваки појединачни елемент третира мало другачије јер је у питању објекат.

Декларација низа објеката потпуно иста као и за обичан низ:

```
tip_podatka[] nazivPromenljive;
```

Са обзиром на то да су елементи низа објекти неке класе, тип податка је управо назив те класе:

```
NazivKlase[] nazivPromenljive;
```

Иницијализација низа је и у овом случају потребна, и врши се на исти начин као и за обичан низ:

```
nazivPromenljive = new NazivKlase[ceo_broj];
```

Прва разлика у односу на обичне низове се може видети при иницијализацији. Код иницијализације низа чији су елементи простог типа, сама иницијализација низа резервише сав потребан меморијски простор за сваки елемент низа и са низом се одмах може радити. Када је у питању низ објекта, иницијализација низа само резервише меморијски простор за показиваче на објекте, али **не иницијализује саме објекте**. Другим речима, **после иницијализације низа објекта, сваки елемент има вредност "null" и са њим се не може радити док се не иницијализује**. У неким ситуацијама је потребно извршити појединачну иницијализацију елемената одмах на почетку, а некад се само уносе већ иницијализовани објекти на одговарајућа места у низу (у том случају се сматра да је место у низу "празно" ако има "null" вредност).

Приступ елементима низа се врши преко индекса, а максимални капацитет низа се добија позивањем команде "length":

```
nazivPromenljive[indeks]
```

```
nazivPromenljive.length
```

Друга разлика у односу на обичне низове се састоји у томе што се над елементима низа објекта директно могу позивати методе. Сваки елемент је по један објекат, па се позиви његових метода могу извршити овако:

```
nazivPromenljive[indeks].nazivMetode(...argumenti...);
```

Пример 64

Направити јавну класу ТемператураМеста која има:

- Приватни атрибут назив који представља назив места. Почетна вредност за овај атрибут је "непознат".
- Приватни атрибут температура (цео број) који представља дневну температуру за то место.
- Одговарајуће јавне гет и сет методе за ова два атрибута.

Направити јавну класу *ДневнаПрогноза* која има:

- Приватни атрибут температуре који представља низ објеката класе *ТемператураМеста*.
- Јавни конструктор који као параметар прима број који представља максималан број места на које се прогноза односи. Ако је унети број већи од нуле, потребно је иницијализовати атрибут температуре на тај капацитет. Ако је унети број нула или мањи од нуле, капацитет се поставља на 10 и исписује се порука о грешци на екрану. У сваком случају је потребно иницијализовати сваки елемент низа.
- Јавну методу *имаСлободнихМеста* која враћа TRUE ако у низу има слободних места за унос температуре, а FALSE ако нема. Место у низу је слободно ако уместо назива места на које се односи прогноза стоји Стринг "непознат".
- Јавну методу *унеси* која као параметре прима назив места и температуру и унеси те податке на прво слободно место у низу. Ако у низу нема слободних места, метода исписује поруку о томе на екрану.
- Јавну методу *избаци* која као улази параметар прима назив места и избације податке о том месту и његовој температури из низа. Избацује се врши тако што се као назив места поставља реч "непознат" а температура добија вредност 0. Ако у низу нема места са унетим називом, ништа се не дешава.
- Јавну методу *исписи* која на екрану исписује целокупну дневну прогнозу.

Направити класу *ТестДневнаПрогноза* која креира један објекат класе *ДневнаПрогноза* капацитета 3 места и у њега уноси податке о температурама за три места: "Београд" (17Ц), "Нови Сад" (13Ц), "Ниш" (16Ц). Исписати податке о свим градовима и њиховим температурама.

```
public class TemperaturaMesta {
    private String naziv = "nepoznat";
    private int temperatuta;

    public String getNaziv() {
        return naziv;
    }

    public void setNaziv(String naziv) {
        this.naziv = naziv;
    }

    public int getTemperatura() {
        return temperatuta;
    }

    public void setTemperatura(int temperatuta) {
        this.temperatuta = temperatuta;
    }
}

public class DnevnaPrognoza {
    private TemperaturaMesta[] temperature;
```

```
public DnevnaPrognoza(int brojMesta) {
    if (brojMesta > 0)
        temperature = new TemperaturaMesta[brojMesta];
    else {
        temperature = new TemperaturaMesta[10];
        System.out.println("Greska");
    }

    // Pojedinacna inicializacija svakog
    // elementa niza.
    for (int i = 0; i < temperature.length; i++)
        temperature[i] = new TemperaturaMesta();
}

public boolean imaSlobodnihMesta() {
    for (int i = 0; i < temperature.length; i++)
        if (temperature[i].getNaziv().equals("nepoznat"))
            return true;

    return false;
}

public void unesi(String naziv, int temperatura) {
    if (!imaSlobodnihMesta())
        System.out.println("Nema slobodnih mesta");
    else
        for (int i = 0; i < temperature.length; i++)
            if (temperature[i].getNaziv().equals("nepoznat"))
            {
                temperature[i].setNaziv(naziv);
                temperature[i].setTemperatura(temperatura);
                break;
            }
}

public void izbaci(String naziv) {
    for (int i = 0; i < temperature.length; i++)
        if (temperature[i].getNaziv().equals(naziv)) {
            temperature[i].setNaziv("nepoznat");
            temperature[i].setTemperatura(0);
            break;
        }
}

public void ispisi(){
    for (int i = 0; i < temperature.length; i++)
        System.out.println(
            "Mesto: "+temperature[i].getNaziv()+
            " Temperatura: "+
            temperature[i].getTemperatura());
}

public class TestDnevnaPrognoza {
```

```
public static void main(String[] args) {
    DnevnaPrognoza d = new DnevnaPrognoza(3);
    d.unesi("Beograd", 17);
    d.unesi("Novi Sad", 13);
    d.unesi("Nis", 16);

    d.ispisi();
}
```

Конструктор класе *ДневнаПрогноза* иницијализује атрибут “температура” који представља низ објеката класе *ТемператураМеста*. Поред иницијализације низа, овај конструктор иницијализује и сваки елемент низа (објекат) појединачно. То се врши позивањем конструктора класе *ТемператураМеста* за сваки елемент. Тек после ове иницијализације се елементи низа могу користити.

Метода “имаСлободнихМеста” проверава да ли је неко место у низу слободно на тај начин што пореди да ли атрибут “назив” објекта који се налази на том месту у низу има вредност “непознат”. Ту се може видети да се вредност његовог атрибута “назив” добија позивањем методе “getNaziv” (“temperature[i].getNaziv()”). Са обзиром на то да ова метода враћа назив (String) потребно га је упоредити са вредношћу “непознат” што се врши позивањем “equals” методе. Цео логички израз у оквиру IF команде због тога изгледа релативно сложено па гласи: “temperature[i].getNaziv().equals(“nepoznat”)”.

9 Листе

Низови омогућавају рад са више променљивих без потребе за појединачним дефинисањем сваке променљиве. Иако су веома корисни, они имају један озбиљан недостатак: једном када се иницијализују, капацитет им је фиксиран. Проширење или смањење капацитета низа захтева поновну иницијализацију низа што, у општем случају, доводи до брисања свих његових елемената.

Листе (“динамички низови”) немају ограничен капацитет па се елементи могу додавати и брисати по жељи. Листа може да садржи десет, сто или било који произвољан број елемената а да не постоји потреба за поновном иницијализацијом. Додатна погодност је та што се, при избацивању елемента из листе, не остављају “празна” места, па је листа увек цела и без “прекида”.

У Јави постоји неколико предефинисаних класа које, у ствари, представљају листе. Једна од њих је и класа **LinkedList**. Пун назив ове класе је “java.util.LinkedList” и потребно ју је увести у програм пре него што се може користити. Једино ограничење које ова класа уводи је то да **елементи листе не могу да буду простог типа, већ само објекти. Сви елементи листе морају бити објекти исте класе**. Декларација листе се врши на следећи начин:

```
LinkedList <NazivKlaseElementa> nazivPromenljive;
```

При декларацији, после назива класе која представља листу (LinkedList) пише се назив класе чији објекти ће бити елементи листе. Назив класе елемената се пише између знакова “мање од” (“<”) и “веће од” (“>”).

Иницијализација листе је веома слична иницијализацији било ког објекта. Са обзиром на то да листе немају ограничен капацитет, иницијализација је увек иста:

```
nazivPromenljive = new LinkedList <NazivKlaseElementa> ();
```

Ако се ипак жели покушати са уношењем елемената простог типа у листу, могу се користити тзв. **класе за “обмотавање” (“wrapper classes”)**. Као што им назив каже, ове класе представљају “омотач” око вредности простог типа па, у ствари, имају атрибут који је простог типа и у који се смешта одговарајућа вредност. Тако, за “double” вредности постоји класа Double, за “int” класа Integer, за “char” класа Character и за “boolean” класа Boolean.

Пример 65

Следи пар примера коришћења класа за обмотавање простих типова:

```
Double d = new Double (2.3);
System.out.println(d.doubleValue());

Integer i = new Integer (55);
System.out.println(i.intValue());

Character c = new Character ('R');
System.out.println(c.charValue());
```

```
Boolean b = new Boolean (false);
System.out.println(b.booleanValue());
```

Као што се може видети, вредност простог типа се уноси при иницијализацији објекта одговарајуће обмотавајуће класе, док се та вредност преузима позивањем неке од “value” метода (“doubleValue”, “intValue”, “charValue” и “booleanValue”).

Класа LinkedList садржи више готових метода које служе за уношење у листу, избацивање из листе, претраживање итд. Следи табела метода класе LinkedList са одговарајућим описима.

| Метода | Опис |
|-----------------------------|---|
| add (<NazivKlase> e) | Додаје елемент на крај листе (ради исто што и AddLast метода). |
| add (int i, <NazivKlase> e) | Уноси елемент у листу и то на позицију чији је индекс једнак “и” (индекси елемената листе крећу од нуле као код низова). |
| addFirst (<NazivKlase> e) | Додаје елемент на почетак листе. |
| addLast (<NazivKlase> e) | Додаје елемент на крај листе. |
| clear () | Брише све елементе из листе. |
| contains (Object o) | Враћа TRUE ако у листи постоји елемент који је исти као унети објекат, а FALSE ако не постоји (погледати детаљнији опис ниже у тексту). |
| get (int i) | Враћа онај елемент из листе чији је индекс “и”. |
| getFirst () | Враћа први елемент из листе. |
| getLast () | Враћа последњи елемент из листе. |
| remove (int i) | Избацује онај елемент из листе чији је индекс “и”. |
| remove (Object o) | Избацује онај елемент из листе који је једнак унетом објекту. |
| removeFirst () | Избацује први елемент из листе. |
| removeLast () | Избацује последњи елемент из листе. |
| set (int i, <NazivKlase> e) | Замењује елемент чији је индекс једнак “и” унетим објектом. |
| size () | Враћа тренутни број елемената у листи. |

Да би се у потпуности разумеле наведене методе, потребно је знати и следеће. **Сваки елемент листе има свој индекс**. Као и код низова, **индекси почињу од нуле**, па је нпр. индекс петог елемента једнак 4. На тај начин све методе које користе индексе могу да пронашу тачно место у листи на којем је потребно унети неки елемент, изменити га или избацити. Број елемената листе може да варира, али се тренутни број елемената добија позивањем методе “size”. Коначно, потребно је знати и то да метода “contains” функционише тако што пролази кроз листу и упоређује елементе листе са унетим објектом. Кључна ствар је та што ова метода **автоматски позива одговарајућу “equals” методу** да би упоредила објекте. У питању је **“equals” метода класе која је унета као елемент листе**.

Пример 66

Направити јавну класу ЧланКомисије која има:

- Приватни атрибут имеПрезиме.

- Приватни атрибут функција. Почетна вредност овог атрибута је `NULL`.
- Одговарајуће јавне `get` и `set` методе за ова два атрибута. Недозвољене вредности за први атрибут су `NULL` и празан `String ("")`. У случају уноса недозвољених вредности, потребно је исписати поруку о грешци на екрану.
- Редефинисану методу `toString` класе `Object` која враћа `String` са подацима о члану комисије уз одговарајући текст. Ако је вредност атрибута функција једнака `NULL`, метода само враћа податке о имену и презимену члана комисије. Ако није `NULL`, враћају се сви подаци о члану комисије.
- Редефинисану методу `equals` класе `Object` која као параметар прима објекат класе `Object`, али се сматра да ће се заиста уносити објекти класе `ЧланКомисије`. Метода враћа `TRUE` ако је вредност атрибута `imePrezime` једнака имену и презимену члана комисије који је унет као параметар, а у супротном `FALSE`.

Направити јавну класу `Комисија` која има:

- Приватни атрибут чланови који представља листу објеката класе `ЧланКомисије`.
- Јавни конструктор који иницијализује атрибут чланови.
- Јавну методу која као параметар прима објекат класе `ЧланКомисије` и уноси га у листу само ако у листи не постоји члан комисије са истим именом и презименом. У супротном, потребно је исписати поруку о томе да је та особа већ члан комисије.
- Јавну методу `спроведиПропис` која проверава да ли је у комисији пет чланова или мање. Ако у комисији има више од 5 чланова, метода брише листу и на екрану исписује поруку о томе да је комисија незажећа.
- Јавну методу `разрешиЧланаКомисије` која као параметар добија објекат класе `ЧланКомисије` и избацује га из комисије.
- Јавну методу која на екрану исписује састав комисије.

Направити јавну класу `ТестКомисије` која креира један објекат класе `Комисија` и у њега уноси податке о два члана комисије: "Пера Перић" - председник и "Мика Микић". Исписати састав комисије.

```
public class ClanKomisije {  
  
    private String imePrezime;  
    private String funkcija = null;  
  
    public String getImePrezime() {  
        return imePrezime;  
    }  
  
    public void setImePrezime(String imePrezime) {  
        if (imePrezime!=null && !imePrezime.equals(""))  
            this.imePrezime = imePrezime;  
        else  
            System.out.println("Morate uneti ime");  
    }  
  
    public String getFunkcija() {  
        return funkcija;  
    }  
  
    public void setFunkcija(String funkcija) {  
        this.funkcija = funkcija;  
    }  
  
    public String toString(){  
        if (funkcija!=null)  
            return "Ime clana: "+imePrezime+" Funkcija: "+funkcija;  
    }  
}
```

```

        else return "Ime clana: "+imePrezime;
    }

public boolean equals(Object o){
    ClanKomisije ck = (ClanKomisije)(o);
    if (imePrezime.equals(ck.getImePrezime())) return true;
    else return false;
}

import java.util.LinkedList;

public class Komisija {

    private LinkedList <ClanKomisije> clanovi;

    public Komisija(){
        clanovi = new LinkedList<ClanKomisije>();
    }

    public void unesi(ClanKomisije ck){
        if (clanovi.contains(ck))
            System.out.println("Ta osoba se vec nalazi u komisiji");
        else clanovi.add(ck);
    }

    public void sprovediPropis(){
        if (clanovi.size()>5){
            clanovi.clear();
            System.out.println("Komisija je nevazeca");
        }
    }

    public void razresiClanaKomisije(ClanKomisije ck){
        clanovi.remove(ck);
    }

    public void ispisi(){
        for (int i=0;i<clanovi.size();i++)
            System.out.println(clanovi.get(i));
    }
}

public class TestKomisija {

    public static void main(String[] args) {
        Komisija k = new Komisija();

        ClanKomisije ck1 = new ClanKomisije();
        ck1.setImePrezime("Pera Peric");
        ck1.setFunkcija("predsednik");

        ClanKomisije ck2 = new ClanKomisije();
        ck2.setImePrezime("Mika Mikic");
    }
}

```

```
k.unesi(ck1);  
k.unesi(ck2);  
  
    k.ispisi();  
}  
}
```

Класа Комисија има приватни атрибут “чланови” који представља листу објеката класе ЧланКомисије. Као што се може видети, иницијализација ове листе се врши у оквиру конструктора.

Метода “унеси” уноси члана комисије у листу позивањем методе “add”. Пре уношења, проверава се да ли листа можда већ садржи члана са истим именом и презименом. Метода “contains” врши ову проверу тако пролази кроз листу елемент по елемент и аутоматски позива методу “equals” класе ЧланКомисије да провери једнакост сваког елемената са унетим објектом.

Метода “спроведиПропис” прво проверава број члanova комисије (позивањем методе “size”) па, ако је тај број већи од 5, брише све елементе из листе (позивањем методе “clear”).

Избацување елемента из листе се врши позивањем неке од “remove” метода. У оквиру методе “разрешиЧланакомисије” се позива она “remove” метода која проналази елемент из листе који је једнак унетом објекту. И овде се једнакост проверава аутоматским позивањем “equals” методе класе ЧланКомисије.

У методи “испиши” се види како се пролази кроз листу елемент по елемент. Појединачним елементима листе се приступа преко индекса - позивањем методе “гет” којој се прослеђује тај индекс. Као и код низова и код листа индекси крећу од нуле.

10 Изузетаки

Већ је наведено да се Јава програм не може компајлирати све док у њему постоје синтаксне грешке. У том случају, програм се не може ни покренути. Међутим, шта се дешава када се деси грешка у току извршавања програма? Механизам “изузетака” (“exception”) у Јави постоји управо да би се овакве ситуације обрађивале и решавале на униформан начин.

Изузетаки (у Јави) су све оне ситуације које наступе у току извршења програма а, при томе, могу да поремете нормалан рад програма. Грешке које изазивају изузетке су углавном у вези са: уносом података (“User input errors”), радом периферних уређаја рачунара (“Device errors”), радом меморијских уређаја рачунара (“Physical limitations”) или самим извршавањем метода (“Code errors”). **Изузетаки нису синтаксне грешке**, па програм који изазива овакве ситуације може нормално да се компајлира и покрене.

Механизам за обрађивање изузетака у Јави функционише на следећи начин. Метода у којој се десила грешка **“баци” (“throws”)** одговарајући изузетак. Поред података о **врсти грешке**, изузетак садржи и **податке о томе у којој методи и на којој линији кода је настала грешка** (тзв. “stack-trace”). Када нека метода баци изузетак, **извршавање те методе али и целог програма се прекида**. То се може видети на једном једноставном примеру.

Пример 67

Направити јавну класу *ИзузетциДемо* која има:

- Атрибут низ који представља низ целих бројева.
- Методу која иницијализује низ на 10 елемената и на екрану исписује 11. елемент.

Написати класу *ТестИзузетциДемо* која креира један објекат класе *ИзузетциДемо* и позива његову методу за исписивање. После тога, потребно је исписати поруку да је 11. елемент низа исписан. Покренути програм.

```
public class IzuzeciDemo {
    int[] niz;

    void ispisiJedanaestog() {
        niz = new int[10];
        System.out.println(niz[10]);
    }
}

public class TestIzuzeciDemo {
    public static void main(String[] args) {
        IzuzeciDemo id = new IzuzeciDemo();

        id.ispisiJedanaestog();

        System.out.println("Jedanaesti element je ispisan.");
    }
}
```

Када се наведени код покрене, изазваће појављивање изузетка. Конкретно, метода "испишиЈеданаестог" ће да иницијализује низ на 10 елемената и да покуша да испише једанаести (што није могуће). Програм ће да се прекине, а на екрану ће да се појави следећа порука:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at IzuzeciDemo.ispisijedanaestog(IzuzeciDemo.java:7)
    at TestIzuzeciDemo.main(TestIzuzeciDemo.java:6)
```

Из саме поруке се може закључити неколико ствари. Прва линија поруке садржи податке о томе која је врста грешке у питању - вредност индекса којим се приступа елементима низа је ван граница ("java.lang.ArrayIndexOutOfBoundsException"). У продолжетку исте линије је наведена и порука која додатно објашњава узрок настајања грешке - "10". То је управо она недозвољена вредност индекса која је изазвала појављивање грешке (низ је иницијализован на 10 елемената, па је индекс последњег елемента 9). Остале линије поруке садрже информацију о томе на којој линији кода је настала грешка. Може се видети да је грешка настала на седмој линији кода класе ИзузециДемо (файл је "IzuzeciDemo.java") и то у оквиру методе "испишиЈеданаестог" ове класе. Такође, може се видети да се у оквиру "main" методе класе ТестИзузециДемо, на шестој линији кода, позива метода која је направила грешку.

На крају, потребно је приметити да се програм прекида чим се појави грешка тј. баца изузетак. Пример за то је команда за исписивање поруке која се налази у "main" методи класе ТестИзузециДемо. Ова команда није уопште извршена (порука није исписана на екрану) јер је бачен изузетак пре него што је могла да се изврши.

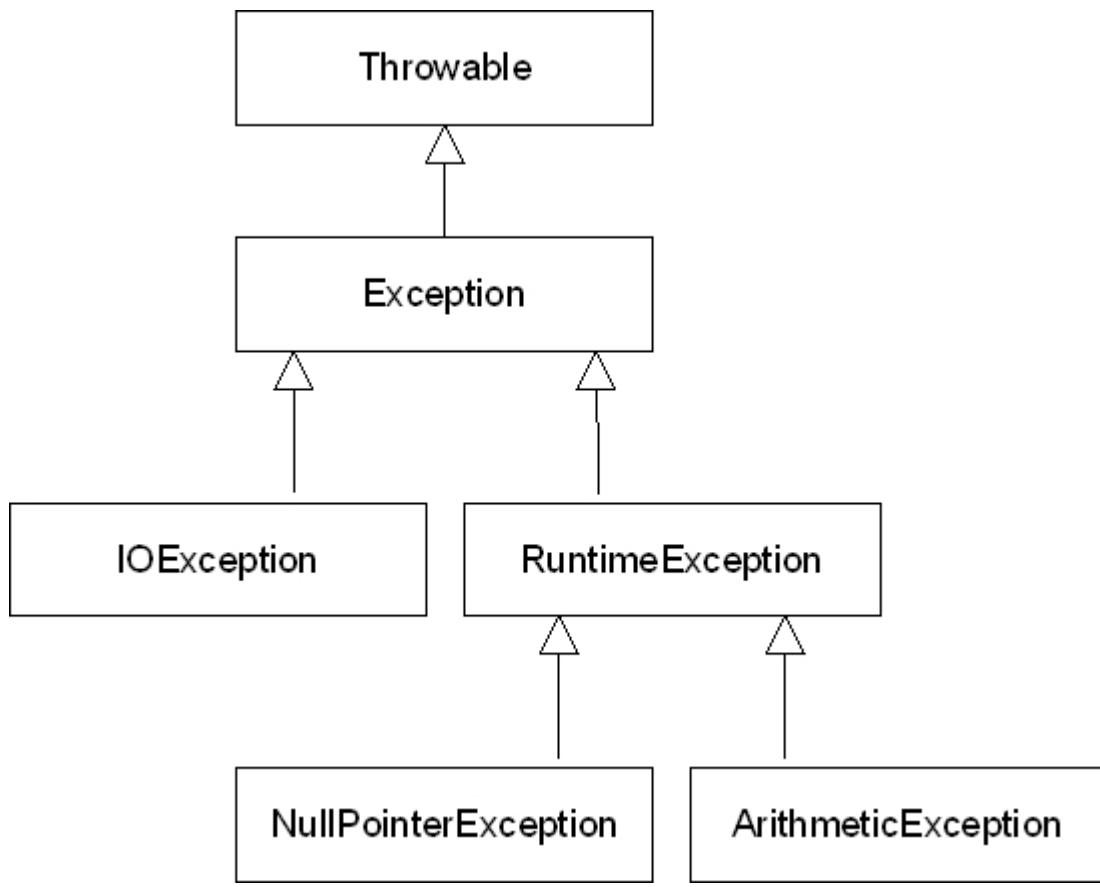
Изузеци у Јави су **имплементирани у виду класа**. Практично, то значи да се, сваки пут када се деси грешка, направи по један објекат класе која представља изузетак, напуни одговарајућим подацима и проследи Јавином механизму обрађивање изузетака. У зависности од тога који је тип грешке у питању, баца се објекат одговарајуће класе изузетка.

Јава садржи неке **предефинисане класе изузетака** које баца аутоматски. Све ове класе припадају пакету "java.lang". Кратка листа ових класа са одговарајућим описима је дата у следећој табели.

| Тип изузетка (класа) | Ситуација у којој се појављује |
|---------------------------------|--|
| ArithmaticException | Било каква грешка при аритметичким операцијама (нпр. дељење са нулом). |
| ArrayIndexOutOfBoundsException | Вредност индекса елемента низа је мања од нуле или већа од максималне дозвољене вредности (нпр. приступа се 11. елементу низа чији је капацитет 10 елемената). |
| StringIndexOutOfBoundsException | Вредност индекса којим се приступа знаку из String-а је мања од нуле или већа од максималне дозвољене вредности (дужине String-а). |
| NegativeArraySizeException | Иницијализација низа са негативним капацитетом. |
| ClassCastException | Неисправна конверзија објекта у објекат друге класе. |
| NullPointerException | Приступање низу или објекту неке класе пре него што је извршена иницијализација (низ тј. објекат има вредност "null"). |

| | |
|-----------------------|--|
| NumberFormatException | Неисправна конверзија String вредности у неки реалан или цео број. |
|-----------------------|--|

Класе изузетака формирају одређену **хијерархију наслеђивања** (Слика 23). Основна класа из које су изведени сви изузетци је класа Throwable. Ова класа садржи само неке основне функционалности и омогућава повезивање са механизmom за обрађивање изузетака. Њена директна подкласа је класа Exception. Класа Exception садржи све потребне функционалности које би један изузетак требало да има, па се може условно рећи да је заправо ова класа надкласа свих изузетака. Њене директне подкласе су, на пример, IOException (која представља грешке у раду са уређајима) и RuntimeException (која представља грешке у току извршавања кода које нису у вези са уређајима). Из класе RuntimeException су изведене предефинисане класе изузетака NullPointerException и ArithmeticException, али и многе друге.



Слика 23: Хијерархија класа изузетака

Јавини предефинисани изузетци се бацају аутоматски сваки пут када се појави одговарајући тип грешке. Међутим, некада је потребно бацити изузетак у оним ситуацијама које се сматрају грешкама (за одређени програм). На пример, ако се као вредност параметра неке методе унесе нека недозвољена вредност, метода би требало да баши изузетак. Изузетак се **може бацити “по потреби” уз помоћ наредбе “throws”**. Ова наредба се користи тако што се иза резервисане речи “throws” наведе (или креира) објект класе изузетка који се жели бацити. Конструкторују класе изузетка се може (али не мора) проследити порука грешке у виду String вредности.

| |
|---|
| <code>throw new NazivKlaseIzuzetka(poruka_greške);</code> |
|---|

Пример 68

Направити јавну класу Особа која има:

- Приватни атрибут име.
- Приватни атрибут презиме.
- Одговарајуће јавне get и set методе за ова два атрибута. Недозвољене вредности за оба атрибута су null String-ови. У случају уноса недозвољених вредности, потребно је бацити изузетак са одговарајућом поруком.

```
public class Osoba {  
  
    private String ime;  
    private String prezime;  
  
    public String getIme() {  
        return ime;  
    }  
    public void setIme(String ime) {  
        if (ime == null)  
            throw new RuntimeException("Ime ne moze biti null");  
        this.ime = ime;  
    }  
  
    public String getPrezime() {  
        return prezime;  
    }  
  
    public void setPrezime(String prezime) {  
        if (prezime == null)  
            throw new RuntimeException(  
                "Prezime ne moze biti null");  
        this.prezime = prezime;  
    }  
}
```

У примеру се може видети да се у случају уноса недозвољених вредности баца изузетак класе `RuntimeException`. Конструктору изузетка је, у оба случаја, прослеђена порука коју је потребно проследити у случају грешке.

Потребно је приметити да у обе set методе IF наредба нема ELSE део у којем се налази наредба за додељивање вредности атрибуту. Разлог је тај што ће се, ако се деси изузетак, извршавање методе или и целог програма прекинути па се наредба за доделу свеједно неће извршити.

Ако у току извршавања програма буде бачен изузетак, извесно је да ће се програм истог тренутка прекинути. Али то не мора да буде тако. Програм ће се прекинути само ако бачени изузетак не буде ухваћен (“catch”). Команда којом се могу хватати и контролисати бачени изузетци је тзв. **“try-catch” блок**. Овај блок се састоји од резервисаних речи “try” и “catch” између којих је потребно поставити све оне команде и позиве методама за које се сумња да могу бацити изузетак. Функција “try-catch” блока је да ухвати изузетак, омогући његову обраду и спречи прекидање програма. Такође, овај блок се активира само ако изузетак буде бачен, а у супротном не утиче на ток извршавања програма. Декларација овог блока се врши на следећи начин.

```
try{
    ...komande koje mogu baciti izuzetke...

} catch(NazivKlaseIzuzetka izuzetak) {
    ...komande koje se izvršavaju samo ako je izuzetak bačen...

}
```

Блок почиње резервисаном речи “try” иза које следе витичасте заграде. У оквиру ових заграда је потребно навести команде за које се зна да могу бацити изузетке. Следећа резервисана реч је “catch” иза које, у обичним заградама стоји назив класе изузетка која се хвата (нпр. IOException или ArithmeticException) и назив променљиве која ће да прими референцу на бачени изузетак. Коначно, на крају је блок наредби које се извршавају само ако изузетак буде бачен, а у супротном не. У оквиру овог, последњег блока наредби се може позивати објекат који представља бачени изузетак.

Потребно је напоменути да ће **“catch” наредба да ухвати само изузетке оне класе (и било које од њених подкласа) чији назив је наведен у загради**. Значи, ако се хвата изузетак класе IOException, а буде бачен изузетак класе ArithmeticException, “try-catch” блок неће да га ухвати и програм ће се прекинути. Са друге стране, ако се као очекивани тип изузетка постави класа Exception, ухватиће се сви изузетци јер је класа Exception надкласа свих изузетака.

Пример 69

Следи пример употребе “try-catch” блока. У овом случају, баца се изузетак класе RuntimeException, а “catch” наредба хвата управо изузетак овог типа. Када се код изврши, програм неће да се прекине, а на екрану ће се само исписати изузетак - његов тип и порука.

```
try{
    int a = 1;
    if (a==1)
        throw new RuntimeException("A je jedan");
} catch(RuntimeException e){
    System.out.println(e);
}
```

ПРЕТХОДНИ пример може да буде написан и тако да “catch” наредба хвата изузетак класе Exception. У том случају, ефекат извршавања је исти јер је Exception надкласа класе RuntimeException па ће сваки изузетак класе Exception или било које њене подкласе бити ухваћен.

```
try{
    int a = 1;
    if (a==1)
        throw new RuntimeException("A je jedan");
} catch(Exception e){
    System.out.println(e);
}
```

Следећи “try-catch” блок, међутим, неће моћи да ухвати изузетак класе ArithmeticException ако буде бачен. Разлог је тај што се очекује изузетак класе NullPointerException.

```
try{
    int a = 1;
    if (a==1)
```

```
    throw new ArithmeticException("A je jedan");
} catch(NullPointerException e) {
    System.out.println(e);
}
```

Ако неки скуп команди може да изазове бацање више различитих типова изузетака, могуће је направити **“try-catch” блок са неколико “catch” наредби** и то тако да свака од наредби хвата по један тип изузетка. Ако изузетак буде бачен, провераваће се очекивани типови изузетака сваке “catch” наредбе и то редом у којем су написане све док се не нађе тип који одговара баченом изузетку. Так тада ће се активирати одговарајућа “catch” наредба (и то само једна). Уколико је потребно да се обавезно изврше неке наредбе након што се десио изузетак, може се употребити и **“finally” блок**. Команде из овог блока ће се извршити после команди одговарајуће “catch” наредбе.

```
try{
    ...komande koje mogu baciti izuzetke...

} catch(NazivKlaseIzuzetka1 izuzetak) {
    ...komande koje se izvršavaju samo ako je izuzetak 1 bačen...

} catch(NazivKlaseIzuzetka2 izuzetak) {
    ...komande koje se izvršavaju samo ako je izuzetak 2 bačen...

} finally{
    ...komande koje se izvršavaju ako je bilo koji izuzetak bačen...
}
```

Потребно је **обратити пажњу и на редослед писања “catch” наредби** у оквиру “try-catch” блока који има неколико “catch” наредби. Подкласе изузетака морају бити наведене као очекивани типови изузетака пре својих надкласа, иначе ће се активирати “catch” наредба надкласе изузетка уместо “catch” наредбе подкласе изузетка.

Пример 70

Следи пример употребе “try-catch” блока са неколико “catch” наредби. У овом случају, баца се изузетак класе *RuntimeException* или *ArithmetricException*, а “catch” наредбе хватају управо изузетке овог типа.

```
try{
    int a = 1;
    if (a==1)
        throw new ArithmetricException("A je jedan");
    else
        throw new RuntimeException ("A nije jedan");
} catch(ArithmetricException e){
    System.out.println("A je jedan "+e);
} catch(RuntimeException e){
    System.out.println("A nije jedan "+e);
}
```

Ако је потребно извршити исте наредбе у случају да се баци било који од изузетака, претходни пример је могао да се напише и на следећи начин. У овом случају “catch” наредба може да ухвати било који тип изузетка.

```
try{
    int a = 1;
    if (a==1)
        throw new ArithmeticException("A je jedan");
    else
        throw new RuntimeException ("A nije jedan");
}catch(Exception e){
    System.out.println(e);
}
```

Конечно, следи пример “try-catch” блока са “finally” блоком. “Finally” блок ће да се активира сваки пут када неки изузетак буде бачен, без обзира на то који је тип изузетка у питању.

```
try{
    int a = 1;
    if (a==1)
        throw new ArithmeticException("A je jedan");
    else
        throw new RuntimeException ("A nije jedan");
}catch(ArithmeticException e){
    System.out.println("A je jedan "+e);
}catch(RuntimeException e){
    System.out.println("A nije jedan "+e);
}finally{
    System.out.println("Greska postoji!");
}
```

Потребно је размотрити и шта се тачно може урадити у случају да изузетак буде ухваћен. У неким ситуацијама, доволно је да се спречи прекидање програма, па блок наредби после резервисане речи “catch” може да остане празан. Али, некада је потребно исписати податке које садржи изузетак, па се морају позивати одговарајуће методе класе изузетка.

Када су класе изузетака у питању, метода **“toString”** враћа String са називом класе изузетка и поруком изузетка. Метода **“getMassage”** враћа String који само садржи поруку изузетка, док метода **“printStackTrace”** исписује “stack-trace” на екрану.

Пример 71

Следи пример позива метода “*toString*”, “*getMassage*” и “*printStackTrace*” баченог изузетка.

```
public class PrimerPozivaMetodaIzuzetka {

    public static void main(String[] args) {
        try{
            throw new RuntimeException("Greska");
        }catch(RuntimeException e){
            //Ispisace se naziv klase izuzetka i poruka
            //java.lang.RuntimeException: Greska
            System.out.println(e);

            //Ispisace se samo poruka izuzetka
            //Greska
            System.out.println(e.getMessage());

            //Ispisace se ceo "stack trace"
        }
    }
}
```

```
//java.lang.RuntimeException: Greska
//at PrimerPozivaMetodaIzuzetka.main
//(PrimerPozivaMetodaIzuzetka.java:6)
e.printStackTrace();
}
}
```

Изузеци у Јави се деле у две групе: **проверавани изузети (“checked”)** и **непроверавани изузети (“unchecked”)**. Класа RuntimeException и све њене подкласе спадају у групу непровераваних изузетака, док су све остале класе из групе провераваних изузетака (нпр. Exception, IOException...). Класе које су наведене у табели Јавиних предефинисаних изузетака наслеђују класу RuntimeException па припадају групи непровераваних изузетака. Али, у чему је разлика?

Ако метода баца неки непроверавани изузетак, позив те **методе може, али не мора бити уоквирен “try-catch” блоком који хвата тај изузетак**. Ако метода баца неки проверавани изузетак, **позив те методе мора да буде уоквирен “try-catch” блоком који хвата тај изузетак, а метода мора да буде означена резервисаном речи “throws” и називом класе изузетка који баца**.

```
tip_vr nazivMetode(...parametri...) throws NazivKlaseIzuzetka{
    //Telo metode
}
```

Пример 72

Написати класу ПровераПарности која има:

- Јавну статичку методу провераПарности која враћа TRUE ако је број који је унет као параметар паран, а FALSE ако није. Метода баца ПРОВЕРАВАНИ изузетак ако је унети број једнак 0.

Написати класу ТестПровераПарности која позива методу класе ПровераПарности.

```
public class ProveraParnosti {

    public static boolean proveraParnosti(int a) throws Exception{
        if (a==0)
            throw new Exception("Uneli ste nulu");
        if (a%2==0) return true;
        else return false;
    }
}

public class TestProveraParnosti {

    public static void main(String[] args) {
        //Svaki poziv ove metode MORA da bude uokviren
        //u "try-catch" blok jer metoda baca proveravani
        //izuzetak.
        try {
            ProveraParnosti.proveraParnosti(10);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
}
```

На крају, потребно је додати и то да се **у Јави могу правити и нове класе изузетака** и то наслеђивањем одговарајућих класа (на пример Exception за провераване и RuntimeException за непровераване изузетке). На овај начин се могу пружати још детаљније поруке о томе где је и зашто настала нека грешка.

11 Литература

- [1] B. Eckel, “Thinking in Java”, треће издање, Prentice Hall, 2002
- [2] М. Чабаркапа, “C++ основе програмирања”, CET, 2007
- [3] М. Чабаркапа, Н. Спалевић, “Методичка збирка задатака из програмирања са решењима у PASCAL-y”, CET, 2007
- [4] M. Fowler, K. Scott, “UML Distilled”, треће издање, Addison Wesley, 2004
- [5] С. Влајић, “Пројектовање програма - Скрипта”, Факултет организационих наука, Београд, 2004

II ДЕО – НАПРЕДНЕ ЈАВА ТЕХНОЛОГИЈЕ

Синиша Влајић, Душан Савић, Илија Антовић,
Војислав Станојевић, Милош Милић

САДРЖАЈ

| | |
|---|-----------|
| 1.НИТИ | 2 |
| 2.1 Главна програмска нит | 2 |
| 2.2 Прављење нити | 3 |
| 2.2.1 Прављење нити реализацијом класе Runnable | 3 |
| 2.2.2 Прављење нити проширењем класе Thread | 7 |
| 2.3 Прављење више нити..... | 7 |
| 2.4 Стана нити | 8 |
| 2.5 Прекид нити | 11 |
| 2.6 Коришћење метода isAlive() и join() | 17 |
| 2.7 Приоритет извршавања нити | 20 |
| 2.8 Себичне (<i>selfish</i>) нити | 21 |
| 2.9 Групе нити..... | 21 |
| 2.10 Синхронизација..... | 24 |
| 2.10.1 Комуникација нити без синхронизације | 25 |
| 2.10.2 Закључавање објекта | 26 |
| 2.10.3 Коришћење синхронизованих метода | 26 |
| 2.10.4 Коришћење синхронизованих објекта..... | 27 |
| 2.11 Комуницирање између нити | 28 |
| 2.12 Међусобно блокирање нити..... | 36 |
| 2.13 Коришћење цеви (pipes) за комуникацију између нити | 38 |
| 3. ПРОГРАМИРАЊЕ (РАД) У МРЕЖИ | 40 |
| 3.1 Адреса рачунара..... | 40 |
| 3.2 URL адреса | 42 |
| 3.3 Сокети | 43 |
| 3.3.1 Адреса сокета | 43 |
| 3.3.2 Конекција..... | 43 |
| 3.3.3 Повезивање сервера са више клијената..... | 47 |
| 3.4 Слање електронске поште..... | 59 |
| 3. РАД СА БАЗОМ – JDBC | 62 |
| 3.1 Поступак повезивања Јава програма и СУБП-а | 62 |
| 3.2 Поступак извршења операција над базом података СУБП | 65 |
| 3.3 Примери SQL упита над MySQL СУБП | 72 |
| 3.3.1 Дефиниција података (data definition) | 72 |
| 3.3.2 Манипулација подацима (data manipulation) | 74 |
| 4.РЕФЕРЕНЦЕ | 84 |

1.НИТИ

Едсгер Дијкстра је рекао да се “конкурентност дешава када постоје два или више извршних токова (процеса) који су способни да се извршавају симултрано (истовремено)”. Процеси могу истовремено да користе дељене ресурсе (*shared resources*) што може да резултује непредвиђеним понашањем система. Увођење **међусобног исхључења** (*mutual exclusion*) може да спречи непредвиђено понашање код коришћења дељених ресурса али може да доведе до појаве **мртвог закључавања** (*deadlock*) и **гладовања** (*starvation*).

Deadlock је мултитаскинг проблем¹ који се дешава када два процеса заузму ресурсе и медусобно се цекају да ослободе те ресурсе. **Starvation** је мултитаскинг проблем који се дешава када неки процес заузме неки ресурс и не дозвољава другим процесима да га користе. То доводи до тога да други процеси не могу да се до краја изврше.

Уколико више процеса међусобно сарађују у извршењу неког задатка јавља се проблем њихове међусобне комуникације и размене података јер сваки процес заузима посебан меморијски простор. Тај проблем је решен појавом **нити** (*threads*) које деле исти меморијски простор.

Јава је један од програмских језика који подржава вишенитно програмирање. То значи да један програм (процес) може да обавља више нити истовремено (конкурентно). **Нит представља део програма који може истовремено да се извршава са другим нитима истог програма.**

Нити, као што је речено, деле исти адресни простор у оквиру једног процеса и комуникација између њих је доста једноставнија у односу на комуникацију између процеса. Радом са више процеса управља оперативни систем, док радом са више нити управља Јава окружење.

2.1 Главна програмска нит

Када програм у Јави почне да се извршава, он аутоматски креира и извршава једну нит која се зове главна програмска нит.

```
/*
 * Primer NT1: Napisati program koji ce da ukaze na glavnu nit koju treba
 * uspavati 5 sekundi.
 */

/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */

class NT1 {
```

¹ *Multitasking* је метода помоћу које више задатака (*tasks*), односно процеса, деле заједничке ресурсе као што је *CPU* (*Central Processing Unit*). У случају компјутера са једним *CPU*, у неком тренутку времена само један задатак може да се извршава, што практично значи да *CPU* активно извршава само инструкције тог задатка. *Multitasking* омогућава да се у неком периоду времена више процеса извршава, тако што прави распоред (*scheduling*) који задатак ће се извршавати у ком тренутку времена а који задаци ће чекати док на њих не дође ред. Додељивање *CPU* од једног до другог задатка се назива **контекстна додела** (*context switch*). Када се *context switch* дешава учестало ствара се илузија истовремености извршења више задатака. *Multitasking* омогућава да се изврши више задатака на једном *CPU* у односу на компјутере са више *CPU* (мултипроцесорски компјутери) који не користе *multitasking*.

```
public static void main(String args[]) {
    Thread gnit = Thread.currentThread();
    System.out.println("Glavna nit:" + gnit + '\n' + "Pauza od 5 sekundi");
    try {
        gnit.sleep(5000); // 5 sekundi pauze
    } catch (InterruptedException e) {
        System.out.println("Prekid niti");
    }
    gnit.setName("Glavna"); // Promena naziva niti
    System.out.println("Glavna nit:" + gnit + '\n' + "Naziv glavne niti:" + gnit.getName());
}
/*
Rezultat:
Glavna nit:Thread[main,5,main]
Pauza od 5 sekundi.
Glavna nit:Thread[Glavna,5,main]
Naziv glavne niti: Glavna
*/
```

2.2 Прављење нити

У Јави се нит може направити на 2 начина:

- Реализацијом интерфејса Runnable
- Проширењем класе Thread

2.2.1 Прављење нити реализацијом класе Runnable

Када се нит прави реализацијом интерфејса Runnable, тада је једино потребно се имплементира (реализује) метода run() интерфејса Runnable².

```
/* Primer NT2: Napraviti nit pomocu interfejsa Runnable.*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */

class NT2 implements Runnable {
    NT2() {
        //nova nit ce pozvati run metodu od this objekta
        nit = new Thread(this, "Nova nit");
        nit.start();
    }
    public void run() {
        System.out.println("Nit:" + nit);
    }

    public static void main(String args[]) {
        NT2 nn = new NT2();
        Thread gnit = Thread.currentThread();
        gnit.setName("Glavna nit");
        System.out.println("Nit:" + gnit);
    }
    Thread nit;
}
/*
Rezultat:
Nit:Thread[Glavna nit,5,main]
Nit:Thread[Nova nit,5,main]
*/
```

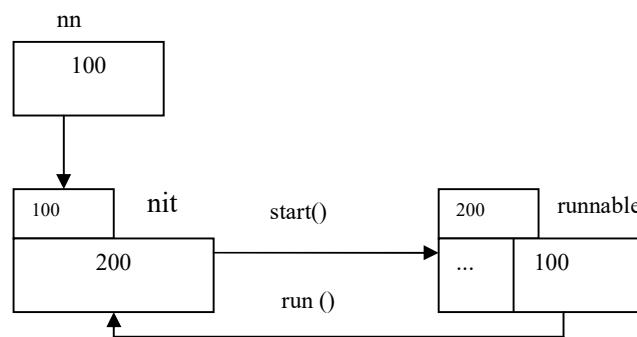
Упрошћено објашњење односа између класа Thread и NT2

² main() метода се за главну нит у суштини понаша исто као и run() метода за нити које су креиране у main() методи.

Однос између класа `NT2` и `Thread`, односно њихових објеката, показаћемо на примеру који наглашава најважније аспекте ове комуникације. Треба нагласити да је класа `Thread` која је дата у овом примеру представља поједностављену верзију оригиналне Јавине класе `Thread`.

```
/**  
 * @author Sinisa Vlajic  
 * SILAB - Labartorija za Softversko Inzenjerstvo  
 * FON - Beograd  
 *http://silab.fon.bg.ac.rs  
 */  
  
class NT2 implements Runnable {  
    NT2() {  
        nit = new Thread(this, "Nova nit");  
        nit.start();  
    }  
    public void run() {  
        ...  
    }  
    public static void main(String args[]) {  
        NT2 nn = new NT2();  
        ...  
    }  
    Thread nit;  
}  
  
public class Thread implements Runnable {  
    ...  
    final Runnable runnable;  
    ...  
    public Thread(Runnable target, String name) {  
        this.runnable = target;  
    }  
    public synchronized void start() {  
        ... // Kreiranje i inicializacija prirodne niti niti preko operativnog sistema  
        runnable.run();  
    }  
    ...  
}
```

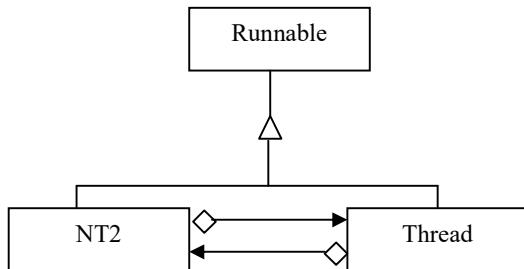
Изглед оперативне меморије наведеног примера:



Објекти на које показују `nn` и `nit` (који се налазе на адресама 100 и 200) показују један на други. На тај начин је могуће да објекат класе `NT2` (на адреси 100) позове методу `start()` од објекта класе `Thread` (на адреси 200), односно да објекат класе `Thread` позове методу `run()` од објекта класе `NT2`.

Метода `start()` класе `Thread` има задатак да креира³ и иницијализује природну нит и да након тога позове методу `run()` класе `NT2`.

Дијаграм класа наведеног примера:



Детаљније објашњење односа између класа `Thread` и `NT2`

Да би се детаљније схватио наведени пример⁴ навешћемо најважније атрибуте и методе класа `Thread` и `VMThread`⁵.

```
public class Thread implements Runnable {  
    ...  
    final Runnable runnable;  
    ...  
    public Thread(Runnable target, String name) {  
        this(null, target, name, 0);  
    }  
    ...  
    /* Ovaj se konstruktor koristi u primeri NT3 */  
    public Thread(String name){  
        this(null, null, name, 0);  
    }  
    public Thread(ThreadGroup group, Runnable target, String name, long size) {  
        ...  
        this.runnable = target;  
        ...  
    }  
    ...  
    public synchronized void start() {  
        ...  
        VMThread.create(this, stacksize);  
    }  
    ...  
    public void run() {  
        if (runnable != null) {  
            runnable.run();  
        }  
    }  
    ...  
}
```

³ Креирање природне нити (*native thread*) ради оперативни систем. То значи да ће преко методе `start()` бити позвана нека од метода (операција) оперативног система која је задужена за креирање природне нити. У даљем тексту ће то бити мало детаљније објашњено.

⁴ Наведене класе су такође потребне да би се схватио пример `NT3`, када се креира нит наслеђивањем класе `Thread`.

⁵ Изворни код наведених класа се може наћи на: <http://www.docjar.com/html/api/java/lang/Thread.java.html> и <http://www.docjar.com/html/api/java/lang/VMThread.java.html>

```

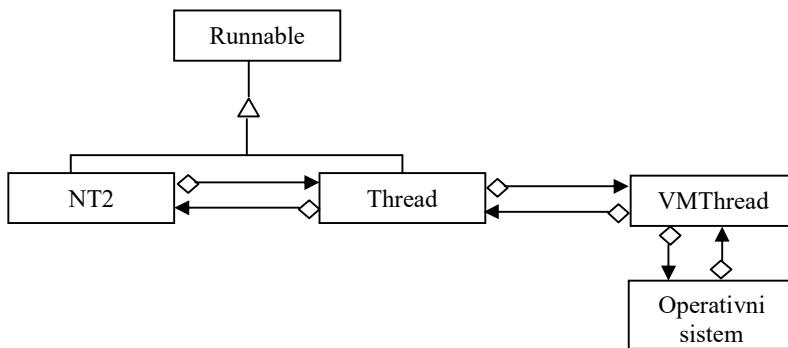
final class VMThread {

    volatile Thread thread;
    ...
    private VMThread(Thread thread) {
        this.thread = thread;
    }
    ...
    private void run() {
        ...
        thread.run();
    }

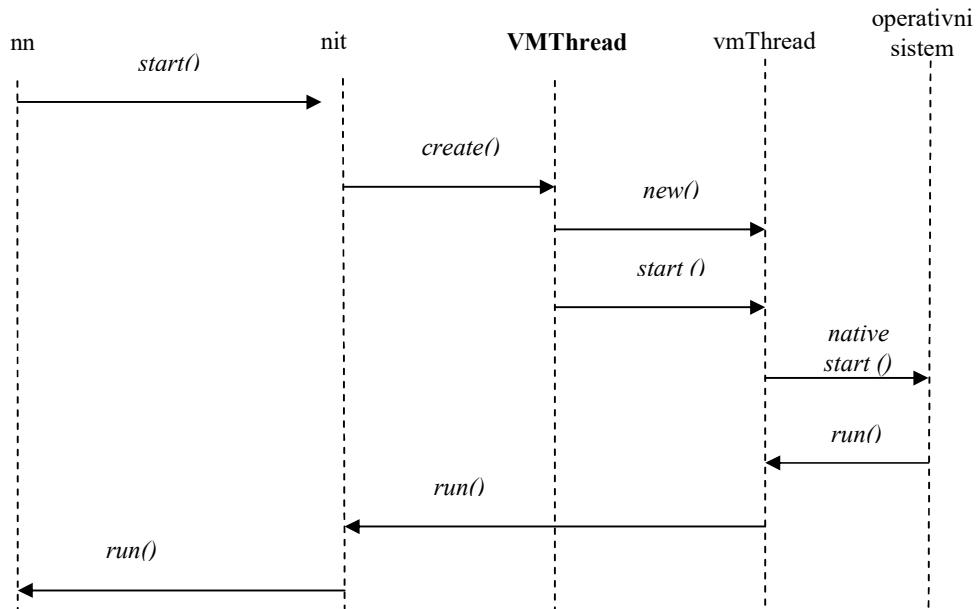
    ...
    static void create(Thread thread, long stacksize) {
        VMThread vmThread = new VMThread(thread);
        vmThread.start(stacksize);
        ...
    }
    ...
    /* Create a native thread on the underlying platform and start it executing on the run
method of this object.
     * @param stacksize the requested size of the native thread stack
     */
    native void start(long stacksize);
    ...
}

```

Детаљни дијаграм класа наведеног примера:



Секвенцни дијаграм односа између објеката наведеног примера:



На основу наведених дијаграма класа и секвенцног дијаграма може се закључити да позивом методе `start()` објекта нит класе `Thread` започиње процес позива метода (*create, new, start*) класе `VMTThread` који се завршава позивом методе `start()` оперативног система која креира природну нит. Након креирања природне нити оперативни систем позива методу `run()` класе `VMTThread` која позива методу `run()` класе `Thread` која на крају позива методу `run` класе `NT2`.

2.2.2 Прављење нити проширењем класе Thread

Када се нит прави проширењем класе `Thread`, тада је потребно се прекрије метода `run()` класе `Thread`. Класа `Thread` поред методе `run()` садржи и друге методе које могу да се прекрију, за разлику од интерфејса `Runnable` који има само методу `run()`.

```
/*Primer NT3: Napraviti nit pomocu klase Thread.*/

/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */

class NT3 extends Thread {
    NT3() {
        super("Nova nit");
        start();
    }
    public void run() {
        System.out.println("Nit:" + currentThread());
    }
    public static void main(String args[]) {
        NT3 nn1 = new NT3();
        Thread gnit = Thread.currentThread();
        gnit.setName("Glavna nit");
        System.out.println("Nit:" + gnit);
    }
}
/*
Rezultat:
Nit: Thread[Glavna nit,5,main]
Nit: Thread[Nova nit,5,main]
*/
```

Задаци:

1. Нацртати изглед детаљног дијаграма класа за пример `NT3`.
2. Нацртати и објаснити секвенцни дијаграм за методу `start()` која се позива у конструктору класе `NT3`.
3. У методи `start()` класе `Thread` код наредбе:
`VMThread.create(this,stacksize);`
шта је `this`?
4. У методи `run()` класе `VMTThread`:
`private void run() { ... thread.run(); }`
шта позива наредба `thread.run()`?

2.3 Прављење више нити

У једном Јава програму може да постоји више нити.

```
/*
 * Primer NT31: Napraviti u jednom programu vise niti.*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 * http://silab.fon.bg.ac.rs
 */

class NT31 extends Thread {
    NT31(String nn) {
        super(nn);
        start();
    }
    public void run() {
        System.out.println("Nit:" + currentThread());
    }
    public static void main(String args[]) {
        NT31 nn1 = new NT31("Nova nit1");
        NT31 nn2 = new NT31("Nova nit2");
        NT31 nn3 = new NT31("Nova nit3");
        Thread gnit = Thread.currentThread();
        gnit.setName("Glavna nit");
        System.out.println("Nit:" + gnit);
    }
}
/*
Rezultat:
Nit: Thread[Glavna nit,5,main]
Nit: Thread[Nova nit1,5,main]
Nit: Thread[Nova nit2,5,main]
Nit: Thread[Nova nit3,5,main]
*/

```

2.4 Стања нити

Нит може бити у једном од 4 стања:

- ново (*new*)
- извршно (*runnable*)
- блокирано (*blocked*)
- свршено (*dead*)

Нова нит

У почетку нит је декларисана: `Thread nit;`

Када се нит креира са `new` оператором: `nit = new Thread(this, "Nova nit");` нит прелази у стање **"ново"**. У том стању, нит се још не извршава. Након тога, позива се метода `start()`.

Извршавање нити

Након позива методе `start()`, нит прелази у стање **"извршно"**. Извршно стање не значи аутоматски да се нит **"извршава"** (*running*). Нит се извршава када контрола програма пређе на тело методе `run()`, коју позива метода `start()`.

```
// NapraviNit.java
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
```

```
* http://silab.fon.bg.ac.rs
*/
class NT3 implements Runnable{
    NT3() {
        nit = new Thread(this, "Nova nit"); // Nit je u stanju "novo"
        nit.start(); // Nit je u stanju "izvrsno"
    }

    public void run() {
        System.out.println("Nit:" + nit);
    } // Nit je u stanju "izvrsava"

    // Nakon izvršenja tela metode run() nit prelazi u stanje "svrseno"
    public static void main(String args[]) {
        NT3 nn = new NT3();
        Thread gnit = Thread.currentThread();
        gnit.setName("Glavna nit");
        System.out.println("Nit:" + gnit);
    }
}

Thread nit;
}
/*
Rezultat:
Nit: Thread[Glavna nit,5,main]
Nit: Thread[Nova nit,5,main]
*/
```

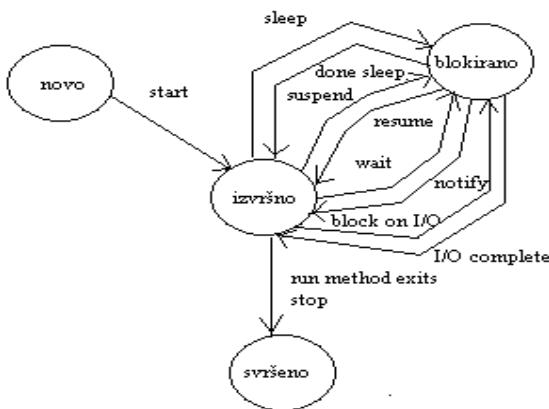
Оперативни систем је задужен да обезбеди процесорско време за нити, како би се оне извршиле. Старије верзије оперативних система (нпр. *Solaris* оперативни систем) су неку нит (почетна нит) извршавале, све до појаве неке нити вишег приоритета. Почетна нит би тада прешла у блокирано стање, док би нит вишег приоритета наставила да се извршава док се не изврши или док се не појави нека нит још већег приоритета.

Код новијих оперативних система (нпр. *Windows 2000* оперативни систем), свака нит добија део процесорског времена (*time – slicing*) да изврши свој задатак. Овај приступ је бољи, јер је у складу са вишенитном извршењем програма код Јаве . На машинама са више процесора, сваки процесор може да извршава једну нит. То значи да више нити може да се извршава истовремено.

Улазак нити у блокирано стање

- Нит прелази у блокирано стање, када се деси једна од следећих акција:
- Нит позива `sleep()` методу. Нит остаје "успавана" задати број милисекунда.
- Нит позива операцију, која је блокирала неки од улазно/излазних уређаја. Таква операција се неће наставити, све док се не ослободи заузети улазно/излазни уређај.
- Низ позива `wait()` методу.
- Нит покушава да закључча објекат који је већ закључан од стране друге нити.
- Нит позива `suspend()` методу. Ова метода је застарела и више се не користи.

На следећој слици се виде стања нити и могуће транзиције:



Излазак нити из блокираног стања

Нит може да пређе из "блокираног" у "извршно" стање, када се деси једна од следећих акција:

- Нит се "пробудила" након што је прошло време "успаваности ", задато методом `sleep()`.
- Операција, која је позвана у току извршења нити, која је блокирала улазно/излазни уређај је завршена.
- Ако је прва нит позвала `wait()` методу, друга нит мора позвати `notify()` или `notifyAll()` методу како би се деблокирала прва нит.
- Када блокирана нит, која чека на закључан објекат, добије контролу над њим, након што је нека нит престала да држи тај објекат закључан.
- Ако је нит била блокирана методом `suspend()`, она се може деблокирати једино позивом методе `resume()`. Метода `resume()` је такође застарела као и метода `suspend()`

Важна напомена: Блокирана нит се може деблокирати једино инверзном операцијом од операције која ју је блокирала (`sleep - done sleeping, suspend - resume, wait - notify, block on I/O - I/O complete`)

Ако се покуша деблокирати нит са неодговарајућом инверзном операцијом јавиће се `IllegalThreadStateException` изузетак.

Уништење нити

Нит прелази у стање "свршено" у следеће 2 ситуације:

- `run()` метода је престала да се извршава природним путем.
- `run()` метода је престала да се извршава јер се у току њеног извршења десио неухваћен изузетак.

Нит се такође може зауставити ако се позове метода `stop()`. Међутим ова метода је застарела и не користи се јер доводи до нестабилног понашања програма.

Уколико се жели испитати да ли је нит тренутно активна (независно од тога да ли се извршава или је блокирана) користи се метода `isAlive()`, која у том случају враћа `true`. Уколико је нит у стању "ново" или "свршено" метода `isAlive()` враћа `false`. Наведена метода ће касније бити детаљније објашњена.

2.5 Прекид нити

Нит се завршава када се тело `run()` методе изврши. Пошто је `stop()` метода превазиђена, користи се логичка контрола за наставак извршења нити:

```
public void run{
    while(signal) {
        izvršenje niti...
    }
}
```

Докле год `signal` има вредност `true`, нит се извршава. Када `signal` добије вредност `false`, позивом нпр. методе `promeni()` нит престаје да се извршава.

```
public void promeni(){
    signal = false;
}
```

Међутим у случају када је нит у блокираном стању, прекид извршења нити не може остварити на предходни начин. Тада се користи `interrupt()` метода која позива нит која је блокирана. Она ће прекинути блокаду нити генерирањем изузетка `InterruptedException`. Блокада може престати ако је иста настала на основу `sleep()` или `wait()` методе.

Наведена `interrupt()` метода прекида блокаду нити, али не прекида извршење нити. Уколико се жели да прекид нити преко `interrupt()` методе прекине извршење нити, тада се ухваћени изузетак `InterruptedException` обрађује на одговарајући начин.

```
public void run (){
    try {...}
        while(signal){
            izvršenje niti...
        }
    } catch(InterruptedException e) {
        // Prekinuta blokada niti koja je nastala na osnovu sleep() ili wait() metode.
    }
    ...
    // izlaz iz run() metode i prekid izvršenja niti.
}
```

Метода `interrupt()` статус нити поставља на `true`.

У случају када `interrupt()` метода позове нит која није блокирана, тада се неће десити изузетак `InterruptedException`. Због тога се код логичке контроле извршења нити позива метода `interrupted()`, која проверава да ли је текућа нит “претрпела” `interrupt()` методу (да ли је статус нити `true`).

Уколико је статус нити `true` треба да се прекине извршење нити:

```
while( !interrupted() && signal){
    izvršenje niti...
}
```

ИЛИ

Метода `interrupted()` има спољни ефекат (*side effect*), јер стање нити поставља на `false`.

Поред наведених постоји и метода `isInterrupted()` која враћа стање прекида нити али нема спољни ефекат:

```
while( !isInterrupted() && signal){
    izvršenje niti...
}
```

На основу наведеног може се закључити да постоје неколико сценарија покушаја прекида нити у зависности од тога: а) да ли је нит блокирана или не и б) како се покушава прекинута нит (`interrupt` методом или логичком контролом).

Сценарио прекида нити логичком контролом – нит није блокирана

Наведени сценарио се састоји из следећих корака:

- нит је покренута и њен статус је постављен на `true`.
- позива се метода промени која мења атрибут `signal` на `false`.
- метода `run` престаје да се извршава јер услов у `while` петљи није више задовољен:

```
while(signal){
    izvrsenje niti...
}
```

Сценарио прекида нити `interrupt` методом – нит није блокирана

- нит је покренута и њен статус је постављен на `true`.
- позива се метода `interrupt()`. Пошто нит није блокирана не генерише се изузетак `InterruptedException`. Статус нити се поставља на `true`.
- метода `run` престаје да се извршава јер услов у `while` петљи није више задовољен:
`while(!interrupted() && signal) { izvrsenje niti... }`
- Метода `interrupted()` враћа статус нити и мења статус нити на `false`.

Слично ће се десити уколико је `while` петља дефинисана са:

```
while( !isInterrupted() && signal) { izvrsenje niti... }
```

Метода `isInterrupted()` враћа статус нити али **не мења** статус нити (статус нити остаје на `true`).

Сценарио прекида нити логичком контролом – нит је блокирана

- нит је покренута и њен статус је постављен на `true`.
- нит се блокира помоћу `sleep()` или `wait()` методе.
- позива се метода промени која мења атрибут сигнал на `false`.
- метода `run()` **не престаје** да се извршава јер је нит блокирана и контрола програма не долази до `while` наредбе:

```
while(signal){
    nit je blokirana ...
}
```

Можемо закључити да се наведеним сценаријом не прекида извршење нити.

Сценарио прекида нити `interrupt` методом – нит је блокирана

- нит је покренута и њен статус је постављен на `true`.
- нит се блокира помоћу `sleep()` или `wait()` методе.
- позива се метода `interrupt()`. Пошто је нит блокирана генерише се изузетак `InterruptedException`. Статус нити се поставља на `true`.
- метода `run` престаје да се извршава јер је генерисан изузетак:

```
public void run(){
    try {
        ...
        while(signal){nit je blokirana
        ...
    }
    catch(InterruptedException e) {
    }
...
}
```

Треба обратити пажњу на следећи потенцијални проблем уколико је try/catch блок унутар while петље:

```
public void run(){
    while (...){
        try {
            nit je blokirana ...
        } catch(InterruptedException e) {
            ...
        }
    }
}
```

У наведеном случају неће доћи до прекида нити јер ће контрола програма остати у while петљи.

У следећа 2 примера су дати примери прекида нити.

```
/*Primer NT4: Pokazati kako se prekida "uspavanost" niti pomocu metode interrupt()*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */
class NT4 extends Thread {
    NT4() {
        start();
    }
    public void run() {
        try {
            sleep(5000);
        } catch (InterruptedException e) {
        }
        System.out.println("Nova nit probudjena!");
    }
    public static void main(String args[]) throws Exception {
        NT4 nn = new NT4();
        Thread gnit = Thread.currentThread();
        gnit.sleep(2000);
        System.out.println("Glavna nit probudjena!");
        nn.interrupt(); // Kada se iskljuci ova naredba nova nit nn je uspavana
        // 5 sekundi, inace se odmah prekida "uspavanost" niti nn.
    }
}
/*
Rezultat:
Glavna nit probudjena
Nova nit probudjena
*/
```

Други пример је мало сложенији.

```
/*
Primer NT5: Pokazati kako se prekida izvrsenje niti u 3 situacije:
a) Ukoliko se nit izvrsava u while petlji i nit nije uspavana.
b) Ukoliko se nit izvrsava u while petlji i nit je uspavana.
c) Ukoliko se nit izvrsava u while petlji i nit nije uspavana a zelimo je
prekinuti preko interrupt() metode.
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */

class NT5 extends Thread {
    NT5(int opcija) {
        signal = true;
    }
}
```

```

        opcija = opcijal;
        start();
    }

    public void run() {
        switch (opcija) {
            case 0:
                ObicanPrekidNiti();
                break;
            case 1:
                InterruptKojiNePrekidaIzvrsenjeNiti();
                break;
            case 2:
                InterruptKojiPrekidaIzvrsenjeNiti();
                break;
            case 3:
                InterruptKojiPrekidaNeuspavanuNit();
                break;
        }
    }

// Iz glavnog programa promenice se status signal promenljive sto ce dovesti do prekida
// niti.
    public void ObicanPrekidNiti() {
        while (signal) {
        }
        System.out.println("Nova nit 0 probudjena!");
    }

// Prekid "uspavanosti" niti vratice kontrolu izvrsenja niti u while petlju. Ovde je
// problem u tome sto se try/catch blok nalazi unutar while petlje.
    public void InterruptKojiNePrekidaIzvrsenjeNiti() {
        while (signal) {
            try {
                sleep(5000);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("Nova nit 1 probudjena!");
    }

// Prekid "uspavanosti" niti prekinuce izvrsenje.niti, jer je try/catch blok izvan
// while petlje a ne unutar nje.
    public void InterruptKojiPrekidaIzvrsenjeNiti() {
        try {
            while (signal) {
                sleep(5000);
            }
        } catch (InterruptedException e) {
        }
        System.out.println("Nova nit 2 probudjena!");
    }

// Ukoliko se desi da interrupt metoda pozove neuspavanu nit potrebno je izvrsiti
// dopunska kontrolu while petlje pomocu interrupted() metode koja vraca true ukoliko
// je za tekucu nit pozvana interrupt() metoda.
    public void InterruptKojiPrekidaNeuspavanuNit() {
        // ** while(!isInterrupted() && signal) { } // ne menja interrupt status niti
        while (!interrupted() && signal) {
            // menja interrupt status niti na false
            System.out.println("Nova nit 3 probudjena!");
            System.out.println("Status prekida : " + isInterrupted());
        }
        // Status prekida bi bio true da se izvrsila naredba **
    }

    public void prekid() {
        signal = false;
    }

    public static void main(String args[]) throws Exception {
        NT5 nn0 = new NT5(0);
        NT5 nn1 = new NT5(1);
        NT5 nn2 = new NT5(2);
        NT5 nn3 = new NT5(3);
        Thread gnit = Thread.currentThread();
        gnit.sleep(2000);
        System.out.println("Glavna nit probudjena!");
    }
}

```

```
    nn0.prekid();
    nn1.interrupt();
    nn2.interrupt();
    nn3.interrupt();
}
boolean signal;
int opcija;
*/
Rezultat:
Glavna nit probudjena
Nova nit 2 probudjena
Nova nit 0 probudjena
Nova nit 3 probudjena
Status prekida: false
*/
```

Задаци за вежбање:

```
/*
Primer NTZ1: Napisati 2 niti tako da jedna od niti u toku svog izvrsavanja prekine
izvrsavanje druge niti.
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */

class NTZ1 implements Runnable {
    NTZ1(NTZ1 n1, String ime) {
        n = n1;
        nit = new Thread(this, ime);
        nit.start();
    }
    public void run() {
        try {
            nit.sleep(5000);
        } catch (InterruptedException ie) {
        }
        System.out.println("Zavrsena je " + nit.getName() + " nit!");
    }
    void prekeidiDruguNit() {
        n.nit.interrupt();
    }
    public static void main(String args[]) {
        NTZ1 n1 = new NTZ1(null, "prva");
        NTZ1 n2 = new NTZ1(n1, "druga");
        n2.prekeidiDruguNit();
    }
    Thread nit;
    NTZ1 n;
}

/*
Rezultat:
Zavrsena je prva nit
Zavrsena je druga nit
*/
```

```
/*
Primer NTZ2: Napisati 2 niti tako da jedna od niti u toku svog izvrsavanja uspava
drugu nit.
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */

class NTZ2 extends Thread {
```

```

NTZ2(NTZ2 n1, String ime) {
    super(ime);
    n = n1;
    start();
}
public void run() {
    while (signal) {
        if (getName().equals("prva") == true) {
            System.out.println("Prva nit se izvrsava!!!");
        }
        if (getName().equals("druga") == true) {
            System.out.println("Druga nit se izvrsava!!!");
        }
    }
}
void promeni() {
    signal = false;
}
void uspavaj() throws InterruptedException {
    System.out.println("Uspavana je prva nit");
    n.sleep(10);
    System.out.println("Probudjena je prva nit");
}
public static void main(String args[]) throws InterruptedException {
    NTZ2 n1 = new NTZ2(null, "prva");
    NTZ2 n2 = new NTZ2(n1, "druga");
    n2.uspavaj();
    n1.promeni();
    n2.promeni();
}
boolean signal = true;
NTZ2 n;
}

/*
Rezultat:
Uspavana je prva nit
Prva nit se izvrsava
Probudjena je prva nit
*/

```

Питања:

1. Да ли у наведеном примеру прва нит у току свог извршења успављује другу нит (погледати резултат)?
2. Зашто прва нит у наведеном примеру није успавала другу нит?
3. Да ли нит, која је креирана преко главног програма, наставља да се извршава и након извршења главног програма?

```

/*
Primer NTZ21: Napisati 2 niti tako da jedna od niti u toku svog izvrsavanja uspava
drugu nit.
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */

class NTZ21 extends Thread {
    NTZ21(NTZ21 n1, String ime) {
        super(ime);
        n = n1;
    }
}

```

```
        start();
    }
    public void run() {
        try {
            while (signal) {
                System.out.println("Izvrsava se " + getName() + " nit");

                if (uspavan == true) {
                    System.out.println("Uspavana je " + getName() + " nit");
                    sleep(1); // 1 msec
                    System.out.println("Probudjena je " + getName() + " nit");
                    uspavan = false;
                }
            }
        } catch (InterruptedException ie) {
        }
    }
    void promeni() {
        signal = false;
    }
    void uspavaj() throws InterruptedException {
        n.uspavan = true;
    }
    public static void main(String args[]) throws InterruptedException {
        NTZ21 n1 = new NTZ21(null, "prva");
        NTZ21 n2 = new NTZ21(n1, "druga");
        n2.uspavaj();
        Thread.sleep(2); // 2 msec
        n1.promeni();
        n2.promeni();
    }
    boolean signal = true;
    boolean uspavan = false;
    NTZ21 n;
}
/*
Rezultat:
Izvrsava se prva nit
Uspavana je prva nit
Izvrsava se druga nit
Izvrsava se druga nit
...
Probudjena je prva nit
*/
```

Задатак NT33: Написати програм који ће да реализације сценарио прекида нити логичком контролом када нит није блокирана.

Задатак NT34: Написати програм који ће да реализације сценарио прекида нити *interrupt* методом када нит није блокирана.

Задатак NT35: Написати програм који ће да реализације сценарио прекида нити *interrupt* методом када је нит блокирана.

2.6 Коришћење метода `isAlive()` и `join()`

Уколико се жели утврдити, да ли се нит још извршава, користи се метода `isAlive()` класе `Thread`. Наведена метода враћа `true`, уколико се нит још извршава, односно `false`, уколико је нит извршена.

Уколико се жели сачекати са извршењем неког дела програма док се нека од нити не изврши користи се метода `join()` класе `Thread`.

```
/* Primer NT7: Onemoguciti da se glavna nit izvrsi pre nove
   niti koriscenjem metode join().
*/
class NT7 extends Thread {
    NT7(String nn) {
        super(nn);
        start();
    }
    public void run() {
        try {
```

```

        sleep(2000);
    } catch (InterruptedException e) {
    }
    System.out.println("Zavrsena nova nit!");
}
public static void main(String args[]) throws InterruptedException {
    NT7 nn1 = new NT7("Nova nit");
    nn1.join();
    System.out.println("Zavrsena glavna nit!");
}
/*
Rezultat:
Zavrsena nova nit!
Zavrsena glavna nit!
*/

```

Задаци за вежбање:

```

/*
Primer NTZ71: Omoguciti da se dve nove niti izvrse sekvensijalno.
Pre nego sto pocne da se izvrsava druga nit treba da se izvrsi prva nit.
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */
class NTZ71 extends Thread {
    NTZ71(String nn) {
        super(nn);
        start();
    }
    public void run() {
        try {
            sleep(2000);
        } catch (InterruptedException e) {
        }
        System.out.println("Zavrsena nit " + getName());
    }
    public static void main(String args[]) throws InterruptedException {
        NTZ71 n1 = new NTZ71("Nit1");
        n1.join();
        NTZ71 n2 = new NTZ71("Nit2");
    }
}
/*
Rezultat:
Zavrsena nit Nit1
Zavrsena nit Nit2
*/

```

```

/* Primer NTZ72: Pokrenuti dve nove niti. Iz glavnog programa blokirati jednu nit
dok se ne izvrsi druga nit koriscenjem metode join().
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */
class NTZ72 extends Thread {
    NTZ72(String naziv) {
        super(naziv);
        start();
    }
    public void run() {
        int i = 1;
        try {
            while (i < 6) // Sta bi se desilo kada bi umesto i<6 stavili i<100
                // a metodu sleep postavili na 1 msec?
            {
                if (uspavan == true) {

```

```
        System.out.println("Uspavana je nit " + getName());
        sleep(1000);
    } else {
        System.out.println(i + "-ti prolaz niti" + getName());
        i++;
    }
}
} catch (InterruptedException ie) {
}
System.out.println("Zavrsena nit " + getName());
}

public static void main(String args[]) throws InterruptedException {
    NTZ72 n1 = new NTZ72("Nit1");
    NTZ72 n2 = new NTZ72("Nit2");
    n1.uspavan = true;
    n2.join();
    n1.uspavan = false;
}
boolean uspavan = false;
/*
Rezultat:
Uspavana je nit Nit1
1-ti prolaz nitiNit2
2-ti prolaz nitiNit2
3-ti prolaz nitiNit2
4-ti prolaz nitiNit2
5-ti prolaz nitiNit2
Zavrsena nit Nit2

1-ti prolaz nitiNit1
2-ti prolaz nitiNit1
3-ti prolaz nitiNit1
4-ti prolaz nitiNit1
5-ti prolaz nitiNit1
Zavrsena nit Nit1
*/

```

```
/*
Zadatak NTZ73: Kreirati novu nit i iz glavnog programa
pratiti kada ce se ona zavrsiti.
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */
class NTZ73 extends Thread {
    NTZ73(String naziv) {
        super(naziv);
        start();
    }
    public void run() {
        try {
            sleep(1);
        } catch (InterruptedException ie) {
        }
    }
}
public static void main(String args[]) throws InterruptedException {
    NTZ73 n1 = new NTZ73("Nit1");
    while (true) {
        if (n1.isAlive() == true) {
            System.out.println("Nit1 jos nije izvrsena");
        } else {
            System.out.println("Nit1 je izvrsena");
            break;
        }
    }
}
/*
Rezultat:
Nit1 jos nije izvrsena
Nit1 jos nije izvrsena

```

```
...
Nit1 je izvrnsena
*/
```

Задатак НТ374: Омогућити да се две нове нити изврше секвенцијално без коришћења `join()` методе.

2.7 Приоритет извршавања нити

У Јави свака нит има свој приоритет. Подразумевано нит наслеђује приоритет родитељске нити. Приоритет нити може да се промени са методом `setPriority()`. Приоритет може да узме једну од вредности из опсега од 1 (`MIN_PRIORITY`) до 10 (`MAX_PRIORITY`). Подразумевани приоритет сваке нити, осим ако се другачије не зада је 5 (`NORM_PRIORITY`).

Нит неког нивоа приоритета се извршава док:

- се не позове метода `yield()` која прекида извршење текуће нити, допуштајући да друга нит, истог или вишег приоритета почне да се извршава.
- не пређе у стање “блокиран” или “свршен”.
- нит вишег приоритета не постане извршна (зато што се “пробудила” или је у/и операција комплетирана или је неко позвао `notify()` методу).

Уколико постоји више нити истог приоритета, поставља се питање о редоследу извршења нити. Јава не гарантује да ће све нити, истог приоритета, бити једнако третиране (да ће све добити исто процесорско време – у пракси они добијају приближно исто време, али никада исто). Извесно је да ће свака нит, истог приоритета добити шансу да се паралелно извршава. Нит вишег приоритета добија више процесорског времена од нити нижег приоритета. То значи да ће нит вишег приоритета брже да се изврши од нити нижег приоритета.

Оно што представља реалан проблем при извршењу вишенитног Јава програма (у смислу транспарентности његовог извршења на различитим оперативним системима), се односи на однос између броја нивоа приоритета нити код Јаве и броја нивоа приоритета нити конкретног оперативног система на коме ће бити извршен Јава програм. На пример оперативни систем NT 4.0 има 7 нивоа приоритета док Јава има 10 нивоа приоритета. Поставља се питање како ће се нивои приоритета из Јаве пресликати у нивое приоритета NT 4.0 оперативног система. Проблем ће се јавити ако се различити нивои приоритета код Јаве пресликају у исти ниво приоритета код NT 4.0 оперативног система.

```
/*
Primer NT8: Napisati program koji ce da procentualno pokaze zauzetost procesora
od strane vise niti, u zavisnosti od prioriteta niti.

*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 * http://silab.fon.bg.ac.rs
 */

import java.text.*;
class NT8 extends Thread {
    NT8(String nn, int Prioritet) {
        super(nn);
        setPriority(Prioritet);
        start();
    }
    public void run() {
        while (signal) {
            brojac++;
        }
    }
}
```

```
    }
    void promeni() {
        signal = false;
    }
    public void prekini() {
        signal = false;
    }
    public static void main(String args[]) throws InterruptedException {
        long SumaKoraka;
        NumberFormat nf = NumberFormat.getNumberInstance();

        NT8 nn1 = new NT8("Nit1", 5); // Visi prioritet
        NT8 nn2 = new NT8("Nit2", 4); // Nizi prioritet
        Thread.sleep(10000); // Sto je veci broj sekundi (kada je isti prioritet-5)
        // odnos vremena tezi ka 50:50 inace ce da se povecava odnos vremena u
        // korist jedne od niti
        nn1.promeni();
        nn2.promeni();
        nn1.join();
        nn2.join();

        SumaKoraka = nn1.brojac + nn2.brojac;
        double p1 = (double) nn1.brojac / SumaKoraka * 100.00;
        double p2 = (double) nn2.brojac / SumaKoraka * 100.00;

        System.out.println("Nit 1 - broj koraka:" + nn1.brojac + " Procenat:" + p1);
        System.out.println("Nit 2 - broj koraka:" + nn2.brojac + " Procenat:" + p2);
    }
    private boolean signal = true;
    long brojac = 0;
}
/*
Rezultat:
a) 10 sekundi se izvrsavaju niti
Nit 1 ce zauzeti pribilzno 98% procesorskog vremena ako ima prioritet 10
Nit 2 ce zauzeti pribilzno 2% procesorskog vremena ako ima prioritet 1
b) 10 sekundi se izvrsavaju niti
Nit 1 ce zauzeti pribilzno 50% procesorskog vremena ako ima prioritet 5
Nit 2 ce zauzeti pribilzno 50% procesorskog vremena ako ima prioritet 5
c) 10 sekundi se izvrsavaju niti
Nit 1 ce zauzeti pribilzno 97% procesorskog vremena ako ima prioritet 5
Nit 2 ce zauzeti pribilzno 3% procesorskog vremena ako ima prioritet 4
*/
```

Задатак за вежбање:

Задатак NTZ4: Написати 2 нити са различитим приоритетима извршавања. У току рада измените приоритетете између нити. Показите како је трошено процесорско време од стране нити пре и после измене приоритета.

2.8 Себичне (*selfish*) нити

Нити би требале периодично да позивају `yield()` или `sleep()` методу како би другим нитима пружиле шансу да се изврше. На тај начин нит укида могућност да из неког разлога има монопол над системом. Оне нити које не дају другим нитима могућност да се изврше, називају се “себичне” нити.

2.9 Групе нити

Неки програми могу да садрже велики број нити. Тада је пожељно да се нити категоризују по функционалности, како би се њима лакше управљало. На пример, у случају интернет читача, који “скида” *web* страну са више слика, читач извршава више нити, за сваку слику по једна. Уколико корисник притисне дугме ”*stop*”, тада ће се прекинути пуњење текуће слике (прекинуће се извршење текуће нити). Тада је потребно да се и остале нити прекину.

Јава програмски језик, омогућава формирање групе нити (*thread group*).

Група нити се креира на следећи начин:

```
String imeGrupe = "novaGrupa";
ThreadGroup g = new ThreadGroup(imeGrupe);
```

Стринг који треба да идентификује групу мора имати особину једнозначности у односу на имена других група нити.

Нит, која ће се звати ”novaNit1” се додаје до групе g на следећи начин:

```
Thread t = new Thread(g, "novaNit1");
```

Уколико се жели проверити да ли је нека од нити још у стању “извршно”, користи се метода activeCount() на следећи начин:

```
if (g.activeCount() == 0) {
    // sve niti u grupi niti su zastavljene.
}
```

Уколико се желе прекинути све нити у групи нити позива се interrupt() метода над групом нити:

```
g.interrupt();
```

Група нити може да има своју подгрупу нити. Методе activeCount() и interrupt() се односе на текућу групу нити и све њене подгрупе нити. То значи, нпр. да када се прекида извршење једне групе нити, све њене подгрупе нити такође прекидају извршење.

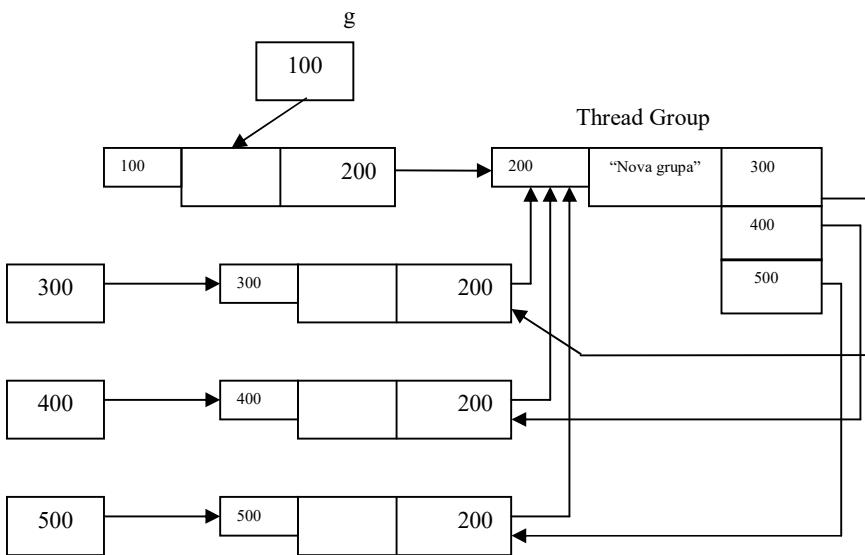
```
/*
Primer NT9: Pokazati kako se kreira i prekida grupa niti.
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */
class NT9 extends Thread {
    NT9(NT9G g, String imeNiti) {
        super(g.tg, imeNiti); // Vrsi se povezivanje izmedju grupe niti i konkretnie niti.
        start();
    }
    public void run() {
        while (!interrupted() && true) {
        }
        System.out.println("Prekinuta nit");
    }
    public static void main(String args[]) throws Exception {
        String imeGrupe = "novaGrupa";
        NT9G g = new NT9G(imeGrupe);
        NT9 t1 = new NT9(g, "novaNit1");
        NT9 t2 = new NT9(g, "novaNit2");
        NT9 t3 = new NT9(g, "novaNit3");
        g.prekini();
    }
}
class NT9G extends Thread {
    NT9G(String imegrupa) {
        tg = new ThreadGroup(imegrupa);
        start();
    }
    public void run() {
        while (true) {
            if (tg.activeCount() == 0) {
                System.out.println("Sve niti u grupi su prekinute! " + tg.activeCount());
                break;
            }
        }
    }
    void prekini() {
```

```

        tg.interrupt();
    }
    ThreadGroup tg;
}

```

Изглед оперативне меморије задатка NT9:



Задаци за вежбање:

```

/*
Primer NTZ5: Napisati grupu niti koja prati stanje nekog objekta.
Kada objekat dobije zadatu vrednost prekinuti izvršenje grupe niti.
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */

class Zalihe {
    int kolicina = 10;
    void smanji() {
        kolicina--;
    }
    int vratiKolicinu() {
        return kolicina;
    }
}

class NTZ5 extends Thread {
    NTZ5(NTZ5G g, String imeNiti, Zalihe z1) {
        super(g.tg, imeNiti);
        z = z1;
        start();
    }
    public void run() {
        smanji();
    }

    void smanji()// Kada se ubaci synchronized tada ne moze da se desi da se izvrsi nit
    // koja je usla u metodu posle neke druge niti.
    {
        while (!interrupted() && true) {
            z.smanji();
            System.out.println("Kolicina na zalihamu: " + z.vratiKolicinu() + " nit " +
getName());
        }
    }
}

```

```

        if (z.vratiKolicinu() <= 0) {
            getThreadGroup().interrupt();
        }
    }
    System.out.println("Prekinuta nit " + getName());
}

public static void main(String args[]) throws Exception {
    String imeGrupe = "novaGrupa";
    Zalihe z = new Zalihe();
    NTZ5G g = new NTZ5G(imeGrupe);
    NTZ5 t1 = new NTZ5(g, "novaNit1", z);
    NTZ5 t2 = new NTZ5(g, "novaNit2", z);
    NTZ5 t3 = new NTZ5(g, "novaNit3", z);

}
Zalihe z;
}

class NTZ5G extends Thread {
    NTZ5G(String imegrupe) {
        tg = new ThreadGroup(imegrupe);
        start();
    }
    public void run() {
        while (true) {
            if (tg.activeCount() == 0) {
                System.out.println("Sve niti u grupi su prekinute!");
                break;
            }
        }
    }
    ThreadGroup tg;
}
/*
Rezultat: (je promenljiv jer se ne moze predvideti koja nit ce da spusti kolicinu zaliha na 0)
Kolicina na zalihamu: 9 nit novaNit1
Kolicina na zalihamu: 8 nit novaNit1
Kolicina na zalihamu: 7 nit novaNit1
Kolicina na zalihamu: 6 nit novaNit1
Kolicina na zalihamu: 5 nit novaNit1
Kolicina na zalihamu: 4 nit novaNit1
Kolicina na zalihamu: 3 nit novaNit1
Kolicina na zalihamu: 2 nit novaNit1
Kolicina na zalihamu: 0 nit novaNit2
Prekinuta nit novaNit2
Kolicina na zalihamu: 1 nit novaNit3 // Nit 3 je usla pre niti 2 u metodu a posle nje se izvrsila.
Prekinuta nit novaNit3
Prekinuta nit novaNit1
Sve niti u grupi su prekinute!
*/

```

2.10 Синхронизација

Уколико се јави потреба да две или више нити деле заједнички ресурс, при чему нити не могу истовремено да користе заједнички ресурс, мора се обезбедити механизам које ће омогућити искључиви (ексклузивни) приступ једне нити до заједничког ресурса. Тек након завршетка обраде заједничког ресурса од стране једне нити, могуће је да друга нит, такође искључиво, приступи до њега.

Монитор обезбеђује механизам за искључиви приступ нити до заједничког ресурса. Када нека нит X уђе у монитор, ниједна друга нит не може у њега ући, све док нит X не изађе из њега. Поступак којим монитор обезбеђује наведени механизам назива се синхронизација.

Потреба истовременог приступа до дељеног објекта доводи до тзв. *race condition*⁶ ситуација.

2.10.1 Комуникација нити без синхронизације

Онемогућавање истовременог приступа нити до дељених објеката се назива синхронизација приступа. Уколико нема синхронизације приступа може се јавити следећа ситуација:

Уколико постоји банка са 10 рачуна (за сваки рачун се прави посебна нит), између којих се врши пренос новца на случајно изабран начин, може се десити да износ истог рачуна истовремено повећава 2 или више нити.

Нпр. `racun[5]` се истовремено повећава помоћу две нити.

Стање рачуна пре промене: `racun[5] = 500;`

Прва нит: `racun[5] +=100;`

Друга нит: `racun[5] +=50;`

Проблем се јавља на нивоу атомских операција наредби:

Прва нит: пуни се меморијски регистар `MR1` са износом од `racun[5]` преко наредбе `load()`.

`MR1 = 500`

`racun[5] = 500`

Након тога се повећава износ регистра за 100 преко наредбе `add()`.

`MR1 = 600`

`racun[5] = 500`

Уколико тада почне да се извршава друга нит, тада се пуни меморијски регистар `MR2` са износом од `racun[5]` са наредбом `load()`.

`MR2 = 500`

`racun[5] = 500`

Након тога се повећава износ регистра за 50 преко наредбе `add()`.

`MR2 = 550`

`racun[5] = 500`

Ако тада `racun[5]` добије вредност регистра `MR2` са наредбом `store()`.

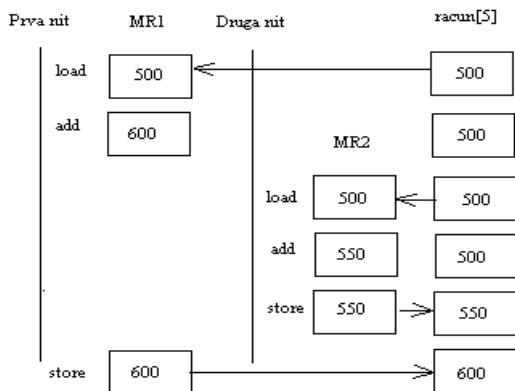
`racun[5] = 550`

На крају ће `racun[5]` добити вредност регистра `MR1` са наредбом `store()`.

`racun[5] = 600`

Добијени износ је погрешан. Валидна вредност коју би `racun[5]` требао да има је 650.

⁶ Када на тркама 2 такмичара, скоро у исто време пролазе кроз циљ, јавља се проблем прецизног одређивања ко је први прошао кроз циљ.



2.10.2 Закључавање објеката

Када нит позове методу објекта који треба да се синхронизује, објекат постаје закључан. Прецизније речено при позиву синхронизоване методе: “Нит налази једини кључ од објекта испред врата, убацује кључ у браву од објекта, откључава га, улази у објекат и браву са друге стране врата закључава. Брава остаје закључана све док нит не изађе из објекта, односно док нит не извади кључ из браве и стави га испред врата”. Нити које чекају на приступ синхронизованим методама дељеног објекта, могу приступити до несинхронизованих метода дељеног објекта.

2.10.3 Коришћење синхронизованих метода

У Јави сваки објекат има свој монитор. Да би се монитор покренуо потребно је да се објекту приступи преко једне од његових синхронизованих метода. Испред метода које треба синхронизовати ставља се кључна реч **synchronized**. Треба нагласити да извршавање једне од синхронизованих метода неког објекта онемогућава приступ до било које друге синхронизоване методе истог објекта. Так након завршетка извршења синхронизоване методе неког објекта могуће је позвати неку другу синхронизовану методу истог објекта. За несинхронизоване методе не важи наведено ограничење.

```
/*
 * Primer NT10S1: Omoguciti sinhronizaciju objekta, kome pristupa vise niti.*/
*/
* @author Sinisa Vlajic
* SILAB - Labartorija za Softversko Inzenjerstvo
* FON - Beograd
*http://silab.fon.bg.ac.rs
*/

class Nit extends Thread {
    Nit(String Naziv, int BrojNiti, Zalihe z) {
        this.Naziv = Naziv;
        this.BrojNiti = BrojNiti;
        this.z = z;
        start();
    }
    public void run() {
        int PreostalaKolicina = z.Uzmi();
        System.out.println("Nit:" + BrojNiti + " Preostala kolicina:" + PreostalaKolicina);
    }
    int BrojNiti;
    String Naziv;
    Zalihe z;
}

class Zalihe {
    private int Kolicina;
    Zalihe() {
        Kolicina = 10;
    }
}
```

```
int Uzmi() // Ukoliko se stavi synchronized int Uzmi() nece biti problema u rezultatu.
{
    Kolicina--;
    try {
        Thread.sleep(1000); // 1 sekunda pauze
    } catch (InterruptedException e) {
        System.out.println("Prekid niti");
    }
    return Kolicina;
}

class NT10S1 {
    public static void main(String args[]) {
        Zalihe z = new Zalihe();
        Nit nn1 = new Nit("Nit1", 1, z);
        Nit nn2 = new Nit("Nit2", 2, z);
        Nit nn3 = new Nit("Nit3", 3, z);
    }
}
/*
Rezultat:
Nit 1: Preostala kolicina:7
Nit 2: Preostala kolicina:7
Nit 3: Preostala kolicina:7
*/
```

Проблем је у томе што се не види ефекат једне од нити када се врати резултат, већ се види ефекат извршења свих нити. Наведени резултат показује да није извршена синхронизација објекта⁷ з преко методе `Uzmi()`. Да би се постигао наведени ефекат потребно је ставити `synchronized int Uzmi()` у класи `Zalihe`, уместо `int Uzmi()`:

```
/* primer: Sinhronizacija.java */
class Zalihe{
    private int Kolicina;
    Zalihe() {
        Kolicina = 10;
    }
    synchronized int Uzmi() {
        ...
    }
}

tada će rezultat program biti sledeći:
/*
Rezultat:
Nit 1: Preostala kolicina:9
Nit 2: Preostala kolicina:8
Nit 3: Preostala kolicina:7
*/
```

Задатак NTZ6: Коришћењем синхронизације онемогућити да објекту класе `Student` приступе истовремено 2 нити и промене његове податке.

2.10.4 Коришћење синхронизованих објеката

У случају да преко метода не може да се синхронизује приступ објекту, могуће је експлицитно синхронизовати сам објекат. Општи облик експлицитне синхронизације објекта је:

```
synchronized (objekat) {
    blok naredbi koje će biti sinhronizovane
}
```

⁷ Када се каже да се објекат синхронизује, то значи да се њему не може приступити истовремено преко две или више нити.

Уколико се жели постићи експлицитна синхронизација објекта, потребно је у предходном примеру да се метода `run()` класе `Nit` замени са:

```
/* primer: NT10S2.java */
class Nit extends Thread{
    ...
    public void run(){
        int PreostalaKolicina;
        synchronized (z){
            PreostalaKolicina = z.Uzmi();
        }
        System.out.println("Nit:" + BrojNiti + " Preostala kolicina:" + PreostalaKolicina);
    }
    ...
}

/*
Rezultat:
Nit 1: Preostala kolicina:9
Nit 2: Preostala kolicina:8
Nit 3: Preostala kolicina:7
*/
```

Када се синхронизација објекта врши преко методе, тада се пре извршења синхронизоване методе закључава објекат. У том статусу објекат остаје све док се не изврши синхронизована метода.

Када се синхронизација објекта врши експлицитно, тада се пре извршења блока наредби које треба синхронизовати закључава објекат. У том статусу објекат остаје све док се не изврши наведени блок наредби.

2.11 Комуницирање између нити

Комуникација између нити у Јави се постиже помоћу метода `wait()`, `notify()` и `notifyAll()` на следећи начин:

Када нит која је у синхронизовану методу, због неког услова не може да изврши комплетну методу, она ће или изаћи из методе необављена посла или ће остати у методи (у блокираном статусу) чекајући да је нека друга нит опет покрене. У блокирани статус метода прелази помоћу методе `wait()`. Тада нит напушта монитор у коме се налази и прелази у листу чекања (`wait list`). У блокираном статусу нит остаје све док нека друга нит не позове методу `notify()` или `notifyAll()`. Тада се нит опет враћа у монитор напуштајући листу чекања.

Метода `notify()` позива једну од нити из листе чекања, док метода `notifyAll()` позива све нити из листе чекања. Када више нити истовремено из листе чекања покуша да се врати у монитор, само ће једна од њих ући у монитор, док ће се остале вратити у листу чекања. Нити вишег приоритета прве ће се вратити у монитор из листе чекања. Када су нити истог приоритета, не постоји правило које може да прецизира која нит има првенство у односу на друге нити истог приоритета.

Методе `wait()`, `notify()` и `notifyAll()` могу се позвати једино из синхронизованих метода.

У следећем примеру две нити ће позивати две различите синхронизоване методе `Uzmi()` и `Stavi()`. На редослед позивања ових метода у овом случају се не може утицати. Тако ће једна од нити прва позвати неколико пута заредом своју методу, па ће тек онда друга нит да позове своју методу неколико пута заредом.

```
/*
```

Primer NT10S3: Napisati program koji će izvršavati dve niti koje će pozivati dve razlike sinhronizovane metode. Jedna nit može da pozove svoju metodu nekoliko puta zaredom.

```
/*
 * 
 */
import java.io.*;
class Proizvodjac extends Thread {
    Proizvodjac(String Naziv, Zalihe z) {
        this.Naziv = Naziv;
        this.z = z;
        start();
    }
    public void run() {
        for (int i = 0; i < 4; i++) {
            z.Stavi(i);
        }
    }
    String Naziv;
    Zalihe z;
}

class Potrosac extends Thread {
    Potrosac(String Naziv, Zalihe z) {
        this.Naziv = Naziv;
        this.z = z;
        start();
    }
    public void run() {
        for (int i = 0; i < 4; i++) {
            z.Uzmi(i);
        }
    }
    String Naziv;
    Zalihe z;
}
class Zalihe {
    boolean signal = false;
    OutputStream dat;
    Zalihe() {
        try {
            dat = new FileOutputStream("izlaz.txt");
        } catch (Exception e) {
            System.out.println("Izuzetak kod otv. dat!");
        }
    }
    synchronized void Uzmi(int i) {
        ZapamtiUDat("Uzmi " + i);
    }
    synchronized void Stavi(int i) {
        ZapamtiUDat("Stavi " + i);
    }
    void ZapamtiUDat(String poruka) {
        try {
            poruka = poruka + "\n";
            dat.write(poruka.getBytes());
        } catch (IOException io) {
            System.out.println("UI izuzetak:" + io);
        }
    }
}
class NT10S3 {
    public static void main(String args[]) {
        Zalihe z = new Zalihe();
        Proizvodjac nn1 = new Proizvodjac("Nit1", z);
        Potrosac nn2 = new Potrosac("Nit2", z);
        try {
            nn1.join();
            nn2.join();
        } catch (InterruptedException e) {
            System.out.println("Prekid glavne niti");
        }
        System.out.println("Zavrsetak programa!");
    }
}
```

```
/*
Rezultat: u datoteci izlaz.txt
Stavi 0
Stavi 1
Stavi 2
Stavi 3
Uzmi 0
Uzmi 1
Uzmi 2
Uzmi 3
*/
```

Уколико се жели постићи ефекат да 2 нити наизменично позивају своје методе треба урадити следеће:

```
/*
Primer NT10S31: Napisati program kod koga ce 2 niti
naizmenicno pozivati svoje metode.
*/

import java.io.*;
class Proizvodjac extends Thread {
    Proizvodjac(String Naziv, Zalihe z) {
        this.Naziv = Naziv;
        this.z = z;
        start();
    }
    public void run() {
        for (int i = 0; i < 4; i++) {
            z.Stavi(i);
        }
        z.Deblokiraj();
    }
    String Naziv;
    Zalihe z;
}

class Potrosac extends Thread {
    Potrosac(String Naziv, Zalihe z) {
        this.Naziv = Naziv;
        this.z = z;
        start();
    }
    public void run() {
        for (int i = 0; i < 4; i++) {
            z.Uzmi(i);
        }
        z.Deblokiraj();
    }
    String Naziv;
    Zalihe z;
}

class Zalihe {
    synchronized void Uzmi(int i) {
        try {
            notify();
            System.out.println("Uzmi " + i);
            wait();
        } catch (Exception e) {
        }
    }
    synchronized void Stavi(int i) {
        try {
            notify();
            System.out.println("Stavi " + i);
            wait();
        } catch (Exception e) {
        }
    }
    synchronized void Deblokiraj() {
        notifyAll();
    }
}

class NT10S31 {
```

```
public static void main(String args[]) {
    Zalihe z = new Zalihe();
    Proizvodjac nn1 = new Proizvodjac("Nit1", z);
    Potrosac nn2 = new Potrosac("Nit2", z);
    try {
        nn1.join();
        nn2.join();
    } catch (InterruptedException e) {
        System.out.println("Prekid glavne niti");
    }
    System.out.println("Zavrsetak programa!");
}
/*
Rezultat:
Stavi 0
Uzmi 0
Stavi 1
Uzmi 1
Stavi 2
Uzmi 2
Stavi 3
Uzmi 3
*/
```

У следећем примеру NT10S41 се онемогућава произвођачу и потрошачу да преко метода Uzmi() и Stavi() подигну стање залиха више од 10 и мање од 0.

```
import java.io.*;
class Proizvodjac extends Thread {
    Proizvodjac(String Naziv, Zalihe z) {
        this.Naziv = Naziv;
        this.z = z;
        start();
    }
    public void run() {
        int i = 0;
        while (true) {
            z.Stavi(i);
            i++;
        }
    }
    String Naziv;
    Zalihe z;
}
class Zalihe{
    synchronized void Uzmi(int i){
        try {
            notify();
            System.out.println("Uzmi "+ i);
            wait();
        }catch(Exception e){}
    }
    synchronized void Stavi(int i){
        try {
            notify();
            System.out.println("Stavi " + i);
            wait();
        }catch(Exception e){}
    }
    synchronized void Deblokiraj() {
        notifyAll();
    }
}
class Potrosac extends Thread {
    Potrosac(String Naziv, Zalihe z) {
        this.Naziv = Naziv;
        this.z = z;
        start();
    }
    public void run() {
        int i = 0;
        while (true) {
            z.Uzmi(i);
            i++;
        }
    }
}
```

```

        }
    }
    String Naziv;
    Zalihe z;
}

class NT10S41
{ public static void main(String args[]){
    Zalihe z = new Zalihe();
    Proizvodjac nn1 = new Proizvodjac("Nit1", z);
    Potrosac nn2 = new Potrosac("Nit2", z);
    try {
        nn1.join();
        nn2.join();
    } catch (InterruptedException e) {
        System.out.println("Prekid glavne niti");
    }
    System.out.println("Zavrsetak programa!");
}
}

```

Питање:

1. Када је нит блокирана у синхронизованој методи неког објекта, да ли нека друга нит може приступити том објекту преко друге или исте синхронизоване методе тог објекта?

Програм **NT10S5.java** бави се проблемом подизања и спуштања стања залиха робе у конкурентном окружењу једног произвођача и више купаца. У наведеном задатку, слично примеру `Sinhronizacija3.java`, синхронизоване методе се наизменично позивају.

```

/*
Primer NT10S5: Napraviti 3 niti, od kojih ce jedna da podize stanje zaliha (metoda Stavi()), za 1 komad, dok ce preostale dve niti da skidaju stanje zaliha (metoda Uzmi()). Jedna nit ce da smanjuje stanje za 2 komada, dok ce druga nit da smanjuje stanje za 3 komada.Omoguciti da se sinhronizovane metode Stavi() i Uzmi() naizmenicno izvrsavaju.
*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.rs
 */

import java.io.*;
class Proizvodjac extends Thread {
    Proizvodjac(String Naziv, Zalihe z, int Kol) {
        this.Naziv = Naziv;
        this.z = z;
        KolDaje = Kol;
        start();
    }
    public void run() {
        while (signal) {
            z.Stavi(KolDaje, Naziv);
        }
    }
    void prekini() {
        signal = false;
    }
    String Naziv;
    Zalihe z;
    int KolDaje;
    volatile boolean signal = true;
}
class Potrosac extends Thread {
    Potrosac(String Naziv, Zalihe z, int Kol, int Prioritet) {
        this.Naziv = Naziv;
        this.z = z;
        setPriority(Prioritet);
        KolUzima = Kol;
        start();
    }
}

```

```
    }
    public void run() {
        while (signal) {
            z.Uzmi(KolUzima, Naziv);
        }
    }
    void prekini() {
        signal = false;
    }
    String Naziv;
    Zalihe z;
    int KolUzima;
    volatile boolean signal = true;
}

class Zalihe {
    private int Kolicina;
    boolean signal = false;
    OutputStream dat;
    Zalihe() {
        Kolicina = 0;
        try {
            dat = new FileOutputStream("izlaz.txt");
        } catch (Exception e) {
            System.out.println("Izuzetak kod otv. dat!");
        }
    }
    synchronized void Uzmi(int Kol, String naziv) {
        if (!signal) {
            try {
                ZapamtiUDat("wait-uzmi ulaz: " + naziv);
                wait();
                ZapamtiUDat("wait-uzmi izlaz: " + naziv);
            } catch (InterruptedException e) {
                System.out.println("Izuzetak!");
            }
        }
        signal = false;
        if (Kolicina - Kol >= 0) {
            Kolicina = Kolicina - Kol;
            ZapamtiUDat("Potrosac:" + naziv + "Kolicina:" + Kolicina);
        } else {
            ZapamtiUDat("Potrosac-bezuspesno:" + naziv + "Kolicina:" + Kolicina);
        }
        notifyAll();
    }
    synchronized void Stavi(int Kol, String naziv) {
        if (signal) {
            ZapamtiUDat("wait-stavi ulaz: " + naziv);
            try {
                wait();
                ZapamtiUDat("wait-stavi izlaz: " + naziv);
            } catch (InterruptedException e) {
                System.out.println("Izuzetak!");
            }
        }
        signal = true;
        Kolicina = Kolicina + Kol;
        ZapamtiUDat("Proizvodjac:" + naziv + "Kolicina:" + Kolicina);
        notifyAll();
    }
    void ZapamtiUDat(String poruka) {
        try {
            poruka = poruka + "\n";
            dat.write(poruka.getBytes());
        } catch (IOException io) {
            System.out.println("UI izuzetak:" + io);
        }
    }
}
class NT10S5 {

    public static void main(String args[]) {
        Zalihe z = new Zalihe();
    }
}
```

```

Proizvodjac pr1 = new Proizvodjac("NIT-PROIZVODJAC ", z, 1);
Potrosac pot1 = new Potrosac("NIT-POTROSAC1 ", z, 2, 5);
Potrosac pot2 = new Potrosac("NIT-POTROSAC2 ", z, 3, 5);

try {
    Thread.sleep(100);
} catch (Exception e) {
}
pr1.prekini();
pot1.prekini();
pot2.prekini();
try {
    pr1.join();
    pot1.join();
    pot2.join();
} catch (InterruptedException e) {
    System.out.println("Prekid glavne niti");
}
System.out.println("Izadjite iz programa!");
}

// Rezultat:Pogledati u datoteci izlaz.txt

```

Програм NT10S6.java се бави проблемом издавања карти за аутобус у конкурентном окружењу више купаца.

```

/*
Programski zahtev: Napraviti program koji ce da prati izdavanje karata za autobus.
Autobus moze da ima najvise 52 mesta. Ukupno ima 120 potencijalnih kupaca karata.
Za svakog potencijalnog kupca pokrenuti nit koja ce biti aktivna sve dok se ili
ne zavrssi program ili dok kupac ne kupi kartu. Takođe pokrenuti 20 niti za
proizvoljnih 20 kupaca koji ce vratiti kartu. Niti koje prate kupce koji vracaju
karte ce biti aktivne sve dok se ili ne zavrssi program ili dok kupac ne vrati
kartu. Na kraju prikazati u datoteci izlaz.txt zauzeta mesta u autobusu
i ime kupca koji je kupio zauzeo to mesto.
*/
import java.io.*;
import java.util.Random;

class KupovinaKarte extends Thread {

    KupovinaKarte(String ImePutnika, Autobus aut) {
        this.ImePutnika = ImePutnika;
        setPriority(5);
        this.aut = aut;
        start();
    }

    public void run() {
        while (!aut.Uzmi(ImePutnika)) {
            try {
                Thread.sleep(45);
            } catch (Exception e) {
            }
        }
        String ImePutnika;
        Autobus aut;
    }
}

class VracanjeKarte extends Thread {

    VracanjeKarte(String ImePutnika, Autobus aut) {
        this.ImePutnika = ImePutnika;
        this.aut = aut;
        setPriority(5);
        start();
    }

    public void run() {
        while (!aut.Vrati(ImePutnika)) {
            try {
                Thread.sleep(15);
            } catch (Exception e) {

```

```
        }
    }
}
String ImePutnika;
Autobus aut;
}

class Sedista {

    int Mesto;
    boolean Zauzeto = false;
    String ImePutnika;
}

class Autobus {

    Sedista sed[];
    OutputStream dat;
    boolean signal = true;

    Autobus() {
        sed = new Sedista[52];
        for (int i = 0; i < 52; i++) {
            sed[i] = new Sedista();
            sed[i].Mesto = i;
            sed[i].ImePutnika = "nema";
        }
        try {
            dat = new FileOutputStream("izlaz.txt");
        } catch (Exception e) {
            System.out.println("Izuzetak kod otv. dat!");
        }
    }

    synchronized boolean Uzmi(String Ime) {
        if (!signal) // Kada se iz main() pozove prekini, da zaustavi nit.
        {
            return true;
        }

        for (int i = 0; i < 52; i++) {
            if (sed[i].Zauzeto == false) {
                sed[i].Zauzeto = true;
                sed[i].ImePutnika = Ime;
                ZapamtiUDat("Putnik " + Ime + " je uzeo kartu.");
                return true;
            }
        }
        ZapamtiUDat("Putnik " + Ime + " NIJE uzeo kartu.");
        return false;
    }

    synchronized boolean Vrati(String Ime) {
        if (!signal) // Kada se iz main() pozove prekini, da zaustavi nit.
        {
            return true;
        }

        for (int i = 0; i < 52; i++) {
            if (sed[i].ImePutnika.equals(Ime)) {
                sed[i].Zauzeto = false;
                sed[i].ImePutnika = "nema";
                ZapamtiUDat("Putnik " + Ime + "je vratio kartu.");
                return true;
            }
        }
        ZapamtiUDat("Putnik " + Ime + "ima nameru da vrati kartu ali je nije jos kupio.");
        return false;
    }

    void Prikazi() {
        ZapamtiUDat("Izvestaj o zauzetim mestima:");
        for (int i = 0; i < 52; i++) {
            if (sed[i].Zauzeto) {
                ZapamtiUDat("Putnik " + sed[i].ImePutnika + "ima sediste " + sed[i].Mesto);
            }
        }
    }
}
```

```

        }

    }

    void ZapamtiUDat(String poruka) {
        try {
            poruka = poruka + "\n";
            dat.write(poruka.getBytes());
        } catch (IOException io) {
            System.out.println("UI izuzetak:" + io);
        }
    }

    void Prekini() {
        signal = false;
    }
}

class NT10S6 {
    public static void main(String args[]) {
        Autobus aut = new Autobus();
        KupovinaKarte kk[] = new KupovinaKarte[120];
        VracanjeKarte vk[] = new VracanjeKarte[20];
        for (int i = 0; i < 120; i++) {
            kk[i] = new KupovinaKarte("Kupac" + i, aut);
        }

        Random r = new Random(0);
        for (int i = 0; i < 20; i++) {
            int vred = (int) (r.nextGaussian() * 51);
            vk[i] = new VracanjeKarte("Kupac" + Math.abs(vred), aut);
        }

        try {
            Thread.sleep(300);
        } catch (Exception e) {
        }

        aut.Prekini();
        try {
            for (int i = 0; i < 120; i++) {
                kk[i].join();
            }
            for (int i = 0; i < 20; i++) {
                vk[i].join();
            }
        } catch (InterruptedException e) {
            System.out.println("Prekid glavne niti");
        }

        aut.Prikazi();
        System.out.println("Izadjite iz programa!");
    }
}

// Rezultat:Pogledati u datoteci izlaz.txt

```

Задатак NTZ7: Направити програм који ће да прати рад 3 нити, тако да свака од нити периодично добије шансу да се изврши. Нити су истог приоритета. У раду користити методе `wait()` и `notify()`.

2.12 Међусобно блокирање нити

Нити могу у конкурентном раду да се међусобно блокирају (*deadlock*). Наведена ситуација настаје у следеће 2 ситуације:

Уколико се једна од нити блокира преко `wait()` методе а друга нит уђе такође преко `wait()` методе у блокаду, не деблокирајући пре тога преко `notify()` или `notifyAll()` методе прву нит, тада и једна и друга нит бивају блокиране и у том статусу остају док се не прекине програм.

```
// NT10S4.java
import java.io.*;
...
class Zalihe {

    boolean signal = false;
    ...
    synchronized void Uzmi(int i) {
        if (!signal) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("Izuzetak!");
            }
        }
        signal = false;
        ZapamtiUDat("Uzmi " + i);
        //notify(); Ukoliko se ne pozove notify() metoda,
        //           nikada nece moci da se odblokira nit
        // koja je usla u blokadu preko wait()
        // pri izvrsenju sinhronizovane metode Stavi().
    }

    synchronized void Stavi(int i) {
        if (signal) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("Izuzetak!");
            }
        }
        signal = true;
        ZapamtiUDat("Stavi " + i);
        //notify(); Ukoliko se ne pozove notify() metoda,
        //           nikada nece moci da se odblokira nit
        // koja je usla u blokadu preko wait()
        // pri izvrsenju sinhronizovane metode Uzmi().
    }
    ...
}
class NT10S4 {
    ...
}
```

Прва нит позива синхронизовану методу објекта А а друга нит позива синхронизовану методу објекта В. Код извршавања, синхронизована метода објекта А покушава да позове једну од синхронизованих метода објекта В. Такође код извршавања, синхронизована метода објекта В покушава да позове једну од синхронизованих метода објекта А. Тада долази до блокаде која ће трајати све док се не прекине програм.

Задатак NTZ71: Направити програм којим ће се заблокирати међусобно нити сходно сценарију б.

```
/*
Primer NT10S7: Napisati program kod koga ce prva nit pozivati sinhronizovanu metodu objekta
A, dok ce druga nit da poziva sinhronizovanu metodu objekta B. Kod izvrsavanja,
sinhronizovana metoda objekta A treba da pokusa da pozove jednu od sinhronizovanih metoda
objekta B. Takode kod izvrsavanja, sinhronizovana metoda objekta B treba da pokusava da
pozove jedna od sinhronizovanih metoda objekta A.
*/
import java.io.*;
class NIT1 extends Thread {
    NIT1(A a, B b) {
        this.a = a;
        this.b = b;
        start();
    }
    public void run() {
        a.mal(b);
    }
}
```

```

        A a;
        B b;
    }
    class NIT2 extends Thread {
        NIT2(A a, B b) {
            this.a = a;
            this.b = b;
            start();
        }
        public void run() {
            b.mb1(a);
        }
        A a;
        B b;
    }
    class A {
        synchronized void ma1(B b) {
            System.out.println("Metoda ma1()");
            try {
                Thread.sleep(5000);
            } catch (Exception e) {
            }
            b.mb2();
        }
        synchronized void ma2() {
            System.out.println("Metoda ma2()");
        }
    }
    class B {
        synchronized void mb1(A a) {
            System.out.println("Metoda mb1()");
            try {
                Thread.sleep(5000);
            } catch (Exception e) {
            }
            a.ma2();
        }
        synchronized void mb2() {
            System.out.println("Metoda mb2()");
        }
    }
}

class NT10S7 {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        NIT1 n1 = new NIT1(a, b);
        NIT2 n2 = new NIT2(a, b);
        try {
            n1.join();
            n2.join();
        } catch (InterruptedException e) {
            System.out.println("Prekid glavne niti");
        }
        System.out.println("Izadjite iz programa!");
    }
}
/*
Rezultat:
Metoda ma1()
Metoda mb1()
*/

```

Задатак NTZ8: Написати програм који ће да доведе до блокаде нити. Покушајте да направите пример који ће се разликовати од предходна 2 примера који се баве проблемом блокаде нити.

2.13 Коришћење цеви (pipes) за комуникацију између нити

Комуникација између нити се обавља тако што једна нит, назvana производијач (*producer*), генерише низ бајтова, док друга нит, назvana потрошач, чита тај низ бајтова. Ако бајтови нису расположиви за читање, нит потрошач се блокира. Ако нит

произвођач генерише више бајтова него што потрошач може да их прочита, нит произвођач се блокира. Класе које се користе у комуникацији између нити су: `PipedInputStream` и `PipedOutputStream` (када се разменjuју бајтови) као и `PipedReader` и `PipedWriter` (када се разменjuју unicode знаци).

Главни разлог за коришћење цеви је једноставност њиховог рада. Нит произвођач шаље бајтове до цеви не знајући која ће их нит прочитати. Нит потрошач чита бајтове из цеви не знајући која нит их је послала у цев. Преко цеви се може повезати међусобно више нити без бриге о њиховој синхронизацији.

```
/*
Primer NT11: Omoguciti komunikaciju izmedju niti proizvodjaca i niti potrosaca
preko cevi (pipes).
*/
import java.util.*;
import java.io.*;

public class NT11 {
    public static void main(String args[]) throws IOException {
        PipedOutputStream cout1 = new PipedOutputStream();
        PipedInputStream cin1 = new PipedInputStream(cout1);
        PipedOutputStream cout2 = new PipedOutputStream();
        PipedInputStream cin2 = new PipedInputStream();
        Proizvodjac pr = new Proizvodjac(cout1);
        Potrosac pot = new Potrosac(cin1);
        pr.start();
        pot.start();
    }
}

class Proizvodjac extends Thread {
    private DataOutputStream out;
    public Proizvodjac(OutputStream os) {
        out = new DataOutputStream(os);
    }
    public void run() {
        try {
            out.writeDouble(67.45);
            out.writeDouble(12.34);
        } catch (Exception ex) {
        }
    }
}

class Potrosac extends Thread {
    private DataInputStream in;
    public Potrosac(InputStream is) {
        in = new DataInputStream(is);
    }
    public void run() {
        try {
            double r1 = in.readDouble();
            double r2 = in.readDouble();
            System.out.println("Potrosac je primio brojeve: " + r1 + " " + r2);
        } catch (Exception ex) {
        }
    }
}
```

Задатак NTZ9: Осмислiti пример комуникације између нити помоћу цеви.

3. ПРОГРАМИРАЊЕ (РАД) У МРЕЖИ

3.1 Адреса рачунара

Сваки рачунар на глобалној мрежи (Интернету) има своју адресу (**Internet address**) која се састоји од 4 троцифрене броја, при чему бројеви могу узимати вредност из опсега од 0 – 255. На пример: 132.163.135.130 представља адресу сервера Националног института за стандарде у Колораду.

Повезивање рачунара у мрежу и пренос података између њих се остварује најчешће помоћу TCP/IP (**Transimion Control Protocol/Internet Protocol**) протокола. IP је одговоран да пронађе интернет адресу циљног рачунара и да усмери податке ка њему од извornог рачунара. TCP је задужен за успостављање и раскидање везе између рачунара, као и за одређене контролне функције. Често се за интернет адресу каже да је то IP адреса.

Класе које омогућавају рад у мрежи у Јави налазе се у пакету `java.net`.

Уколико се жели показати интернет адреса локалне машине користи се метода `getLocalHost()`. Класа која садржи информације о интернет адреси је `InetAddress`.

```
/* Primer Mrezal: Prikazati Internet adresu tekuce masine. */

/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/
import java.net.*;
class Mrezal {
    public static void main(String args[]) throws UnknownHostException {
        InetAddress tekucaAdresa = InetAddress.getLocalHost();
        System.out.println(tekucaAdresa);
    }
}
/*
Rezultat:
gsi/147.91.128.109
*/
```

Поред нумеричке адресе рачунара постоји и симболичка адреса рачунара, која се представља преко скупа симбола (знакова). Пример симболичке адресе је: `gsi.fon.bg.ac.rs`. Једна симболичка адреса може бити везана за више нумеричких адреса рачунара.

Симболичка адреса састоји се из два дела:

- матичног имена (**host name**) рачунара и
- имена домена (**domain name**) коме припада матични рачунар.

На пример симболичка адреса `gsi.fon.bg.ac.rs` састоји се из матичног имена: `gsi` и имена домена: `fon.bg.ac.rs`.

Сервис за именовање домена (**Domain Naming Service - DNS**), повезује симболичке адресе са интернет адресама. DNS сервиси се извршавају на DNS серверима.

Уколико се жели видети интернет адреса симболичке адресе, користи се метода `getByName()`.

```
/*
 * Primer Mreza2: Prikazati IP adresu masine koja ima simbolicku adresu "java.fon.bg.ac.rs" .
 */
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/
import java.net.*;
class Mreza2 {
    public static void main(String args[]) throws UnknownHostException {
        InetAddress tekucaAdresa = InetAddress.getByName("java.fon.bg.ac.rs");
        System.out.println(tekucaAdresa); // Prikazuje simbolicku i IP adresu.
        System.out.println(tekucaAdresa.getHostAddress()); // Prikazuje simbolicku adresu.
        System.out.println(tekucaAdresa.getHostName()); // Prikazuje IP adresu.
    }
}
/*
Rezultat:
java.fon.bg.ac.rs/147.91.128.18
147.91.128.18
java.fon.bg.ac.rs
*/
```

Уколико се желе видети све IP адресе које су везане за изабрано симболичко име користи се метода `getAllByName()`.

```
/*
 * Primer Mreza3: Prikazati IP adrese masina koje su vezane za simbolicku adresu:
 * "www.nba.com"
 */
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/
import java.net.*;
class Mreza3 {
    public static void main(String args[]) throws UnknownHostException {
        InetAddress nizIPAdresa[] = InetAddress.getAllByName("www.cnn.com");
        System.out.println("Broj:" + nizIPAdresa.length);
        for (int i = 0; i < nizIPAdresa.length; i++) {
            System.out.println(nizIPAdresa[i]);
            // Prikazuje IP adrese racunara.
        }
    }
}
/*
Rezultat: Broj:8
www.cnn.com/64.236.91.24
www.cnn.com/64.236.16.20
www.cnn.com/64.236.16.52
www.cnn.com/64.236.24.12
www.cnn.com/64.236.29.120
www.cnn.com/64.236.91.21
www.cnn.com/64.236.91.22
www.cnn.com/64.236.91.23
*/
```

Задатак MP31: Написати програм који ће преко стандардног улаза да прихвати произвољну симболичку адресу рачунара. Приказати IP адресе задате симболичке адресе.

Задатак MP32: Написати програм који ће преко стандардног улаза да прихвати низ симболичких адреса рачунара. Одредите за сваку симболичку адресу IP адресе. Приказати симболичке адресе и њихове IP адресе у сортираном редоследу (симболичке адресе које имају више IP адреса биће прве приказане).

3.2 URL адреса

Интернет и симболичке адресе рачунара омогућавају да се преко њих приступи до жељених рачунара у мрежи. Уколико се жели приступити до одређених сервиса и датотека на рачунару користи се **URL (Uniform Resource Locator)** адреса.

URL адреса састоји се из четири дела:

Протокол који се користи (**http, ftp, gopher или file**), који је одвојен : од остатка адресе. У последње време углавном се ради преко http протокола.

Адреса рачунара (интернет или симболичка адреса). Испред адресе се ставља ‘//’ а на крај адресе се ставља ‘/’ уколико не постоји порт, односно ‘:’ ако постоји.

Број порта (прикључка). Иза порта се ставља ‘/’. Овај део није обавезан.

Путања до датотеке, укључујући и име датотеке.

Следећи пример показује све делове задате **URL** адресе.

```
/* Programska zahtev Mreza4: Prikazati svaki od delova URL adrese. */
/*
@auther Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/

import java.net.*;

class Mreza4 {
    public static void main(String args[]) throws MalformedURLException {
        URL hp = new URL("http://java.fon.bg.ac.rs:80/index.htm");
        System.out.println("Protokol:" + hp.getProtocol());
        System.out.println("Port:" + hp.getPort());
        System.out.println("Racunar:" + hp.getHost());
        System.out.println("Datoteka:" + hp.getFile());
        System.out.println("Zajedno:" + hp.toExternalForm());
    }
}
/*
Rezultat:
Protokol: http
Port: 80
Racunar: java.fon.bg.ac.rs
Datoteka: /index.html
Zajedno: http://java.fon.bg.ac.rs:80/index.htm
*/
```

Уколико се жели прочитати произвoльна датотека на задатој **URL** адреси, користи се класа **URLConnection**. Следећи пример показује како се користи **URLConnection** класа.

```
/* Primer Mreza5: Prikazati sadrzaj datoteke index.html kojoj se pristupa pomocu URL
adrese.
*/
/*
@auther Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/

import java.net.*;
import java.io.*;
import java.util.Date;

class Mreza5 {

    public static void main(String args[]) throws Exception {
```

```

int c;
URL url = new URL("http://java.fon.bg.ac.rs/index.htm");
URLConnection urlc = url.openConnection();
InputStream is = urlc.getInputStream();
System.out.println("Datum:" + new Date(urlc.getDate()));
System.out.println("Vrsta sadrzaja:" + urlc.getContentType());
System.out.println("Rok trajanja:" + urlc.getExpiration());
System.out.println("Vreme zadnje izmene:" + new Date(urlc.getLastModified()));
while ((c = is.read()) != -1) {
    System.out.println((char) c);
}
is.close();
}
}

```

Задатак МР33: Успоставити везу са произвољном *URL* адресом на којој се налази произвољна датотека. Прочитати датотеку и запамтити њен садржај . Запамћени садржај претражити по задатој речи. Приказати редне бројеве позиција задате речи и укупан број њених појављивања.

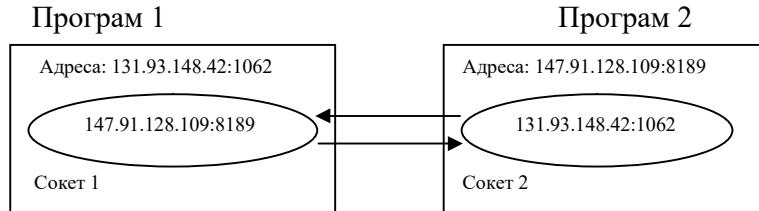
3.3 Сокети

Сокет, у ширем смислу, је механизам који омогућава комуникацију између програма који се извршавају на различитим рачунарима у мрежи. Он је пројектован тако да подржи имплементацију клијент/серверских апликација. Сокети користе *TCP/IP* протокол при повезивању, контроли и преносу података између два или више програма

При повезивању два програма преко сокета, по један сокет се генерише за сваки програм. Сваки од сокета садржи референцу на други сокет. То практично значи да први сокет садржи референцу на други сокет, док други сокет садржи референцу на први сокет.

3.3.1 Адреса сокета састоји се из два дела:

- адресе рачунара на коме се налази програм који је генерисао сокет
- броја порта који је генерисан помоћу сокета



3.3.2 Конекција између два програма је остварена када се успостави веза између њихових сокета. Сокети, у ужем смислу, представљају објекте помоћу којих се шаљу/прихватају подаци ка/од других сокета.

Сокет је по својој природи улазно-излазни ток и он се понаша на сличан начин као:

- системски објекат `System.in`, помоћу кога се подаци прихватају са стандардног улаза (тастатуре), док се код сокета подаци прихватају са спољашњег улаза (са мреже) од другог сокета.
- системски објекат `System.out`, помоћу кога се подаци шаљу ка стандардном излазу (екрану), док се код сокета подаци шаљу ка спољашњем излазу (ка мрежи) до другог сокета.

Сокети се повезују са улазно-излазним токовима на сличан начин као што је то случај са `System.in` и `System.out` објектима, када се жели извршити обрада података коју сокети разменjuју.

Типичан сценарио **развоја клијент/серверских апликација** помоћу сокета је следећи:

Покреће се програм на серверском рачунару. Интернет адреса сервера рачунара је 147.91.128.109.

Коришћењем наредбе:

```
ServerSocket ss = new ServerSocket(8189);
```

прави се сервер сокет који се повезује⁸ нпр. са портом 8189⁹.

Адреса сервера сокета `ss` је 147.91.128.109:8189.

Након тога извршава се наредба

```
Socket soketS = ss.accept();
```

којом сервер сокет долази у стање чекања на клијенте. Он ће бити у том стању све док се клијент не повеже са њим.

На клијентској страни се извршава наредба:

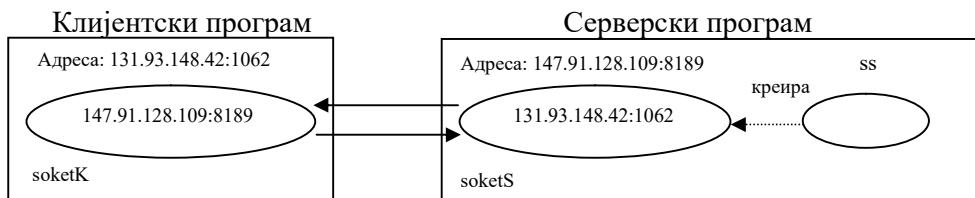
```
Socket soketK = new Socket("147.91.128.109", 8189);
```

којом се врши повезивање клијента са сервером¹⁰. Сокет који је направљен на клијентској страни прослеђује до серверског сокета своју адресу: 131.93.148.42:1062. У току успостављања везе (конекције) између серверског сокета и клијентског сокета дешавају се следеће активности:

Клијентски сокет `soketK` добија референцу (147.91.128.109", 8189) на серверски сокет.

Серверски сокет `ss` генерише нови сокет `soketS`. Адреса новог сокета ће бити иста као и адреса серверског сокета.

Нови сокет `soketS` добија референцу (131.93.148.42:1062) на клијентски сокет.



Како резултат наведених активности добијени су сокети `soketS` и `soketK` који показују један на другог. Пошто се жели извршити размена података између сокета, тако да исти могу да се обраде преко стандардних улазно-излазних уређаја, сокети се повезују са улазно-излазним објектима на следећи начин:

```
BufferedReader in = new BufferedReader(new InputStreamReader(X.getInputStream()));
PrintWriter out = new PrintWriter(X.getOutputStream(),true);
```

`X` ∈ (`soketS`, `soketK`)

Након тога је могуће да клијент пошаље поруку до сервера и обрнуто са наредбом:

```
out.println(" Y je spreman za rad\n");
```

`Y` ∈ (CLIENT, SERVER)

⁸ За сво време трајања програма сокет ослушкије (прати) рад наведеног порта.

⁹ Порт 8189 не користи ни један од стандардних сервиса.

¹⁰ Сокет на клијентској страни се повезује са сервер сокетом на серверској страни.

Клијент, односно сервер прихвата податке на следећи начин:

```
String line = in.readLine();
```

На крају клијент, односно сервер на стандардном излазу приказују поруку коју је примио:

```
System.out.println(" Z je primio poruku od Z1:" + line);
Z ≠ Z1
Z , Z1 ∈ {Klijent, Server}
```

Из наведеног се може закључити да постоји симетрија метода код размене података између сокета. То значи да место креирања сокета (клијентски или серверски програм), након успостављања конекције, не утиче ни на који начин на размену података. Сокети су равноправни у комуникацији.

Серверски и клијентски програм за наведени пример имају следећи изглед:

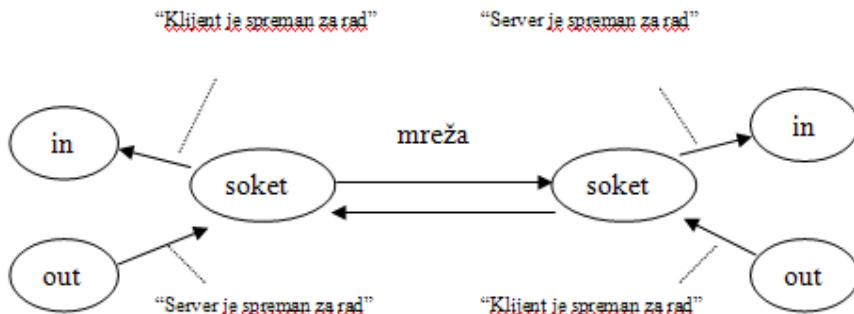
Серверски програм

```
/*
 * Primer MR6S: Napisati program koji ce kreirati serverski soket na portu 8189.
 * Nakon toga se povezati sa klijentskim soketom. Na kraju poslati poruku klijentskom
 * soketu.
 */
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/
import java.io.*;
import java.net.*;
public class ServerSoket {
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(8189);
            Socket soketS = ss.accept();
            BufferedReader in = new BufferedReader(new InputStreamReader(soketS.getInputStream()));
            PrintWriter out = new PrintWriter(soketS.getOutputStream(), true);
            out.println(" SERVER je spremam za rad\n");
            String line = in.readLine();
            System.out.println(" SERVER je primio poruku od klijenta:" + line);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Клијентски програм

```
/*
 * Primer MR6K: Napisati program koji ce kreirati klijentski soket, koji ce se
 * povezati sa serverskim soketom koji je podignut na racunaru cija je adresa
 * 147.91.128.109 na portu 8189. Poslati poruku serverskom racunaru.
 */
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/
import java.io.*;
import java.net.*;
public class SoketKlijent {
    public static void main(String[] args) {
        try {
            Socket soketK = new Socket("147.91.128.109", 8189);
            BufferedReader in = new BufferedReader(new InputStreamReader(soketK.getInputStream()));
            PrintWriter out = new PrintWriter(soketK.getOutputStream(), true);
```

```
    out.println(" KLIJENT je spreman za rad\n");
    String line = in.readLine();
    System.out.println(" KLIJENT je primio poruku od servera:" + line);
} catch (Exception e) {
    System.out.println(e);
}
}
```



Уколико се жељи видети:

- порт сокета на који показује други сокет, користи се метода `getPort()` .
 - IP адреса сокета на који показује други сокет, користи се метода `getInetAddress()` .
 - локални порт на коме је подигнут сокет користи се метода `getLocalPort()` . То значи да се пунा адреса сокета на који показује други сокет добија као: `getInetAddress() + getPort()` што се може видети у следећем примеру:

Серверски програм

```
/*
Primer MR7S: Prikazati IP adresu racunara i broj porta na kome se nalazi
klijentski soket. Na kraju prikazati broj porta na kome se nalazi serverski soket.
*/
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/

import java.io.*;
import java.net.*;

public class ServerSoket {
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(8189);
            System.out.println("SERVER");
            Socket soketS = ss.accept();
            // ia dobija IP adresu racunara na kome se nalazi klijentski soket
            InetAddress ia = soketS.getInetAddress();
            // getPort() metoda prikazuje port na kome se nalazi klijentski soket
            System.out.println(ia + " " + soketS.getPort());
            // getLocalPort() metoda prikazuje port na kome se nalazi serverski soket (8189)
            System.out.println(soketS.getLocalPort());
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Клијентски програм за наведени пример је:

```

/*
MR7K: Napisati program koji ce kreirati klijentski soket koji ce se
povezati sa serverskim soketom koji je podignut na lokalnom racunatu na portu 8189.
Prikazati IP adresu racunara i broj porta na kome se nalazi serverski soket.
Na kraju prikazati broj porta na kome se nalazi klijentski soket.
*/
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/

import java.io.*;
import java.net.*;

public class SoketKlijent {
    public static void main(String[] args) {
        try {
            String s;
            Socket soketK = new Socket("147.91.128.109", 8189);
            InetAddress ia = soketK.getInetAddress();
            System.out.println(ia + " " + soketK.getPort() + " " + soketK.getLocalPort());
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

```

Задатак МР34: Направити серверски и клијентски сокет који међусобно размењују поруке све док се не искљуци један од сокета (*chat* 2 сокета). Нацртати изглед оперативне меморије.

3.3.3 Повезивање сервера са више клијената

Сокети омогућавају да се више клијентских програма (клијент сокета) повеже на један серверски програм (серверски сокет). За сваки од клијентских сокета прави се по једна нит, тако да се у оквиру серверског програма конкурентно извршава више нити. Наведене нити могу да приступе заједничким ресурсима сервера.

У следећа 2 примера, више клијената повезаће се са серверским програмом. У току извршења, наведени клијенти ће приступати и обраћивати заједнички атрибут *Kolicina* серверског програма. У првом примеру повезаће се сервер са више telnet програма (клијенти), док ће у другом примеру сервер да се повеже са Јава клијентима, односно програмима који су направљени у Јави.

Server и Telnet програми (клијенти)

```

/*
Primer MR8S: Napisati program koji ce kreirati serverski soket na portu 8189. Serverski
soket moze da se poveze sa najvise 10 klijenata (klijentskih soketa). Za svakog
klijenta napraviti posebnu nit koja ce se nezavisno izvrsavati u odnosu na druge niti.
U okviru svake niti ce se vrsiti obrada kolicine robe (prodaja i nabavka).
Kolicina robe ce biti zajednicki atribut svih klijenata.
*/

/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/

import java.io.*;
import java.net.*;

```

```

public class ObradaRobe {
    public static void main(String[] args) {
        try {
            KreiranjeNiti kn = new KreiranjeNiti();
            kn.Kreiranje();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
class KreiranjeNiti {
    int kolicina;
    ObradaNiti on[];
    ServerSocket ss;

    KreiranjeNiti() {
        on = new ObradaNiti[10];
    }
    public void Kreiranje() {
        try {
            ss = new ServerSocket(8189);
            kolicina = 10;
            for (int brojKlijenta = 0; brojKlijenta < 10; brojKlijenta++) {
                Socket soketsS = ss.accept();
                System.out.println("Klijent " + brojKlijenta);
                on[brojKlijenta] = new ObradaNiti(soketsS, brojKlijenta, this);
                on[brojKlijenta].start();
            }
        } catch (Exception e) {
            System.out.println(e + " greska!");
        }
    }
}
class ObradaNiti extends Thread {
    public ObradaNiti(Socket soketS1, int c, KreiranjeNiti kn1) {
        soketS = soketS1;
        brojKlijenta = c + 1;
        kn = kn1;
    }
    public void run() {
        try {
            in = new BufferedReader(new InputStreamReader(soketS.getInputStream()));
            out = new PrintWriter(soketS.getOutputStream(), true);
            boolean done = false;
            while (!done) {
                out.println("Izaberite jednu od sledecih opcija:\n");
                out.println("1. PRODAJA 2. NABAVKA 3. IZLAZ");
                out.println(" ");
                String line = in.readLine();
                if (line.equals("")) {
                    done = true;
                } else {
                    switch (line.charAt(0)) {
                        case '1':
                            out.println("Klijent: (" + brojKlijenta + "): IZABRANA PRODAJA");
                            if (kn.kolicina == 0) {
                                out.println("Nema robe na zalihamu!");
                            } else {
                                kn.kolicina = kn.kolicina - 1;
                            }
                            break;
                        case '2':
                            out.println("Klijent: (" + brojKlijenta + "): IZABRANA NABAVKA");
                            kn.kolicina = kn.kolicina + 1;
                            break;
                        default:
                            done = true;
                    }
                }
                out.println("Ukupno je ostalo komada:" + kn.kolicina);
            }
            out.println("999");
            soketS.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

```

```

    }
    private Socket soketS;
    private int brojKlijenta;
    private KreiranjeNiti kn;
    BufferedReader in;
    PrintWriter out;
}

```

Када се позове *telnet* програм, који је клијентски програм појавиће се:

Microsoft Telnet >

Повезивање са серверским програмом се ради преко наредбе:

Open 127.0.0.1 8189

Детаљни опис програма MR8S:

На серверској страни коришћењем наредбе:

```
ServerSocket ss = new ServerSocket(8189);
```

прави се сервер сокет који се повезује¹¹ са портом 8189¹².

Након тога извршава се наредба

```
Socket soketK = ss.accept();
```

којом сервер сокет долази у стање чекања на клијенте. Он ће бити у том стању све док се клијент не повеже преко одговарајуће IP адресе и порта 8189 са њим.

На клијентској страни (телнет програм) се извршава наредба:

```
Socket socket = new Socket(147.91.128.109, 8189);
```

којом се врши повезивање клијента са сервером. Адреса сокета који је направљен на клијентској страни (преко наредбе `new Socket(147.91.128.109, 8189)`), се прослеђује до сервера.

Након тога извршава се наредба:

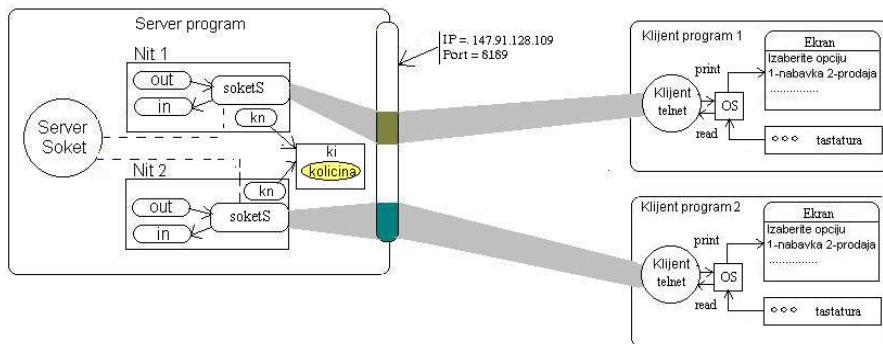
```
on[brojKlijenta] = new ObradaNiti(soketS, brojKlijenta, this);
```

која за сваки позив клијента прави нит, коме се као аргументи конструктора шаљу: сокет `SoketS` (садржи адресу клијентског сокета), који је генерирао серверски сокет `ss`, број клијента који је добијен по редоследу повезивања клијената са сервером и објекат типа `KreiranjeNiti` који контролише извршење комплетног програма. Новокреирана нит се чува у атрибуту `on[brojKlijenta]13` и она је одговорна за комуникацију између сервера и клијента.

¹¹ За сво време трајања програма сокет ослушкије (прати) рад наведеног порта.

¹² Порт 8189 не користи ни један од стандарних сервиса.

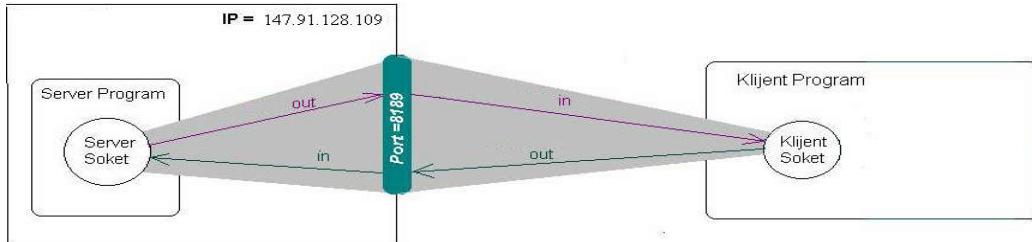
¹³ При чemu атрибут пр претставља низ могућих нити. Свака нит је везана за једног клијента



Новоје креирана нит покреће методу `start()` која врши иницијализацију нити. Након иницијализације метода `start()` покреће методу `run()`. У оквиру методе `run()` креирају се објекти `in` (класе `BufferedReader`) и `out` (класе `PrintWriter`) који претстављају улазне, односно излазне токове сокета¹⁴:

```
BufferedReader in = new BufferedReader(new InputStreamReader(soketS.getInputStream()));
PrintWriter out = new PrintWriter(soketS.getOutputStream(), true);
```

Све што се пошаље преко излазног тока сервера постаје улазни ток за клијента. Такође сви излазни токови од клијента постају улазни токови за сервер



У нашем примеру клијент је био програм ***telnet*** помоћу кога смо се повезали са сервером.

Наредбе као што су:

```
out.println("Izaberite jednu od sledećih opcija:\n");
out.println("1.PRODAJA. 2.NABAVKA 3. IZLAZ");
...
out.println(...)
```

приказивале су наведене садржаје на екрану преко ***telnet*** телнет програма.

Наредба `out.println()` сервера шаље податке до клијента (***telnet***).

Са друге стране наредба `in.readLine()` сервера прихвата податке од клијента.

При извршењу наредбе `run()`, свака од креираних нити чува референцу (атрибут `kn` који је добио `this`) према објекту који садржи атрибут (`Kolicina`), који је дељив за сваку од креираних нити.

За излаз из нити може се користити метода `stop()` али се иста сматра застарелом (она може да доведе до проблема у раду оперативног система). За излаз из нити најбоље је изаћи из `run()` методе, што је у нашем случају избор опције 3, која ће да доведе до прекида рада клијента. На крају рада са сокетом позива се метода `close()`.

Кораци код рада са програмом MR8S:

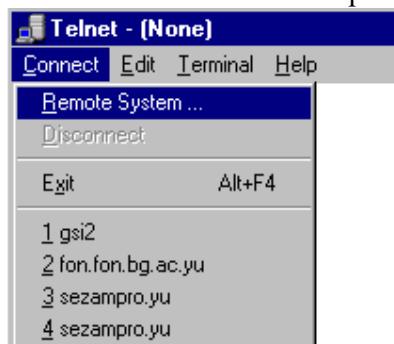
¹⁴ Помоћу наведених токова се врши двосмерна комуникација сервера и клијента.

1. Отворите едитор
2. Укуцајте програмски код за серверски сокет
3. Сачувавјте код - "ime_dat.java"
4. Компајлирајте код - "ime_dat.class"
5. Извршите програм - серверски сокет
6. Покрените телнет програме да симулирајте клијентске сокете

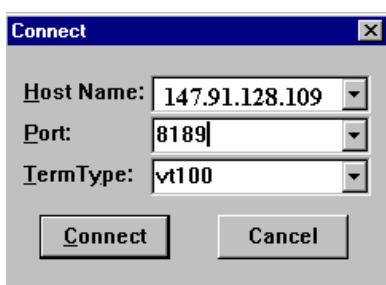
Код покретања *telnet* програма на екрану се појављује следеће:



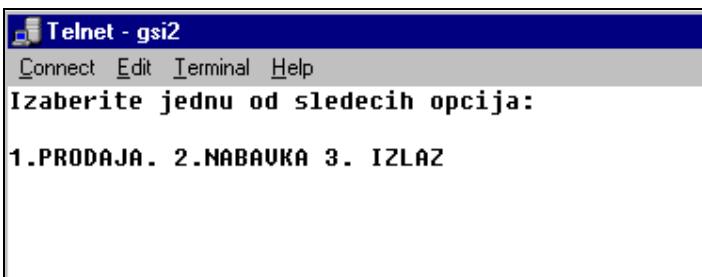
Након тога позвовите изаберите опцију *Connect-Remote System*



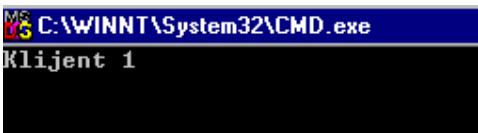
На екрану се појављује:



Изаберите IP рачунара (у нашем случају то је 147.91.128.109) и број порта 8189.
Након тога, уколико је успешно извршено логовање појавиће се:



На серверском програму, након повезивања клијента са њим појавиће се:



То значи да се први клијент повезао са серверским програмом.

На сличан начин можете да повежете друге клијенте са серверским програмом.

Суштина наведеног програма своди се на то да више клијената користи исти серверски програм, који ради над деливим атрибутом `Kolicina`. То практично значи да сваки клијент може да повећава, односно смањује заједнички атрибут `Kolicina`, коришћењем опција (1. PRODAJA, 2. NABAUKA).

Задатак MP35: Допуните наведени пример тако да серверски програм јави свим клијентима (који су са њим повезани) да је атрибут `Kolicina` добио вредност 0.

Сервер и Јава програми (клијенти)

У случају када је сервер повезан са више Јава програма (клијената), клијентски програм је следећи (серверски програм је исти као и програм MR8S):

```
/*
Primer MR9K: Napisati program koji ce kreirati klijentski soket koji ce se
povezati sa serverskim soketom koji je podignut na racunaru cija je IP adresa
147.91.128.109 na portu 8189.
*/
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/
import java.io.*;
import java.net.*;
public class SoketKlijent1 {
    public static void main(String[] args) {
        try {
            String s, line;
            Socket soketK = new Socket("147.91.128.109", 8189);
            BufferedReader in = new BufferedReader(new InputStreamReader(soketK.getInputStream()));
            PrintWriter out = new PrintWriter(soketK.getOutputStream(), true);
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            while (true) {
                line = in.readLine();
                // Kada serverski program posalje "999" prekinuce se izvrsenje klijenta.
                if (line.equals("999")) {
                    soketK.close();
                }
            }
        }
    }
}
```

```
        break;
    }
    System.out.println(line);
    // Kada serverski program posalje " ", klijent dobija mogucnost da izabere opciju.
    if (line.equals(" ")) {
        s = br.readLine();
        out.println(s);
    }
} //while
} catch (Exception e) {
    System.out.println(e);
}
}
```

Уколико се јави потреба да серверски програм пошаље клијентима обавештење, уколико вредност количине падне на 0, тада серверски програм има следећи изглед:

```

/*
Primer MR10S: Napisati program koji ce kreirati serverski soket na portu 8189. Serverski
soket moze da se poveze sa najvise 10 klijenata (klijentskih soketa). Za svakog
klijenta napraviti posebnu nit koja ce se nezavisno izvrsavati u odnosu na druge niti.
U okviru svake niti ce se vrsiti obrada kolicine robe (prodaja i nabavka).
Kolicina robe ce biti zajednicki atribut svih klijenata.
Kada roba padne na koliciju jednaku 0 server treba da o tome obavesti sve klijente.
*/
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/
import java.io.*;
import java.net.*;

public class ObradaRobe {
    public static void main(String[] args) {
        try {
            KreiranjeNiti kn = new KreiranjeNiti();
            kn.Kreiranje();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

class KreiranjeNiti {
    int kolicina;
    ObradaNiti on[];
    ServerSocket ss;
    int brojKlijenta;

    KreiranjeNiti() {
        on = new ObradaNiti[10];
    }
    public void Kreiranje() {
        try {
            ss = new ServerSocket(8189);
            kolicina = 10;
            for (brojKlijenta = 0; brojKlijenta < 10; brojKlijenta++) {
                Socket soketsS = ss.accept();
                System.out.println("Klijent " + brojKlijenta);
                on[brojKlijenta] = new ObradaNiti(soketsS, brojKlijenta, this);
                on[brojKlijenta].start();
            }
        } catch (Exception e) {
            System.out.println(e + " greska!");
        }
    }
    public void Prodaja(float p) {
        kolicina -= p;
        if (kolicina <= 0) {
            Azuriranje();
        }
    }
}

```

```

        }
        public void Nabavka(float n) {
            kolicina += n;
        }

        public void Azuriranje() {
            NitObavestenje obavestenje = new NitObavestenje(this, "Magacin se upravo izpraznio !");
            obavestenje.start();
            System.out.println("kreirao nit obavestenje");
        }
    }

    class ObradaNiti extends Thread {
        public ObradaNiti(Socket soketS1, int c, KreiranjeNiti kn1) {
            soketS = soketS1;
            brojKlijenta = c + 1;
            kn = kn1;
        }
        public void run() {
            try {
                in = new BufferedReader(new InputStreamReader(soketS.getInputStream()));
                out = new PrintWriter(soketS.getOutputStream(), true);
                boolean done = false;
                while (!done) {
                    out.println("Izaberite jednu od sledecih opcija:\n");
                    out.println("1.PRODAJA 2.NABAVKA 3. IZLAZ");
                    out.println(" ");
                    String line = in.readLine();
                    if (line.equals("")) {
                        out.println("Zavrsetak rada klijenta");
                        out.println("999");
                        done = true;
                    } else {
                        switch (line.charAt(0)) {
                            case '1':
                                out.println("Echo: (" + brojKlijenta + "): IZABRANA PRODAJA");
                                if (kn.kolicina - 4 < 0) {
                                    out.println("Nema dovoljno robe na zalihama da bi se izvršila prodaja!");
                                }
                            case '2':
                                out.println("Echo: (" + brojKlijenta + "): IZABRANA NABAVKA");
                                kn.Nabavka(2);
                            case '3':
                                out.println("Zavrsetak rada klijenta");
                                out.println("999");
                                done = true;
                            }
                        }
                    out.println("Ukupno je ostalo komada:" + kn.kolicina);
                }
                soketS.close();
                System.out.println("Zatvorio klijenta " + brojKlijenta);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
        private Socket soketS;
        public int brojKlijenta;
        private KreiranjeNiti kn;
        BufferedReader in;
        PrintWriter out;
    }

    class NitObavestenje extends Thread {
        public NitObavestenje(KreiranjeNiti k, String o) {
            kn = k;
            obavestenje = o;
        }
        private KreiranjeNiti kn;
        private String obavestenje;
    }
}

```

```

public void run() {
    for (int i = 0; i < kn.brojKlijenta; i++) {
        try {
            kn.on[i].out.println("Echo(" + kn.on[i].brojKlijenta + ") " + obavestenje);
            System.out.println("Obavestio klijenta " + (i + 1));
        } catch (Exception e) {
            System.out.println("Nema klijenta " + (i + 1));
        }
    }
}

```

Клијент има две нити, главну која ће да шаље податке до сервера, док ће друга нит да прихвата обавештење од сервера.

```

/*
Primer MR10K: Napisati program koji ce kreirati klijentski soket koji ce se
povezati sa serverskim soketom koji je podignut na racunatu cija je IP adresa
127.0.0.1 na portu 8189. Omoguciti da klijent moze u svakom trenutku da primi
poruku od servera.
*/
/*
@auther Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/
import java.io.*;
import java.net.*;

public class SoketKlijent1 {
    public static void main(String[] args) {
        try {
            String s;
            Socket soketK = new Socket("127.0.0.1", 8189);
            BufferedReader in = new BufferedReader(new InputStreamReader(soketK.getInputStream()));
            PrintWriter out = new PrintWriter(soketK.getOutputStream(), true);
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            boolean signal = true;
            NitKlijent nk = new NitKlijent(in);
            nk.start();
            while (true) { // Prihvata preko tastature podatke
                s = br.readLine();
                // Salje podatke do servera
                out.println(s);
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

class NitKlijent extends Thread {

    NitKlijent(BufferedReader in1) {
        in = in1;
        signal = true;
    }

    public void run() {
        try {
            while (signal) { // Prihvata podatke od servera
                String line = in.readLine();
                if (line.equals("999")) {
                    break;
                }
                // Prikazuje podatke na monitoru
                System.out.println(line);
            }
        } catch (Exception e) {
            System.out.println("Lose primljena poruka od servera!");
        }
    }
}

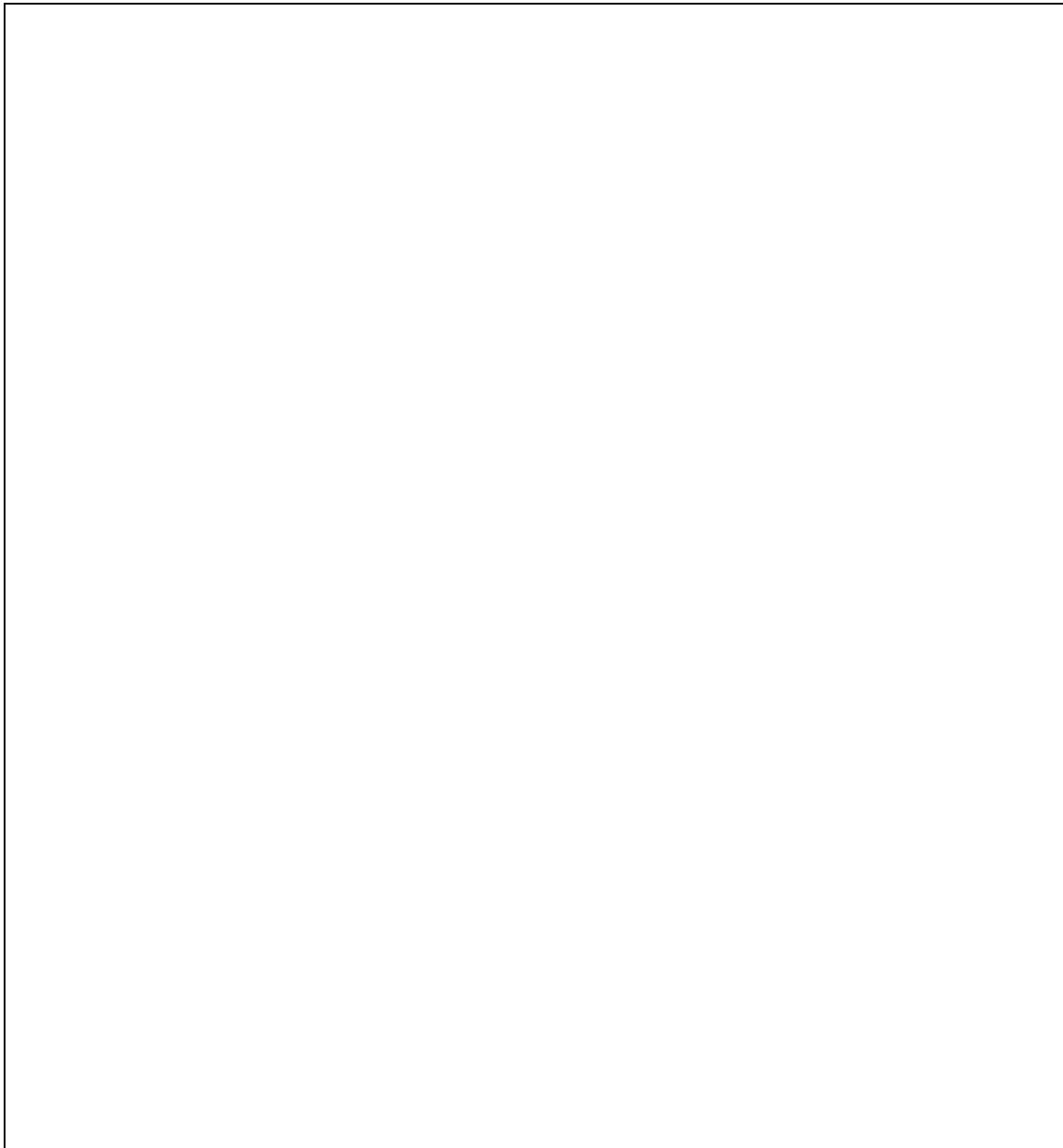
void Prekini() {
}

```

```
    signal = false;
}
boolean signal = true;
BufferedReader in;
}
```

Задатак MP36: Написати сервер који ће прекинути рад клијента у случају да клијент посаље поруку серверу да се њему (клијенту) прекине рад.

Вежба BMP31: Нацртати изглед оперативне меморије за програме PR10S и PR10K.



Уколико постоји потреба да сервер посредује у комуникацији више клијената (*chat* програм), дајемо следећи пример:

```
/*Primer MR11S: Napraviti serverski soket koji ce da posreduju u
 komunikaciju izmedju najvise 10 ucesnika
 */
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/

import java.io.*;
import java.net.*;

public class Chat {
    public static void main(String[] args) {
        try {
            PovezivanjeSaUcesnicima psu = new PovezivanjeSaUcesnicima();
            psu.Kreiranje();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

class PovezivanjeSaUcesnicima {

    int brojUcesnika;
    NitUcesnika[] np = new NitUcesnika[10]; // Moze najvise 10 ucesnika da se poveze sa
    // serverom.
    public void Kreiranje() {
        System.out.println("SERVER JE SPREMAN ZA RAD!!!\n");
        try {
            ServerSocket ss = new ServerSocket(8189);
            for (brojUcesnika = 0; brojUcesnika < 10; brojUcesnika++) {
                Socket soketS = ss.accept();
                System.out.println("Klijent " + (brojUcesnika + 1));
                np[brojUcesnika] = new NitUcesnika(soketS, brojUcesnika, this);
                np[brojUcesnika].start();
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    public void PosaljiSvimaPoruku(String poruka, int bklijenta) {
        NitObavestenje obavestenje = new NitObavestenje(this, poruka, bklijenta);
        obavestenje.start();
    }
}

class NitUcesnika extends Thread {

    public NitUcesnika(Socket soketS1, int brojUcesnikal, PovezivanjeSaUcesnicima psu1) {
        soketS = soketS1;
        brojUcesnika = brojUcesnikal + 1;
        psu = psu1;
        try {
            in = new BufferedReader(new InputStreamReader(soketS.getInputStream()));
            out = new PrintWriter(soketS.getOutputStream(), true);
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    public void run() {
        try {
            out.println("SERVER U PRIPREMI POSREDOVANJA U RAZGOVORU:\n");
            out.println("Unesi svoje ime:\n");
            ime = in.readLine();
            out.println("Uneto je ime:\n\n" + ime);
            out.println("SERVER JE SPREMAN ZA POSREDOVANJA U RAZGOVORU:\n");
            while (true) {

```

```

        String line = in.readLine();
        psu.PosaljiSvimaPoruku(line, brojUcesnika);
    }
} catch (Exception e) {
    System.out.println(e);
}
}
public Socket soketS;
public int brojUcesnika;
private PovezivanjeSaUcesnicima psu;
BufferedReader in;
PrintWriter out;
String ime;
}

class NitObavestenje extends Thread {

    public NitObavestenje(PovezivanjeSaUcesnicima psul, String o, int bucesnikal) {
        psu = psul;
        obavestenje = o;
        bucesnika = bucesnikal - 1;
    }
    private String obavestenje;

    public void run() {
        for (int i = 0; i < psu.brojUcesnika; i++) {
            try {
                psu.np[i].out.println(psu.np[bucesnika].ime + " :" + obavestenje);
            } catch (Exception e) {
                System.out.println("Nema klijenta " + (i + 1));
            }
        }
    }
}

private PovezivanjeSaUcesnicima psu;
int bucesnika;
}

```

```

/*
Primer MR11K: Napisati program koji ce kreirati klijentski soket koji ce se
povezati sa serverskim soketom koji je podignut na racunatu cija je IP adresa
127.0.0.1 na portu 8189. Serverski soket treba da omoguci menjusobnu razmenu
poruka vise ucesnika u razgovoru.
*/

/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/

import java.io.*;
import java.net.*;

public class SoketKlijent1 {

    public static void main(String[] args) {
        try {
            String s;
            Socket soketK = new Socket("127.0.0.1", 8189);
            BufferedReader in = new BufferedReader(new InputStreamReader(soketK.getInputStream()));
            PrintWriter out = new PrintWriter(soketK.getOutputStream(), true);
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            boolean signal = true;
            NitKlijent nk = new NitKlijent(in);
            nk.start();
            s = br.readLine();
            out.println(s);
            while (true) {
                s = br.readLine();
                out.println(s);
            }
        } catch (Exception e) {

```

```

        System.out.println(e);
    }
}

class NitKlijent extends Thread {
    NitKlijent(BufferedReader in1) {
        in = in1;
        signal = true;
    }
    public void run() {
        try {
            while (signal) {
                String line = in.readLine();
                System.out.println(line);
            }
        } catch (Exception e) {
            System.out.println("Lose primljena poruka od servera!");
        }
    }
    //void Prekini() { signal = false; }
    boolean signal = true;
    BufferedReader in;
}

```

Задатак MP7: Написати сервер програм који ће да прати број порука који је сваки од клијената послао у једној *chat* сесији. Уколико неко од клијената пређе дозвољен број порука упозорава се да више не може да шаље поруке. Уколико клијент након упозорења покуша да пошаље поруку сервер прекида његов рад и о томе обавестава остале клијенте.

Вежба BMP32: Нацртати изглед оперативне меморије за програме PR11S и PR11K.

3.4 Слање електронске поште

Применом концепта сокета, може се омогућити комуникација са *mail* серверима самим тим и слање електронске поште. При размени података са *mail* сервером користи се *smtp* протокол.

SMTP (*Simple Mail Transfer Protocol*) је стандардни *e-mail* протокол Интернета којим се дефинише формат поруке која се шаље. Код сваког програма за пријем електронске поште увек се дефинише одговарајући *SMTP* сервер.

Комуникација која се успоставља између *mail* сервера и сокета који је повезан са њим се обавља на тај начин да се излазним током од сокета ка *mail* серверу шаљу *smtp* команде, а у супротном смеру, кроз улазни ток сокета долазе одговори севера на те команде.

Основне *SMTP* команде које омогућавају конекцију са *mail* сервером и слање електронске поште су:

1. **MAIL< space>FROM reverse_path<enter>**

Ова команда означава почетак нове *mail* трансакције

reverse_path треба да садржи адресу рачунара пошиљаоца на коју треба да се шаљу поруке

2. **RCPT<space> TO <forward_path>**

Ова команда даје адресуједног примаоца. Она се може поновити произвольан број пута.

3. **DATA**<enter>
 Subject:Subject<enter>
 From:From<enter>
 <enter>
 <.>

Овде се уписује текст поруке. Ако желите да прикажете поља *Subject* и *From* можете их дефинисати на горе наведени начин. Крај *DATA* одељка се задаје тачком и то у одвојеном реду.

4. **QUIT**<enter>
 Затварање *SMTP* канала

```
/*
MR12:Napisati program koji ce uspostaviti konekciju sa proizvoljnim mail serverom
i poslati mu poruku.
*/
/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.rs
*/

import java.io.*;
import java.net.*;
import java.util.Date;

class SlanjeEMaila {

    InputStream is;
    OutputStream os;
    String status;

    SlanjeEMaila() {
        status = "";
    }

    void Slanje() throws IOException {//povezuje soket sa mail serverom
        Socket socket = new Socket("smtp.fon.bg.ac.rs", 25);
        //izlazni tok iz soketa postaje ulazni tok mail servisa
        is = socket.getInputStream();
        //Ulazni tok soketa postaje izlazni tok mail servisa
        os = socket.getOutputStream();
        SMTPTransakcija();
    }

    void odgovorServera() throws IOException {
        for (long l = System.currentTimeMillis(); System.currentTimeMillis() - l < 10000L;) //Daje se odredjeno vreme za odgovor (10 sekundi)
        {
            status = "";
            status = CitanjeIzToka();
            if (status.length() > 0) {
                System.out.println("ODGOVOR SERVERA: " + status);
            }
        }
    }

    //Metoda koja salje zahteve i prima odgovore od servera

    void SMTPTransakcija() throws IOException {
        Date date = new Date();
        odgovorServera();
        UpisivanjeUTok("HELO 127.0.0.1\r\n");
        odgovorServera();
        UpisivanjeUTok("MAIL FROM: vlajic\r\n");
        odgovorServera();
    }
}
```

```
UpisivanjeUTok("RCPT TO: vlajic@fon.bg.ac.rs\r\n");
odgovorServera();
UpisivanjeUTok("DATA\r\n");
odgovorServera();
UpisivanjeUTok("Subject: " + "Test poruka" + "\r\nFrom: Sinisa\r\nTo: ");
UpisivanjeUTok("vlajic@fon.bg.ac.rs\r\nDate: " + date.toString());
UpisivanjeUTok("\r\n\r\nDanas je lep dan!!!\r\n.\r\n");
odgovorServera();
UpisivanjeUTok("QUIT\r\n");
odgovorServera();
System.out.println("kraj    SMTP transakcije");
}

void UpisivanjeUTok(String s) throws IOException {
    if (s.length() > 0) {
        for (int i = 0; i < s.length(); i++) {
            os.write(s.charAt(i));
        }
    }
}

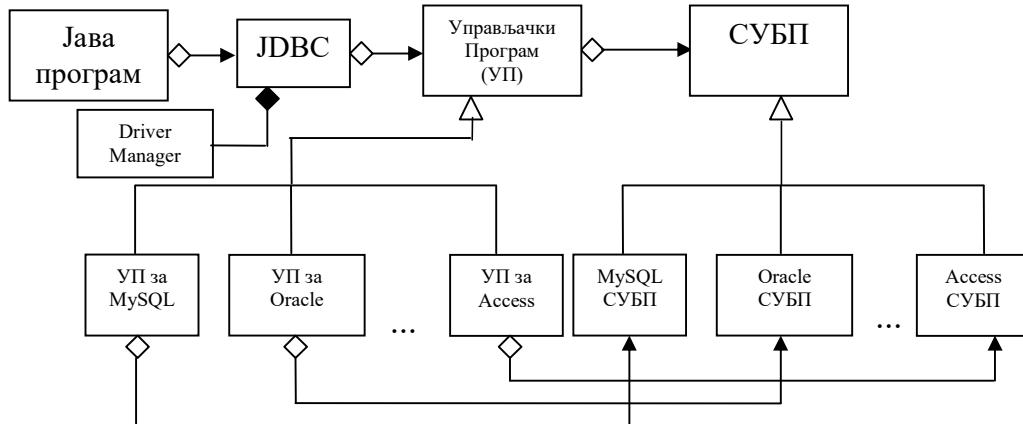
String CitanjeIzToka() throws IOException {
    String s;
    for (s = new String(""); is.available() > 0 && s.length() < 255;
         s = s + String.valueOf((char) is.read())) {

    }
    return s;
}

public static void main(String args[]) throws IOException {
    SlanjeEMaila sem = new SlanjeEMaila();
    sem.Slanje();
}
}
```

3. РАД СА БАЗОМ – JDBC

Повезивање неког програма који је написан Јави и неког од Система за управљање базом података (*MySQL*, *Oracle*, *SQL Server*,..., *MS Access*) ради се преко: а) Јавиног *JDBC* (*Java Database Connectivity*) API-а и б) управљачког програма (драјвера) који се прави посебно за сваки СУБП (Слика СлБП1).



СлБП1: Веза Јава програма са СУБП

Једном написан Јава програм који се извршава над неким Системом за управљање базом података (СУБП) може се извршавати непромењен и над другим СУБП¹⁵. Једини предуслов извршења неког Јава програма над неким СУБП јесте постојање управљачког програма за тај СУБП.

3.1 Поступак повезивања Јава програма и СУБП-а

Поступак повезивања Јава програма и базе података изабраног СУБП се изводи у следећим корацима:

- укључивање у Јава програм *JDBC API-а*
- учитавање управљачког програма у Јава програм
- успостављање конекције (везе) између Јава програма и базе података изабраног СУБП

У даљем тексту ће наведени кораци бити детаљно објашњени.

а) укључивање у Јава програм *JDBC API-а*

Укључивање *JDBC API-а* у Јава програм се ради преко следеће наредбе¹⁶:

```
import java.sql.*;
```

б) учитавање управљачког програма у Јава програм

Учитавање управљачког програма у Јава програм се ради преко следеће наредбе:

```
Class.forName("com.mysql.jdbc.Driver");
```

¹⁵ У суштини се Јава програм извршава над неком базом података која се налази унутар изабраног СУБП. Нпр. Јава програм се повезује са базом података *Student* која се налази унутар *MySQL* СУБП.

¹⁶ У пакету *java.sql* налазе се класе које се користе у раду са базама података СУБП.

```
/* Primer BP1: Pokazati na jednom primeru kako se vrši ucitavanje drajvera za MySQL SUBP i za
MS Access SUBP*/

class BP1 {
    public static void main(String[] args) {
        try {
            // Ucitavanje drajvera za MySQL bazu
            Class.forName("com.mysql.jdbc.Driver");
            // Ucitavanje drajvera za MS Access bazu
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            System.out.println("Upravljacki programi su ucitani!");
        } catch (ClassNotFoundException cnfe) {
            System.out.println("Nije ucitan upravljacki program: " + cnfe);
        }
    }
}
```

У даљем тексту покушаћемо да детаљније објаснимо шта је све потребно урадити у окружењу Јава програма како би се успешно извршила наредба:

```
Class.forName("com.mysql.jdbc.Driver");
```

Управљачки програм је у суштини Јавина *class* датотека која се обично налази у некој *jar* датотеци. Тако се на пример један од *MySQL* управљачких програма (*Driver.class*) налази у датотеци:

mysql-connector-java-3.1.12-bin.jar

Датотека *Driver.class* налази се унутар *mysql-connector-java-3.1.12-bin.jar*¹⁷ датотеке у пакету *com.mysql.jdbc*¹⁸

На основу наведеног може се закључити да:

1. параметар *forName* методе класе *Class*:

```
"com.mysql.jdbc.Driver"
```

у суштини представља путању (*com.mysql.jdbc*) унутар *mysql-connector-java-3.1.12-bin.jar* датотеке до *Driver.class* датотеке, која је, наглашавамо, **управљачки програм**.

2. уколико желимо да учитамо управљачки програм у Јава програм потребно је повезати Јава програм са *mysql-connector-java-3.1.12-bin.jar* датотеком.

Повезивање Јава програма са *mysql-connector-java-3.1.12-bin.jar* датотеком се ради на стандардан начин¹⁹, као и са било којом другом *jar* датотеком.

ц) успостављање конекције (везе) између Јава програма и базе података изабраног СУБП.

¹⁷ Драјвер за *MS Access* се налази у *rt.jar* датотеци.

¹⁸ Када би се датотека *mysql-connector-java-3.1.12-bin.jar* распаковала она би креирала фолдер *jdbc* који се налази испод фолдера *com/mysql*. Прецизније речено добија се следећа хијерархија фолдера:

com/mysql/jdbc. У фолдеру *jdbc* се налази управљачки програм (*Driver.class*) за *MySQL* СУБП.

¹⁹ Уколико се ради са неким од једноставнијих Јавиних развојних окружења (нпр. *TextPad*) тада је потребно у системској променљивој *CLASSPATH* да се се наведе пут до наведене *jar* датотеке:

C:\Install\MySQL5.0\mysql-connector-java-3.1.12\mysql-connector-java-3.1.12\mysql-connector-java-3.1.12-bin.jar;

Уколико се ради у неком од сложенијих Јавиних развојних окружења (нпр. *NetBeans*) могуће је у Јава програм на једноставан начин (*properties/AddJar*) укључити наведену *jar* датотеку .

Успостављање конекције се ради помоћу *JDBC DriverManager* класе. Као резултат успостављања конекције са базом података преко *JDBC DriverManager* класе добија се објекат класе *Connection* који чува конекцију ка бази података.

```
/*Primer BP21: Pokazati kako se uspostavlja veza (konekcija) sa MySQL bazom podataka.
*/
import java.sql.*;

class BP21 {
    public static void main(String[] args) {
        try {
            String dbUrl = "jdbc:mysql://127.0.0.1:3306/student";
            String user = "root";
            String pass = "root";
            Class.forName("com.mysql.jdbc.Driver");
            Connection naredba = DriverManager.getConnection(dbUrl, user, pass);
            System.out.println("Uspostavljena je konekcija izmedju driver manager-a i baze");
        } catch (ClassNotFoundException cnfe) {
            System.out.println("Nije ucitan upravljacki program: " + cnfe);
        } catch (SQLException sqle) {
            System.out.println("Greska: " + sqle);
        }
    }
}
```

Објашњење најважнијих делова програма BP21:

Када се учита *MySQL* управљачки програм у Јава програм преко наредбе:

```
Class.forName ("com.mysql.jdbc.Driver");
```

тада управљачки програм добија симболички назив преко кога се он касније позива²⁰. У случају наведене наредбе управљачки програм добија симболички назив²¹: *jdbc:mysql*

У примеру BP21 се налази наредба:

```
String dbUrl="jdbc:mysql://127.0.0.1:3306/student";
```

у којој променљива *dbUrl* добија следећу *url* адресу:

```
"jdbc:mysql://127.0.0.1:3306/student"
```

Наведена адреса се може декомпоновати на следеће делове:

Назив управљачког програма (*jdbc:mysql*) који је учитан преко наредбе:

```
Class.forName ("com.mysql.jdbc.Driver");
```

у Јава програм.

IP адреса (*127.0.0.1*) машине на којој се налази *MySQL* СУБП.

Порт (*3306*) на коме је подигнут *MySQL* СУБП.

Име базе података (*student*) са којом Јава програм успоставља конекцију преко наредбе:

```
Connection CONECTION=DriverManager.getConnection(dbUrl,user,pass);
```

На основу наведеног се може закључити да наредба:

```
String dbUrl="jdbc:mysql://127.0.0.1:3306/student";
```

²⁰ Тај симболички назив се чува у *Driver Manager*-у. Када се *Driver Manager* напуни са симболичким називом неког драјвера, за такав драјвер кажемо да је регистрован.

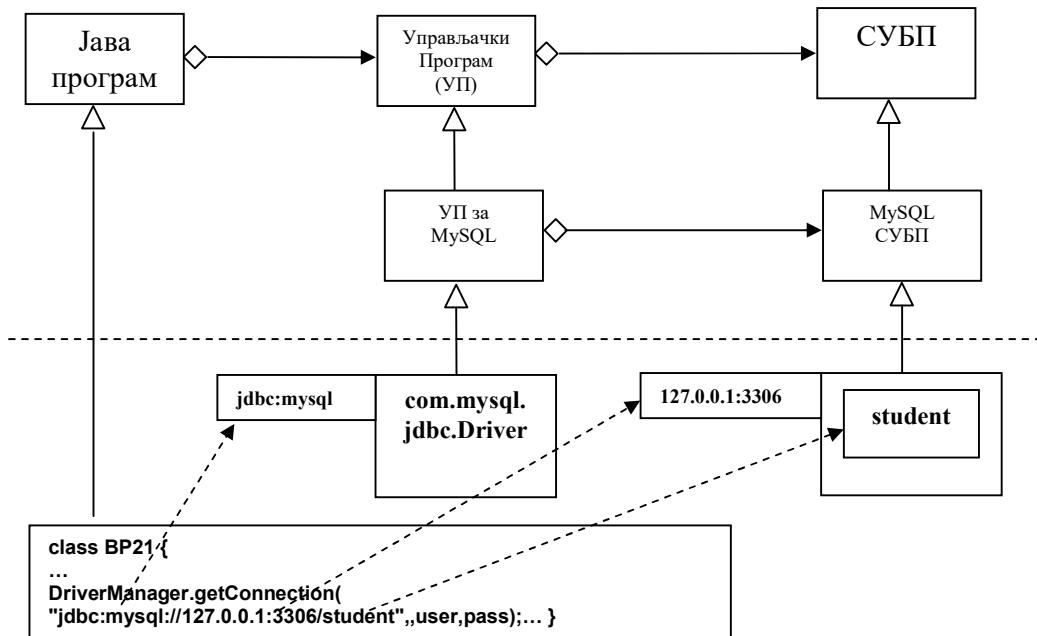
²¹ Када се учита *MS Access* управљачки програм у Јава програм преко наредбе:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
```

тада управљачки програм добија симболички назив:

```
jdbc:odbc
```

даје адресу до базе података **student** која се налази на *MySQL* серверу који се налази на адреси: **127.0.0.1:3306** преко управљачког програма чији је симболички назив: **jdb:mysql**. Наведени однос између Јава програма B21, управљачког програма се види на слици СлБП2.



СлБП2: Однос између Јава програма, управљачког програма и базе података

На крају се помоћу наредбе:

```
Connection CONECTION=DriverManager.getConnection(dbUrl, user, pass);
```

успоставља конекција са базом података између Јава програма и изабране базе података (*student*). Када се позове метода *getConnection()*, она итеративно пролази кроз драјвере регистроване у *DriverManager*-у и испитује да ли постоји драјвер са задатим називом. Уколико се пронађе драјвер, класа *DriverManager* прави објекат *Connection*, који успоставља везу са базом података, помоћу пронађеног драјвера.

Задатак BPZ1: Креирати базу података *Prodaja* у *MySQL* СУБП и остварити конекцију са њом. База података *Prodaja* има табелу *Račun* која има следеће атрибуте: *BrojRacuna* типа *String*, *NazivPartnera* типа *String*, *UkupnaVrednost* типа *double*, *Obradjen* типа *boolean*, *Storaniran* типа *boolean*.

3.2 Поступак извршења операција над базом података СУБП

Поступак извршења операција над СУБП се изводи у следећим корацима:

a) прављење објекта класе **statement**

Помоћу конекције која је успостављена преко наредбе:

```
Connection konekcija =DriverManager.getConnection(dbUrl, user, pass);
```

се прави објекат наредба класе *Statement* преко:

```
Statement naredba=konekcija.createStatement();
```

Помоћу објекта наредба се изводе операције над базом података преко:

```
naredba.executeQuery(upit);
```

У примеру BP31 ће се видети како се изводи операција *SELECT* над базом података.

```
/* Primer BP31: Napisati program kojim se prikazuje trenutni sadrzaj tabele Student.
Baza podataka, u kojoj se nalazi tabela Student, je realizovana preko MySQL i MS Access SUBP22*/
```

```
import java.sql.*;
```

```
class BP31 {
    public static void main(String[] args) {
        try {
            String dbUrl = new String();
            String user = "root";
            String pass = "root";

            if (args[0].equals("1")) {
                dbUrl = "jdbc:odbc:student";
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            }

            if (args[0].equals("2")) {
                dbUrl = "jdbc:mysql://127.0.0.1:3306/student";
                Class.forName("com.mysql.jdbc.Driver");
            }

            Connection konekcija = DriverManager.getConnection(dbUrl, user, pass);
            Statement naredba = konekcija.createStatement();
            String upit = "SELECT brind,ime,prezime FROM Student";
            ResultSet rs = null;
            try {
                rs = naredba.executeQuery(upit);
            } catch (SQLException sqle) {
                System.out.println("Greska u izvr. upita: " + sqle);
            }

            System.out.println("Trenutan izgled tabele studenata!");
            while (rs.next()) {
                System.out.println(rs.getString("brind") + " " + rs.getString("ime")
                    + " " + rs.getString("prezime"));
            }
        }

        naredba.close();
        konekcija.close();
    } catch (ClassNotFoundException cnfe) {
        System.out.println("Nije ucitan upravljacki program: " + cnfe);
    } catch (SecurityException se) {
        System.out.println("Nedozvoljena operacija: " + se);
    } catch (SQLException sqle) {
        System.out.println("Greska konekcije: " + sqle);
    }
}
}
```

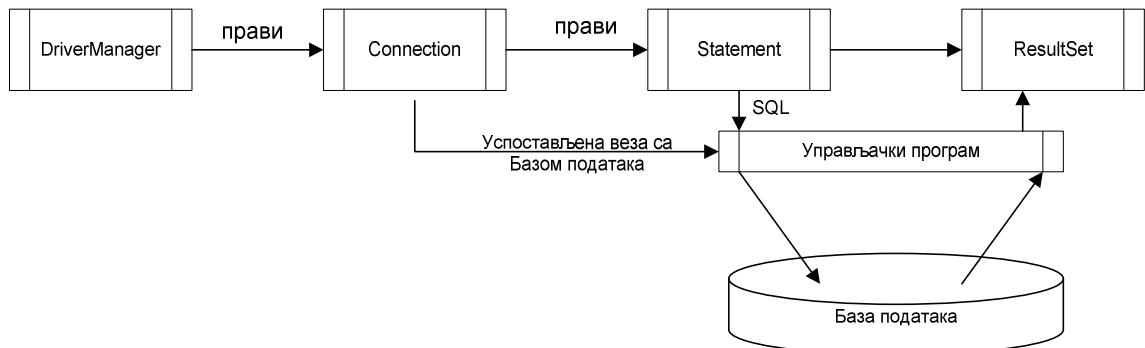
У наведеном примеру се упит:

```
"SELECT brind,ime,prezime FROM Student"
```

²² Уколико се жели из Јава програма приступити некој *MS Access* бази података која се налази на локалном рачунару, мора се приступити регистрацији базе података. Регистрација базе се на *Windows* оперативним системима обавља у *ODBC Data Source Administrator*-у који се налази на путањи: *Start -> Control Panel -> Administrative Tools -> Data Sources*.

шальје као параметар методе `executeQuery` која се извршава над објектом наредба. Резултат те наредбе се чува у `ResultSet`-у `rs`. У `ResultSet`-у се чувају сви слогови табеле `Student`. Кроз наведени `ResultSet` се пролази и приказује се његов садржај.

На слици СЛБП3 се виде одговорности класа које учествују у поступку извршења операције над базом података..



СЛБП3: Поступак извршења операције над базом података

Уколико желимо да покренемо програм BP31 преко bat датотеке њен садржај је:

```

set classpath=C:\Install\MySQL5.0\mysql-connector-java-3.1.12\mysql-connector-java-3.1.12\mysql-connector-java-3.1.12-bin.jar;
C:
CD \Predavanja\JDBCPetiCas\Zadaci
javac BP31.java
java BP31 2
PAUSE

```

```

/* Primer BP4: Napisati program koji u tabelu Student sa atributima broj indeksa, ime i prezime unosi novog studenta sa brojem indeksa 01/06 ,cije je ime Pera , a prezime Peric.*/
import java.sql.*;

class BP4 {
    public static void main(String[] args) {
        try {
            String dbUrl = "jdbc:odbc:student";
            String user = "";
            String pass = "";
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection CONECTION = DriverManager.getConnection(dbUrl, user, pass);
            String upit = "INSERT INTO Student(brind,ime,prezime) VALUES (?,?,?)";
            PreparedStatement PSTATEMENT = CONECTION.prepareStatement(upit);
            PSTATEMENT.setString(1, new String("01/06"));
            PSTATEMENT.setString(2, new String("Pera"));
            PSTATEMENT.setString(3, new String("Peric"));
            try {
                PSTATEMENT.executeUpdate();
                System.out.println("Novi student je zapamcen u bazi");
            } catch (SQLException e) {
                System.out.println("Izuzetak: " + e);
            }
            PSTATEMENT.close();
            CONECTION.close();
        } catch (ClassNotFoundException cnfe) {
            System.out.println("Nije ucitan upravljacki program: " + cnfe);
        } catch (SQLException sqle) {
            System.out.println("Greska kod konekcije: " + sqle);
        }
    }
}

```

```

/* Primer BP5: Napisati program koji ce da prihvata i cuva podatke u tabelu Student. Tabela se cuva u bazi koja je realizovana u okviru MySql SUBP. Naziv baze je Student. Program treba da bude

```

```

tronivojski, sto znaci da treba razdvojiti nivoe korisnickog interfejsa,
poslovne logike i baze podataka.
*/
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.*;

import javax.swing.*;
import org.netbeans.lib.awtextra.AbsoluteConstraints;
import org.netbeans.lib.awtextra.AbsoluteLayout;

/** *****PREZENTACIONI NIVO PROGRAMA***** */
public class BP5GUI extends JFrame {
    // Labela koja sadrzi naziv ekranske forme koja se otvara.
    private JLabel LNazivForme;

    // Polja preko kojih se obradjuju podaci.
    private JTextField PbrojInd;
    private JTextField PIime;
    private JTextField PPrezime;

    // Labele koje opisuju polja za obradu podataka.
    private JLabel LbrojInd;
    private JLabel LIime;
    private JLabel LPrezime;

    // Dugme preko koga se poziva sistemska operacija.
    private JButton BZapamti;

    // Glavni program
    public static void main(String args[]) {
        BP5GUI gui = new BP5GUI();
        gui.show();
    }

    // 1. Konstruktor ekranske forme
    public BP5GUI() {
        KreirajKomponenteEkranskeForme(); // 1.1
        PokreniMenadzeraRasporedaKomponeti(); // 1.2
        PostaviImeForme(); // 1.3
        PostaviPoljaZaPrihvataPodataka(); // 1.4
        PostaviLabeleZaPrihvataPodataka(); // 1.5
        PostaviDugmeZapamti(); // 1.6
        pack();
    }

    // 1.1 Kreiranje i inicijalizacija komponenti ekranske forme
    void KreirajKomponenteEkranskeForme() {
        LNazivForme = new JLabel();
        PbrojInd = new JTextField();
        PIime = new JTextField();
        PPrezime = new JTextField();

        LbrojInd = new JLabel();
        LIime = new JLabel();
        LPrezime = new JLabel();
        BZapamti = new JButton();
    }

    // 1.2 Kreiranje menadjera rasporeda komponenti i njegovo dodeljivanje do
    // kontejnera okvira(JFrame komponente).
    void PokreniMenadzeraRasporedaKomponeti() {
        getContentPane().setLayout(new AbsoluteLayout());
    }

    // 1.3 Odredivanje naslovnog teksta i njegovo dodeljivanje do kontejnera
    // okvira.
    void PostaviImeForme() {
        LNazivForme.setFont(new Font("Times New Roman", 1, 12));
        LNazivForme.setText("UNOS STUDENATA");
        getContentPane().add(LNazivForme, new AbsoluteConstraints(20, 10, -1, -1));
    }

    // 1.4
    void PostaviPoljaZaPrihvataPodataka() { // Dodeljivanje pocetne vrednosti i
        // formata polja.
        PbrojInd.setValue(new String(""));
    }
}

```

```

PIme.setValue(new String(""));
PPrezime.setValue(new String(""));
// Polja se dodaju kontejneru okvira (JFrame).
getContentPane().add(PprojInd, new AbsoluteConstraints(120, 70, 100, -1));
getContentPane().add(PIme, new AbsoluteConstraints(120, 100, 100, -1));
getContentPane().add(PPrezime, new AbsoluteConstraints(120, 130, 100, -1));
}

// 1.5
void PostaviLabeleZaPrihvataPodataka() {
    LbrojInd.setText("Broj indeksa:");
    getContentPane().add(LbrojInd, new AbsoluteConstraints(20, 70, 100, -1));
    LIme.setText("Ime:");
    getContentPane().add(LIme, new AbsoluteConstraints(20, 100, 100, -1));
    LPrezime.setText("Prezime:");
    getContentPane().add(LPrezime, new AbsoluteConstraints(20, 130, 100, -1));
}

// *****
// 1.6
// Deklarisanje i kreiranje objekta koji je odgovoran za logiku programa.
Logika l = new Logika();

void PostaviDugmeZapamti() {
    BZapamti.setText("Zapamti");
    BZapamti.addActionListener(new ActionListener() {
        /**
         * DOGADAJA KOJI INICIRA POZIV SISTEMSKE OPERACIJE
         */
        public void actionPerformed(ActionEvent evt) {
            try {
                /**
                 * POZIV SISTEMSKE OPERACIJE**
                 */
                String slog = "!" + PprojInd.getValue() + ", "
                    + (String) PIme.getValue() + ',' + (String) PPrezime.getValue()
                    + "!";
                l.pamtiSlog(slog);
            } catch (Exception e) {
                l.postaviPoruku("Nije zapamcen slog!");
            }
            PrikaziPoruku();
            show();
        }
    });
}

// Kraj addActionListener metode
// *****

getContentPane().add(BZapamti, new AbsoluteConstraints(260, 60, -1, -1));
}

void PrikaziPoruku() {
    if (!l.signal) {
        Poruka p = new Poruka(this);
        p.prikazi(l.poruka);
        p.show();
    }
}

class Poruka extends JDialog {
    public Poruka(JFrame roditelj) {
        super(roditelj, "UPOZORENJE", true);
    }

    void prikazi(String poruka) {
        Box b = Box.createVerticalBox();
        b.add(new JLabel(poruka));
        getContentPane().add(b);
        setSize(350, 70);
    }
}

// Logicki nivo programa i nivo baze podataka, odnosno klase Logika i
// BazaPodataka
// nalaze se u datoteci BP5LogikaiBaza.java

/**
 * *****LOGICKI NIVO
 * PROGRAMA*****
 */
class Logika {
    String poruka = "";
}

```

```

boolean signal;
BazaPodataka bp;

public boolean pamtiSlog(String slog) {
    bp = new BazaPodataka();
    if (!bp.otvoribazu()) {
        poruka = bp.poruka;
        return false;
    }
    if (!bp.pamtiSlog("Student", slog)) {
        poruka = bp.poruka;
        return false;
    }
    if (!bp.zatvoriBazu()) {
        poruka = bp.poruka;
        return false;
    }
    poruka = bp.poruka;
    return true;
}

void postaviPoruku(String pom) {
    poruka = new String(pom);
    signal = false;
}

}

/**
 * *****NIVO BAZE
 * PODATAKA*****
 */
class BazaPodataka {
    Connection con;
    String poruka;

    boolean otvoribazu() {
        String dbUrl = new String();
        String user = "root";
        String pass = "root";
        try {
            dbUrl = "jdbc:mysql://127.0.0.1:3306/student?useUnicode=true&characterEncoding=UTF-8";
            // VAŽNA NAPOMENA: Na ovaj način je omogućeno da se pamte čirilična i
            // latinična
            // slova u MySQL bazi. Na strani baze takodje je potrebno da atributi
            // tabele budu
            // vezani za charset UTF-8.
            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection(dbUrl, user, pass);
        } catch (ClassNotFoundException cnfe) {
            poruka = "Nije ucitan upravljacki program: " + cnfe;
            System.out.println(poruka);
            return false;
        } catch (SQLException sqle) {
            poruka = "Greska kod konekcije: " + sqle;
            System.out.println(poruka);
            return false;
        }
        return true;
    }

    boolean pamtiSlog(String imetabele, String vrednostiAtributa) {
        String upit;
        try {
            Statement st = con.createStatement();
            upit = "INSERT INTO " + imetabele + " VALUES (" + vrednostiAtributa + ")";
            st.executeUpdate(upit);
            poruka = "Slog je uspesno zapamcen u bazi";
            System.out.println(poruka);
            st.close();
        } catch (SQLException esql) {
            poruka = "Nije uspesno zapamcen slog u bazi: " + esql;
            System.out.println(poruka);
            return false;
        }
        return true;
    }
}

```

```

boolean zatvoriBazu() {
    try {
        con.close();
    } catch (Exception e) {
        poruka = "Nije uspesno zatvorena baza:" + e;
        System.out.println(poruka);
        return false;
    }
    return true;
}

```

```

/* Primer BP6: Napisati program koji ce da brise slogove iz
tabele Student. Tabela se cuva u bazi koja je realizovana
u okviru MySQL SUBP. Naziv baze je Student. Program treba da bude
tronivojski, sto znaci da treba razdvojiti nivoe korisnickog interfejsa,
poslovne logike i baze podataka.
*/

```

Наведени задатак је сличан предходном задатку. У наставку ће бити дате само разлике везане за овај програм.

```

// Prezentacioni nivo programa, odnosno klase BP6GUI i Poruka nalaze se u
// datoteci BP6GUI.java

/*********************PREZENTACIONI NIVO PROGRAMA*****/
public class BP6GUI extends JFrame
{
    ...
    // Dugme preko koga se poziva sistemska operacija.
    private JButton BObrisni;

    // Glavni program
    public static void main(String args[])
    { BP6GUI gui = new BP6GUI ();
        gui.show();
    }

    // 1. Konstruktor ekranске forme
    public BP6GUI ()
    { ...
        PostaviDugmeObrisi();
        pack();
    }

    ...
    // 1.6
    // Deklarisanje i kreiranje objekta koji je odgovoran za logiku programa.
    Logika l = new Logika();
    void PostaviDugmeObrisi()
    { BObrisni.setText("Obrisi");
        BObrisni.addActionListener(new ActionListener() {
            ...
            // DOGADJAJ KOJI INICIRA POZIV SISTEMSKE OPERACIJE
            public void actionPerformed(ActionEvent evt)
            { try { /*POZIV SISTEMSKE OPERACIJE*/
                String uslov = "BrInd = '" + PbrojInd.getValue() + "'";
                l.oberisiSlog(uslov);
            } catch(Exception e) {l.postaviPoruku("Nije obrisan slog!");}
                PrikaziPoruku();
                show();
            }
        });
        ...
        getContentPane().add(BObrisni, new AbsoluteConstraints(260, 60, -1, -1));
    }

    class Poruka extends JDialog
    {...}

    // Logicki nivo programa i nivo baze podataka, odnosno klase Logika i BazaPodataka
    // nalaze se u datoteci BP6LogikaiBaza.java
}

```

```
...
*****LOGICKI NIVO PROGRAMA*****
class Logika
{
    String poruka="";
    boolean signal;
    BazaPodataka bp;
    public boolean obrisiSlog(String uslov)
    { bp = new BazaPodataka();

        if (!bp.otvoriBazu()) { poruka = bp.poruka;return false; }
        if (!bp.brisiSlog("Student",uslov)) { poruka = bp.poruka;return false; }
        if(!bp.zatvoriBazu()) {poruka = bp.poruka;return false; }
        poruka = bp.poruka;
        return true;
    }

    void postaviPoruku(String pom)
    { poruka = new String(pom);
        signal = false;
    }
}

*****NIVO BAZE PODATAKA*****
class BazaPodataka
{ Connection con;
    String poruka;

    boolean otvoriBazu()
    {...}

    public boolean brisiSlog(String imeTabele,String uslovZaNadjislog)
    { String upit;
        try { Statement st;
            st = con.createStatement();
            upit ="DELETE FROM " + imeTabele + " WHERE " + uslovZaNadjislog;
            st.executeUpdate(upit);
            poruka = "Slog je uspesno obrisan";
            System.out.println(poruka);
            st.close();
        } catch(SQLException esql) { poruka = "Nije uspesno obrisan slog u bazi: " +
esql;
            System.out.println(poruka); return
false; }
        return true;
    }

    boolean zatvoriBazu() { ... }
}
```

Задатак BPZ2: Омогућити памћење, промену и брисање слогова табеле *Račun* базе *Prodaja* преко стандардног улаза.

Задатак BPZ3: Приказати све слогове табеле *Račun* базе *Prodaja*.

Задатак BPZ4: Омогућити памћење, промену и брисање слогова табеле *Račun* базе *Prodaja* преко *Swing GUI-a*. Програм треба да буде тронивојски.

3.3 Примери SQL упита над MySQL СУБП

3.3.1 Дефиниција података (data definition)

Пример CRDB1: Направити базу података по имениу *racun*. Подразумевани формат података треба да буде **UTF8**.

```
CREATE DATABASE racun
CHARACTER SET UTF8
```

Пример ALDB1: Код базе података racun променити формат податка на Latin1.

```
ALTER DATABASE racun
CHARACTER SET Latin1
```

Пример DRDB1: Обрисати базу података racun1 ако постоји.

```
DROP DATABASE IF EXISTS racun1
```

Пример CRTAB1: Направити табеле:

racun са атрибутима BrojRacuna, NazivPartnera, UkupnaVrednost, Obradjeni и Storniran. Примарни кључ је атрибут BrojRačuna.

stavkaracuna са атрибутима BrojRacuna, RB, SifraProizvoda, Kolicina, ProdajnaCena и ProdajnaVrednost. Примарни кључ је BrojRacuna и RB.

```
CREATE TABLE `racun`.`racun` (
  `BrojRacuna` VARCHAR(10) NOT NULL,
  `NazivPartnera` VARCHAR(50) NOT NULL,
  `UkupnaVrednost` DOUBLE NOT NULL,
  `Obradjeni` ENUM('N','Y') NOT NULL DEFAULT 'N',
  `Storaniran` ENUM('N','Y') NOT NULL DEFAULT 'N',
  PRIMARY KEY(`BrojRacuna`)
)
CREATE TABLE `racun`.`stavkaracuna` (
  `BrojRacuna` VARCHAR(10) NOT NULL,
  `RB` INTEGER NOT NULL,
  `SifraProizvoda` VARCHAR(20) NOT NULL,
  `Kolicina` INTEGER NOT NULL DEFAULT 0,
  `ProdajnaCena` DOUBLE NOT NULL DEFAULT 0,
  `ProdajnaVrednost` DOUBLE NOT NULL DEFAULT 0,
  PRIMARY KEY(`BrojRacuna`, `RB`)
```

Пример RNTAB1: Променити назив табеле stavkaracuna у stavkaracuna1.

```
RENAME TABLE stavkaracuna TO stavkaracuna1
```

Напомена: После промене назива табеле треба вратити стари назив како би могли остали примери да се тестирају:

```
RENAME TABLE stavkaracuna1 TO stavkaracuna
```

Пример ALTAB1: Додајте у табелу stavkaracuna атрибут JedinicaMere који је знаковног типа дужине 4, који ће бити постављен после атрибута SifraProizvoda.

```
ALTER TABLE stavkaracuna
```

```
ADD COLUMN JedinicaMere VARCHAR(4) AFTER SifraProizvoda
```

Пример ALDB2: Обрисати у табели stavkaracuna атрибут SifraProizvoda.

```
ALTER TABLE stavkaracuna
```

```
DROP COLUMN JedinicaMere
```

Пример DRTAB1: Креирати табелу test са атрибутом testatribut и након тога је обрисати.

```
CREATE TABLE `racun`.`test` (
  `testatribut` VARCHAR(10) NOT NULL,
)
DROP TABLE `racun`.`test`
```

Пример CRIND1: Поставити индекс над атрибутом NazivPartnera табеле racun. Прогласити наведени атрибут да буде примарни кључ.

```
CREATE UNIQUE INDEX IndNazPart ON racun(NazivPartnera)
```

Напомена: Индексирањем се повећава брзина приступа до слогова преко атрибута који је индексиран.

Пример DRIND1: Уклонити индекс над атрибутом **NazivPartnera** табеле **racun**.

```
DROP INDEX IndNazPart ON racun
```

6.3.2 Манипулација подацима (data manipulation)

Пример Insert1: Унос три слога рачуна

```
INSERT INTO Racun
VALUES ('1', 'Meridian invest D.O.O', 0, 'N', 'N');
```

```
-----  
INSERT INTO Racun
VALUES ('2', 'Perihard inženjering', 0, 'N', 'N');
```

```
-----  
INSERT INTO Racun
VALUES ('3', 'Sajam', 0, 'N', 'N');
```

```
-----  
INSERT INTO StavkaRacuna
VALUES ('1', '1', 'pr1', 5, 45.00, 0);
```

```
-----  
INSERT INTO StavkaRacuna
VALUES ('1', '2', 'pr2', 1, 45.00, 0);
```

```
-----  
INSERT INTO StavkaRacuna
VALUES ('1', '3', 'pr3', 2, 56.34, 0);
INSERT INTO StavkaRacuna
VALUES ('2', '1', 'pr1', 7, 12.56, 0);
```

```
-----  
INSERT INTO StavkaRacuna
VALUES ('2', '2', 'pr2', 7, 12.00, 0);
```

```
-----  
INSERT INTO StavkaRacuna
VALUES ('3', '1', 'pr1', 3, 45.00, 0);
```

Пример Update1²³: Промени рачуне код којих је назив партнера “Sajam”. Уместо тога ставити “Beogradski sajam”

```
UPDATE Racun SET NazivPartnera = "Beogradski sajam"
WHERE NazivPartnera="Sajam";
```

Пример Update2: Израчунати и запамтити продајне вредности ставки свих рачуна по формули: ProdajnaVrednost = ProdajnaCena * Kolicina
UPDATE StavkaRacuna SET ProdajnaVrednost = ProdajnaCena*Kolicina

Пример Dell1: Обрисати рачун број 3.

```
DELETE
FROM Racun
WHERE BrojRacuna= "3";
```

Пример Sell: Приказивање свих слогова за све атрибуте навођењем имена атрибута -
Приказати све рачуне

²³ Може и овако:

```
UPDATE Racun SET Racun.NazivPartnera = "Beogradski sajam"
WHERE Racun.NazivPartnera="Sajam";
```

SELECT * FROM racun;

| BrojRacuna | NazivPartnera | UkupnaVrednost | Obradjen | Storniran |
|------------|-----------------------|----------------|----------|-----------|
| 1 | Meridian invest D.O.O | 0 | N | N |
| 2 | Perihard inzenjerинг | 0 | N | N |

Пример Sel2: Приказивање свих слогова за све атрибуте преко * - Приказати све рачуне

SELECT * FROM racun

| BrojRacuna | NazivPartnera | UkupnaVrednost | Obradjen | Storniran |
|------------|-----------------------|----------------|----------|-----------|
| 1 | Meridian invest D.O.O | 0 | N | N |
| 2 | Perihard inzenjerинг | 0 | N | N |

Пример Sel3: Приказивање свих слогова за један или неколико атрибути (не свих) навођењем имена атрибути - Приказати бројеве рачуна и називе партнера за све рачуне

SELECT BrojRacuna, NazivPartnera FROM racun;

| BrojRacuna | NazivPartnera |
|------------|-----------------------|
| 1 | Meridian invest D.O.O |
| 2 | Perihard inzenjerинг |

Пример Dist1: Приказивање различитих слогова табеле - Приказати различите бројеве рачуна са ставке рачуна.

SELECT DISTINCT BrojRacuna FROM StavkaRacuna;

| BrojRacuna |
|------------|
| 1 |

Пример Dist2: Приказивање различитих слогова табеле - Приказати различите редне бројеве рачуна.

SELECT DISTINCT rb FROM StavkaRacuna;

| RB |
|----|
| 1 |
| 2 |
| 3 |

Пример Sel4: Приказивање слогова табеле под неким условом - Приказати податке о рачуну чији је број 2.

SELECT * FROM racun WHERE BrojRacuna = "2"

| BrojRacuna | NazivPartnera | UkupnaVrednost | Obradjen | Storniran |
|------------|----------------------|----------------|----------|-----------|
| 2 | Perihard inzenjerинг | 0 | N | N |

Пример OrdBy1: Приказ слогова табеле у сортираном редоследу - Приказати редне бројеве ставки рачуна, у опадајућем редоследу, за рачун чији је број 1.

SELECT * FROM StavkaRacuna WHERE BrojRacuna="1" ORDER BY RB DESC;

| BrojRacuna | RB | SifraProizvoda | Kolicina | ProdajnaCena | ProdajnaVrednost |
|------------|----|----------------|----------|--------------|------------------|
| 1 | 3 | pr3 | 2 | 56.34 | 112.68 |
| 1 | 2 | pr2 | 1 | 45 | 45 |
| 1 | 1 | pr1 | 5 | 45 | 225 |

Пример Max1: Функције за израчунавање сумарних информација - Нађи највећу количину код ставки рачуна чији је број 1.

```
SELECT Max(Kolicina) AS MaxKolicina
FROM StavkaRacuna
WHERE StavkaRacuna.BrojRacuna="1";
```

| MaxKolicina |
|-------------|
| 5 |

Пример Sum1: Функције за израчунавање сумарних информација - Приказати укупну вредност рачуна чији је број 1.

```
SELECT Sum(ProdajnaVrednost) AS UkupnaVrednost
FROM StavkaRacuna
WHERE StavkaRacuna.BrojRacuna="1";
```

| UkupnaVrednost |
|----------------|
| 382.68 |

Пример Count1: Функција Count за рачунање броја слогова у табели - Нађи број ставки рачуна чији је број 1.

```
SELECT Count(*) AS BrojStavki
FROM StavkaRacuna
WHERE BrojRacuna="1";
```

| BrojStavki |
|------------|
| 3 |

Пример Count2: Функција Count за рачунање броја слогова у табели - Нађи број ставки рачуна чији је број 1 по атрибуту ProdajnaCena.

```
SELECT Count(ProdajnaCena) AS BrojStavki
FROM StavkaRacuna
WHERE BrojRacuna="1";
```

| BrojStavki |
|------------|
| 3 |

Пример Count3: Функција Count за рачунање броја слогова у табели - Нађи број различитих ставки рачуна чији је број 1 по атрибуту ProdajnaCena.

```
SELECT Count(Distinct ProdajnaCena) AS BrojStavki
FROM StavkaRacuna
WHERE BrojRacuna="1";
```

| BrojStavki |
|------------|
| 2 |

Računa se број слогова који имају разлиčиту цену (56.34,45.00).

Пример GroupBy1: Груписање слогова табеле по једном или више атрибута - Приказати колико је продато комада за сваку групу (врсту) производа.

Да би се схватио наведени пример потребно је анализирати табелу StavkaRacuna.

| BrojRacuna | RB | SifraProizvoda | Kolicina | ProdajnaCena | ProdajnaVrednost |
|------------|----|----------------|----------|--------------|------------------|
| 1 | 1 | pr1 | 5 | 45 | 225 |
| 1 | 2 | pr2 | 1 | 45 | 45 |
| 1 | 3 | pr3 | 2 | 56.34 | 112.68 |
| 2 | 1 | pr1 | 7 | 12.56 | 87.92 |
| 2 | 2 | pr2 | 7 | 12 | 84 |
| 3 | 1 | pr1 | 3 | 45 | 135 |

```
SELECT SifraProizvoda, Sum(Kolicina) as UkupnaKolicina FROM stavkaracuna
Group By SifraProizvoda
```

| SifraProizvoda | UkupnaKolicina |
|----------------|----------------|
| pr1 | 15 |
| pr2 | 8 |
| pr3 | 2 |

Пример GroupBy2: Груписање слогова табеле по једном или више атрибута - За сваки рачун показати колико је продато производа (сумарно).

```
SELECT BrojRacuna, Sum(Kolicina) as UkupnaKolicina FROM stavkaracuna Group
By BrojRacuna
```

| BrojRacuna | UkupnaKolicina |
|------------|----------------|
| 1 | 8 |
| 2 | 14 |
| 3 | 3 |

Пример GroupBy3: Груписање слогова табеле по једном или више атрибута - За сваки рачун одредити број ставки.

```
SELECT BrojRacuna, Count(*) as BrojStavki FROM stavkaracuna Group By
BrojRacuna
```

| BrojRacuna | BrojStavki |
|------------|------------|
| 1 | 3 |
| 2 | 2 |
| 3 | 1 |

Пример GroupBy4: Груписање слогова табеле по једном или више атрибута - За сваки производ одредити рачуне. Рачуне сортирати по шифри производа у растућем редоследу и количини у опадајућем редоследу.

```
SELECT SifraProizvoda, BrojRacuna, Kolicina
FROM stavkaracuna
GROUP BY SifraProizvoda, BrojRacuna
ORDER BY SifraProizvoda, Kolicina Desc
```

| SifraProizvoda | BrojRacuna | Kolicina |
|----------------|------------|----------|
| pr1 | 2 | 7 |
| pr1 | 1 | 5 |
| pr1 | 3 | 3 |
| pr2 | 2 | 7 |
| pr2 | 1 | 1 |
| pr3 | 1 | 2 |

Пример Having1: Селекција слогова преко Having наредбе - Приказати рачуне који имају број ставки већи од 1.

```
SELECT BrojRacuna, Count(*) as BrojStavki FROM stavkaracuna Group By
BrojRacuna HAVING Count(*) > 1
```

| BrojRacuna | BrojStavki |
|------------|------------|
| 1 | 3 |
| 2 | 2 |

Пример ArFun1: Аритметичке функције - Заокруглити продајну вредност ставки рачуна на 1 децималу.

```
SELECT RB, SifraProizvoda, Kolicina, ProdajnaCena, Round(ProdajnaVrednost,1) as
ProdajnaVrednost FROM stavkaracuna
```

| RB | SifraProizvoda | Kolicina | ProdajnaCena | ProdajnaVrednost |
|----|----------------|----------|--------------|------------------|
| 1 | pr1 | 5 | 45 | 225.0 |
| 2 | pr2 | 1 | 45 | 45.0 |
| 3 | pr3 | 2 | 56.34 | 112.7 |
| 1 | pr1 | 7 | 12.56 | 87.9 |
| 2 | pr2 | 7 | 12 | 84.0 |
| 1 | pr1 | 3 | 45 | 135.0 |

Пример IfNull1: Показати како се користи функција ifnull.

Претпоставимо да треба да се унесе ставка за рачун број 3 за коју се не зна производ који је унет:

```
INSERT INTO StavkaRacuna VALUES ('3', '2', null, 3, 45.00, 0);
```

Претпоставимо да атрибут SifraProizvoda може да добије null vrednost.

Након извршења упита:

```
SELECT BrojRacuna, RB, SifraProizvoda, Kolicina, ProdajnaCena,
ProdajnaVrednost
FROM stavkaracuna
WHERE BrojRacuna = '3'
```

добија се:

| BrojRacuna | RB | SifraProizvoda | Kolicina | ProdajnaCena | ProdajnaVrednost |
|------------|----|----------------|----------|--------------|------------------|
| 3 | 1 | pr1 | 3 | 45 | 135 |
| 3 | 2 | null | 4 | 67.45 | 0 |

Уколико желимо да код извештаја уместо null вредности ставимо нпр. вредност “nepoznata” писаћемо:

```
SELECT BrojRacuna, RB, Ifnull(SifraProizvoda, "nepoznata"),
Kolicina, ProdajnaCena,
ProdajnaVrednost FROM stavkaracuna
WHERE BrojRacuna = '3'
```

| BrojRacuna | RB | SifraProizvoda | Kolicina | ProdajnaCena | ProdajnaVrednost |
|------------|----|----------------|----------|--------------|------------------|
| 3 | 1 | pr1 | 3 | 45 | 135 |
| 3 | 2 | nepoznata | 4 | 67.45 | 0 |

Пример ArFun2: Функција за рад са стринговима - За сваки рачун приказати број знакова назива партнера.

```
Select NazivPartnera, Length(NazivPartnera) as DuzinaNaziva From Racun
```

| NazivPartnera | DuzinaNaziva |
|-----------------------|--------------|
| Meridian invest D.O.O | 21 |
| Perihard inzenjering | 20 |
| Sajam | 5 |

Пример Ug1: Угњеждени упити (subquery) - Приказати рачуне код којих је продато више од 7 производа.

Пре него што се прикаже решење овога примера погледаћемо колико је по сваком рачуну продато производа:

```
Select BrojRacuna, Sum(Kolicina) as UkupnaKolicina FROM stavkaracuna Group
By BrojRacuna
```

| BrojRacuna | UkupnaKolicina |
|------------|----------------|
| 1 | 8 |
| 2 | 14 |
| 3 | 7 |

Решење примера Ug1:

```
Select BrojRacuna, suma From (Select BrojRacuna, Sum(Kolicina) as suma From stavkaracuna Group By BrojRacuna) as sume Where sume>7
```

| BrojRacuna | suma |
|------------|------|
| 1 | 8 |
| 2 | 14 |

Пример Ug2: Угњеждени упити (subquery) - Приказати ставке рачуна које су направљене за *Perihard inzenjering*.

```
Select * From stavkaracuna Where BrojRacuna = (Select BrojRacuna from Racun Where NazivPartnera = "Perihard inzenjering")
```

| BrojRacuna | RB | SifraProizvoda | Kolicina | ProdajnaCena | ProdajnaVrednost |
|------------|----|----------------|----------|--------------|------------------|
| 2 | 1 | pr1 | 7 | 12.56 | 87.92 |
| 2 | 2 | pr2 | 7 | 12 | 84 |

Правило: Унутрашњи упит мора да врати само једну вредност ако се у Where клаузули користи оператори поређења ($>=,<=,=,<,>$).

Уколико би направили још један рачун за *Perihard inzenjering*:

```
INSERT INTO Racun VALUES ('4', 'Perihard inženjering', 0, 'N', 'N');  
са једном ставком:
```

```
INSERT INTO StavkaRacuna VALUES ('4', '1','pr1',5,71.68,0);
```

тада би се при извршењу упита:

```
Select * From stavkaracuna Where BrojRacuna = (Select BrojRacuna from Racun Where NazivPartnera = "Perihard inzenjering")
```

јавила порука: *Subquery returns more than 1 row* (Подупит враћа више од 1 реда)

Тада се користи **IN** наредба уместо оператора **=**

```
Select * From stavkaracuna Where BrojRacuna IN (Select BrojRacuna from Racun Where NazivPartnera = "Perihard inženjering")
```

| BrojRacuna | RB | SifraProizvoda | Kolicina | ProdajnaCena | ProdajnaVrednost |
|------------|----|----------------|----------|--------------|------------------|
| 2 | 1 | pr1 | 7 | 12.56 | 87.92 |
| 2 | 2 | pr2 | 7 | 12 | 84 |
| 4 | 1 | pr1 | 5 | 71.68 | 0 |

Пример Ug3: Угњеждени упити (subquery) - Приказати рачун код кога је продато највише производа.

```
Select BrojRacuna,suma From (Select BrojRacuna, Sum(Kolicina) as suma From stavkaracuna Group By BrojRacuna) as sume Where suma = (Select Max(suma) From (Select BrojRacuna,sum(Kolicina) as suma From stavkaracuna Group By BrojRacuna) as sume)
```

| BrojRacuna | suma |
|------------|------|
| 2 | 14 |

Друго решење које је урађено преко погледа:

a) Прво се направи поглед

```
Create View Sume as (Select BrojRacuna, Sum(Kolicina) as suma From stavkaracuna GroupBy BrojRacuna)
```

б) Над погледом се направи упит

```
Select BrojRacuna,suma From Sume Where suma = (Select Max(suma) From Sume)
```

Поглед се може преко *MySQL* SUBP-а запамтити.

Пример Uni1: Унија две табеле - Приказати све ставке рачуна 1 и 2 преко уније.

```
SELECT * FROM stavkaracuna WHERE BrojRacuna = '1'
UNION
SELECT * FROM stavkaracuna WHERE BrojRacuna = '2'
```

| BrojRacuna | RB | SifraProizvoda | Kolicina | ProdajnaCena | ProdajnaVrednost |
|------------|----|----------------|----------|--------------|------------------|
| 1 | 1 | pr1 | 5 | 45 | 225 |
| 1 | 2 | pr2 | 1 | 45 | 45 |
| 1 | 3 | pr3 | 2 | 56.34 | 112.68 |
| 2 | 1 | pr1 | 7 | 12.56 | 87.92 |
| 2 | 2 | pr2 | 7 | 12 | 84 |

Пример Raz1: Разлика - Приказати рачуне свих пословних партнера осим *Perihard inzenjeringu*.

```
SELECT * FROM racun WHERE NazivPartnera != "Perihard inzenjeringu"
```

| BrojRacuna | NazivPartnera | UkupnaVrednost | Obradjen | Storniran |
|------------|-----------------------|----------------|----------|-----------|
| 1 | Meridian invest D.O.O | 0 | N | N |
| 3 | Beogradski sajam | 0 | N | N |

Пример Equijoin1: Спајање две или више табела - Спајање на једнакост (*equijoin*) - Приказати за сваку ставку рачуна име партнера за који је везана ставка.

```
SELECT nazivpartnera, stavkaracuna.* FROM racun, stavkaracuna WHERE racun.BrojRacuna = stavkaracuna.BrojRacuna
```

| Nazivpartnera | BrojRacuna | RB | SifraProizvoda | Kolicina | ProdajnaCena | ProdajnaVrednost |
|-----------------------|------------|----|----------------|----------|--------------|------------------|
| Meridian invest D.O.O | 1 | 1 | pr1 | 5 | 45 | 225 |
| Meridian invest D.O.O | 1 | 2 | pr2 | 1 | 45 | 45 |
| Meridian invest D.O.O | 1 | 3 | pr3 | 2 | 56.34 | 112.68 |
| Perihard inzenjeringu | 2 | 1 | pr1 | 7 | 12.56 | 87.92 |
| Perihard inzenjeringu | 2 | 2 | pr2 | 7 | 12 | 84 |
| Beogradski sajam | 3 | 1 | pr1 | 3 | 45 | 135 |
| Beogradski sajam | 3 | 2 | | 4 | 67.45 | 0 |
| Perihard inženjeringu | 4 | 1 | pr1 | 5 | 71.68 | 0 |

Пример CarJoin1: Спајање две или више табела - **Декартов производ (cartesian join)**- Приказати све комбинације слогова табела Racun i StavkaRacuna (из предходног примера смо изоставили Where клаузулу).

```
SELECT nazivpartnera, stavkaracuna.BrojRacuna, stavkaRacuna.RB FROM racun,
stavkaracuna
```

| nazivpartnera | BrojRacuna | RB |
|-----------------------|------------|----|
| Meridian invest D.O.O | 1 | 1 |
| Perihard inzenjeringu | 1 | 1 |
| Beogradski sajam | 1 | 1 |
| Perihard inženjeringu | 1 | 1 |
| Meridian invest D.O.O | 1 | 2 |
| Perihard inzenjeringu | 1 | 2 |
| Beogradski sajam | 1 | 2 |
| Perihard inženjeringu | 1 | 2 |
| Meridian invest D.O.O | 1 | 3 |
| Perihard inzenjeringu | 1 | 3 |
| Beogradski sajam | 1 | 3 |
| Perihard inženjeringu | 1 | 3 |

| | | |
|-----------------------|---|---|
| Meridian invest D.O.O | 2 | 1 |
| Perihard inzenjering | 2 | 1 |
| Beogradski sajam | 2 | 1 |
| Perihard inženjering | 2 | 1 |
| Meridian invest D.O.O | 2 | 2 |
| Perihard inzenjering | 2 | 2 |
| Beogradski sajam | 2 | 2 |
| Perihard inženjering | 2 | 2 |
| Meridian invest D.O.O | 3 | 1 |
| Perihard inzenjering | 3 | 1 |
| Beogradski sajam | 3 | 1 |
| Perihard inženjering | 3 | 1 |
| Meridian invest D.O.O | 3 | 2 |
| Perihard inzenjering | 3 | 2 |
| Beogradski sajam | 3 | 2 |
| Perihard inženjering | 3 | 2 |
| Meridian invest D.O.O | 4 | 1 |
| Perihard inzenjering | 4 | 1 |
| Beogradski sajam | 4 | 1 |
| Perihard inženjering | 4 | 1 |

За 4 рачуна и 8 ставки рачуна има укупно 32 комбинације (парова) рачун-ставка рачуна.

Пример LeftJoin1: Спајање две или више табела - Спљоно спајање (*outer join*) – лево спајање (*left join*) - Приказати све шифре производа и ставке рачуна које су повезане са тим производима.

Креираћемо табелу производ:

```
CREATE TABLE `racun`.`proizvod` (
  `SifraProizvoda` VARCHAR(20) NOT NULL,
  PRIMARY KEY(`SifraProizvoda`)
)
```

Унећемо неколико производа.

```
INSERT INTO proizvod VALUES ('pr1');
INSERT INTO proizvod VALUES ('pr2');
INSERT INTO proizvod VALUES ('pr3');
INSERT INTO proizvod VALUES ('pr4');
INSERT INTO proizvod VALUES ('pr5');
```

Након тога ће се извршити упит:

```
SELECT Proizvod.SifraProizvoda, StavkaRacuna.BrojRacuna, StavkaRacuna.RB,
StavkaRacuna.SifraProizvoda
FROM Proizvod LEFT JOIN StavkaRacuna ON Proizvod.SifraProizvoda =
StavkaRacuna.SifraProizvoda;
```

| SifraProizvoda | BrojRacuna | RB | SifraProizvoda |
|----------------|------------|----|----------------|
| pr1 | 1 | 1 | pr1 |
| pr1 | 2 | 1 | pr1 |
| pr1 | 3 | 1 | pr1 |
| pr1 | 4 | 1 | pr1 |
| pr2 | 1 | 2 | pr2 |
| pr2 | 2 | 2 | pr2 |
| pr3 | 1 | 3 | pr3 |
| pr4 | | | |
| pr5 | | | |

Правило: Приказују се сви слогови са леве (*LEFT*) стране од *JOIN* (*Proizvod*) и само они слогови са десне од *JOIN* (*StavkaRacuna*) који задовољавају услов *Proizvod.SifraProizvoda = StavkaRacuna.SifraProizvoda*

Пример RightJoin1: Спајање две или више табела - Спљоно спајање (*outer join*) – десно спајање (*right join*) - Приказати све ставке рачуна независно од тога да ли су везане за постојеће шифре производа

```
SELECT Proizvod.SifraProizvoda, StavkaRacuna.BrojRacuna, StavkaRacuna.RB
FROM Proizvod RIGHT JOIN StavkaRacuna ON Proizvod.SifraProizvoda =
StavkaRacuna.SifraProizvoda;
```

| SifraProizvoda | BrojRacuna | RB |
|----------------|------------|----|
| pr1 | 1 | 1 |
| pr2 | 1 | 2 |
| pr3 | 1 | 3 |
| pr1 | 2 | 1 |
| pr2 | 2 | 2 |
| pr1 | 3 | 1 |
| | 3 | 2 |
| pr1 | 4 | 1 |

Правило: Приказују се сви слогови са десне (RIGHT) стране од JOIN (StavkaRacuna) и само они слогови са леве стране од JOIN (Racun) који задовољавају услов Proizvod.SifraProizvoda = StavkaRacuna.SifraProizvoda

Пример SelfJoin1: Спајање две или више табела - Спомоно спајање (*outer join*) – Само спајање (*selfjoin*) - Приказати хијерархију производа почев од производа *pr1* који је на врху.

Проширићемо табелу производ са допунским атрибутом *Nadredjeni*.

```
ALTER TABLE `racun`.``proizvod` MODIFY COLUMN `SifraProizvoda` VARCHAR(20)
NOT NULL, ADD COLUMN `Nadredjeni` VARCHAR(20) NOT NULL;
```

Направићемо следећу хијерархијску саставницу (1 подређени производ улази у 1 надређени производ; у један надређени производ може да уђе више подређених производа). Производ *pr1* се састоји из производа *pr2* и *pr3*. Производ *pr2* се састоји од производа *pr4* и *pr5*.

```
UPDATE proizvod SET Nadredjeni = 'pr1' Where SifraProizvoda = 'pr2';
UPDATE proizvod SET Nadredjeni = 'pr1' Where SifraProizvoda = 'pr3';
UPDATE proizvod SET Nadredjeni = 'pr2' Where SifraProizvoda = 'pr4';
UPDATE proizvod SET Nadredjeni = 'pr2' Where SifraProizvoda = 'pr5';
UPDATE proizvod SET Nadredjeni = 'vrh' Where SifraProizvoda = 'pr1';
```

На крају ћемо извршити упит:

```
SELECT p.SifraProizvoda,p.Nadredjeni,s.SifraProizvoda as Podredjeni FROM
proizvod p, proizvod s Where p.SifraProizvoda = s.Nadredjeni
```

| SifraProizvoda | Nadredjeni | Podredjeni |
|----------------|------------|------------|
| pr1 | vrh | pr2 |
| pr1 | vrh | pr3 |
| pr2 | pr1 | pr4 |
| pr2 | pr1 | pr5 |

Објашњење: Табеле *p* и *s* се спајају преко *p.SifraProizvoda = s.Nadredjeni* на следећи начин:

Табела *p*

| SifraProizvoda | Nadredjeni |
|----------------|------------|
| pr1 | vrh |
| pr2 | pr1 |
| pr3 | pr1 |
| pr4 | pr2 |
| pr5 | pr2 |

Tabela s

| Sifra Proizvoda (ovo je Podredjeni) | Nadredjeni |
|-------------------------------------|------------|
| pr1 | vrh |
| pr2 | pr1 |
| pr3 | pr1 |
| pr4 | pr2 |
| pr5 | pr2 |

4.РЕФЕРЕНЦЕ

[J2EE14TUT] : *The J2EE 1.4 Tutorial*, Sun Microsystems, August 31, 2004

[R2D1J2EEKOMP] : *Java Technology Concept Map*, Sun Microsystems,
<http://java.sun.com/developer/onlineTraining/new2java/javamap/intro.html>

[WikiJavaPlatform] : http://en.wikipedia.org/wiki/Java_platform

[JJU] JavaJazzUp magazine: <http://www.javajazzup.com/>, issue 1.

[IJW] Introducing JAX-WS 2.0 With the Java SE 6 Platform:
http://java.sun.com/developer/technicalArticles/J2SE/jax_ws_2/

[DCTJ] David Chappell, Tyler Jewell: *Java Web Services*, O'Reilly, 2004, Sebastopol, CA, USA

[W3S] W3Schools Online Web Tutorials: <http://www.w3schools.com/>

[IvHo] Ivor Horton: *Ivor Horton's Beginning Java 2, JDK 5 Edition*, Wiley publishing, 2004, Indianapolis, Indiana, USA

[JoZu] John Zukowski: *Java 6 Platform Revealed*, Apress, 2006, Berkeley, CA, USA

[GrHa] Graham Hamilton (editor): *JavaBeans™ API specification*, Sun Microsystems, 1997, Mountain View, CA, USA

[Wiki] Wikipedia, слободна енциклопедија: <http://en.wikipedia.org/>

III ДЕО – ПРОЈЕКТОВАЊЕ СОФТВЕРА

Синиша Влајић

САДРЖАЈ

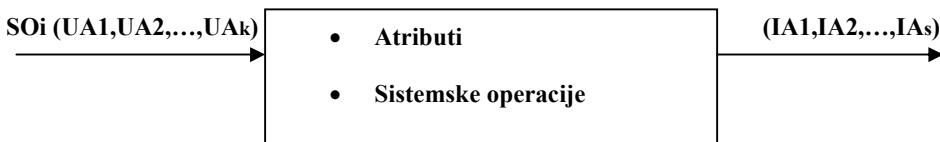
| | |
|---|----|
| 1. Основе пројектовања софтвера | 3 |
| 1.1 Софтверски систем | 3 |
| 1.2 Контекст пројектовања софтвера | 4 |
| 1.2.1 Развој (животни циклус) софтверског система | 4 |
| 1.2.2 Однос информационог и софтверског система | 6 |
| 1.3 Дефиниција пројектовања софтвера | 8 |
| 2. Развој софтверског система | 9 |
| 2.1 Прикупљања захтева од корисника | 9 |
| 2.1.1 Захтеви (Requirements) | 9 |
| 2.1.2 Опис захтева помоћу модела случаја коришћења | 10 |
| 2.2 Анализа | 15 |
| 2.2.1 ПОНАШАЊЕ СОФТ. СИСТЕМА - Сист. дијаграми секвенци | 15 |
| 2.2.2 Дефинисање уговора о системским операцијама | 22 |
| 2.2.3 ограничења при извршењу системских операција | 23 |
| 2.2.4 СТРУКТУРА СОФТ. СИСТЕМА - Концептуални (доменски) модел | 24 |
| 2.2.5 СТРУКТУРА СОФТ. СИСТЕМА - Релациони модел | 26 |
| 2.3 Пројектовања | 27 |
| 2.3.1 Пројектовање корисничког интерфејса | 29 |
| 2.3.1.1 Пројектовање екранске форме | 29 |
| 2.3.1.2 Пројектовање контролера корисничког интерфејса | 40 |
| 2.3.2 Пројектовање апликационе логике | 43 |
| 2.3.2.1 Контролер апликационе логике | 43 |
| 2.3.2.2 Пословна логика | 44 |
| 2.3.2.2.1 Пројектовање понашања софтверског система – системске операције | 44 |
| 2.3.2.2.2 Пројектовање структуре софтверског система | 50 |
| 2.3.2.3 Брокер базе података | 50 |
| 2.3.3 Пројектовање складишта података | 60 |

1. Основе пројектовања софтвера

1.1 Софтверски систем¹

Софтверски систем² се састоји од **атрибута** и **системских операција**³. Атрибути описују **структурну** систем, док системске операције описују **понашање** система. Атрибути представљају концепте реалног система који описују статичке карактеристике система (нпр. *Račun*, *Partner*, ...). Системске операције представљају **основне** (атомске) **функције** система, које се могу користити из окружења система (нпр. *UnesiRačun*, *PromeniRačun*, *IzracunajRacun*, ...). **Допустиви улаз** у софтверски систем је дефинисан потписом (сигнатуром), који садржи назив системске операције која се позива и скупом улазних аргумента (нпр. *IzracunajRacun(Racun)*, ...). **Излаз** из софтверског система је представљен преко скупа излазних аргумента (нпр. *Racun*, *signal*, ...). Излаз се добија као резултат извршења неке од системских операција над атрибутима система.

СОФТВЕРСКИ СИСТЕМ



$$\text{Atributi} = \{\text{AT1}, \text{AT2}, \dots, \text{ATn}\}$$

$$\text{Sistemske operacije} = \{\text{SO1}, \text{SO2}, \dots, \text{SOm}\} - \text{Основне функције система}$$

$$\text{SOi} \in \text{Sistemske operacije}, i = (1, \dots, m)$$

UA1, UA2, ..., UAk – Улазни аргументи

IA1, IA2, ..., IAs – Излазни аргументи

¹ Радећи на овом материјалу, често сам долазио у наизглед нерешиве ситуације код повезивање спецификације проблема са њеном реализацијом (имплементацијом). Оно што је правило највећи проблем јесте схватање шта је софтверски систем и које су његове границе. Питање које ми је задавало највише главобоље односило се на кориснички интерфејс: *Да ли је кориснички интерфејс део софтверског система или је он изван система.*

Одговор на наведено питање, и на многа друга око схватања шаренила многих појмова и термина који се користе у софтверском инжењерству, дао ми је проф. Лазаревић Бранислав. Он ми је рекао да је кориснички интерфејс реализација улаза и/или излаза софтверског система. У првом тренутку то ми је изгледало нелогично, али након неког времена, јасна и прецизна аргументација проф. Лазаревића ме је уверила у исправност његових ставова. Наведени став смо представили преко следеће дефиниције:

Деф. Лаз1: Кориснички интерфејс представља реализацију улаза и/или излаза софтверског система.

Наведена дефиниција, колико год једноставно изгледала, је отворила пут да софтверски систем схватимо као систем у правом смислу дефиниције система[PB]. Надам се да ће системски приступ у схватању развоја софтвера, коначно од софтверског инжењерства направити нешто што ће бити вредно научног а не само технолошког поштовања.

² У даљем тексту систем.

³ ППРС користи термин системска операција када жели да нагласи разлику између операција софтверског система и операција које се налазе изван софтверског система.

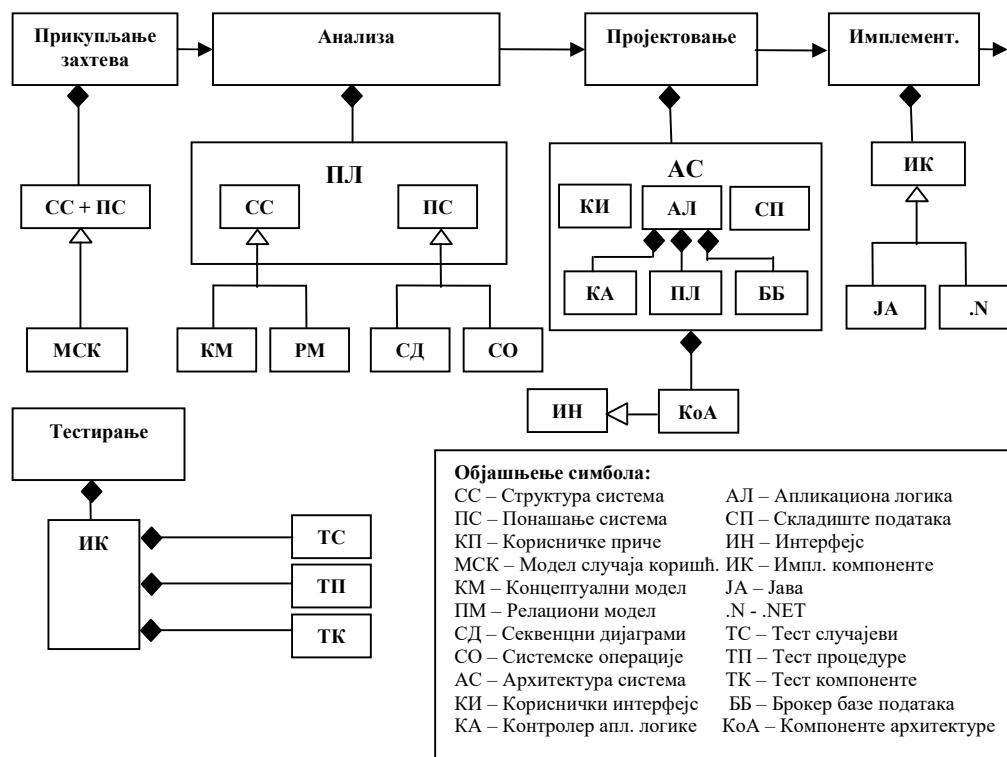
1.2 Контекст пројектовања софтвера

1.2.1 Развој (животни циклус) софтверског система

Развој (животни циклус) софтверског система⁴ се састоји из следећих фаза (Слика RASS):

- Прикупљање захтева од корисника
- Анализе
- Пројектовања
- Имплементације
- Тестирања

У фази прикупљања захтева се дефинишу својства и услови које софтверски систем или шире гледајући пројекат треба да задовољи [Larman]. Захтеви се могу поделити као функционални и нефункционални. Функционални захтеви дефинишу захтеване функције система, док нефункционални захтеви дефинишу све остале захтеве. Нефункционални захтеви као што су *употребљивост, поузданост, перформанс и подржливост система* представљају атрибуте квалитета (*quality attributes*) софтверског система. Функционални захтеви се описују преко модела случаја коришћења. Наведени модел садржи елементе структуре и понашања софтверског система. Елементи структуре софтверског система се описују помоћу *именица* и *придева* док се елементи понашања описују помоћу *глагола*. У фази анализе се описује логичка структура и понашање софтверског система односно *пословна логика* софтверског система.



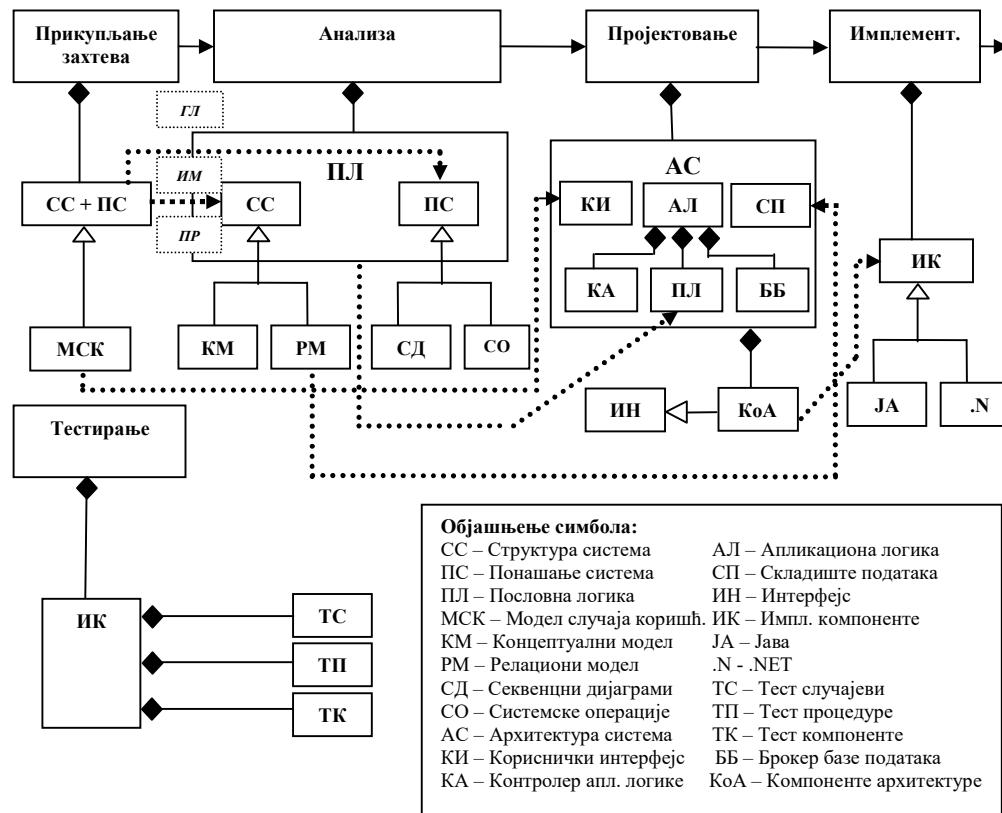
Слика RASS: Развој софтверског система

Структура софтверског система се описује преко *концептуалног (доменског) и релационог модела* док се понашање софтверског система описује помоћу *секвенчних дијаграма и системских операција*. У фази пројектовања се описује *архитектура* софтверског система која је *tronivojska* и састоји се од: а) *корисничког интерфејса* б) *апликационе логике* и ц) *складишта података*. Апликациона логика се састоји од: б1) *контролера апликационе логике* б2) *пословне логике* и б3) *брокера базе података*. Сваки од наведених нивоа тронивојске архитектуре је реализован преко скупа *класа* које представљају *компоненте архитектуре*

⁴ Развој софтверског система је објашњен помоћу поједностављене Ларманове методе развоја софтвера која се користи на ФОН-у, у извођењу наставе на предмету Пројектовања програма и Пројектовање софтвера, од 2003. године.

софтверског система. Компоненте архитектуре су дефинисане преко *интерфејса*. У фази имплементације се праве *имплементационе компоненте* које се реализују у некој од постојећих технологија (*Java*, *.NET*,...). У фази тестирања, свака од компоненти се тестира тако што се за сваку од њих праве *тест случајеви*, *тест процедуре* и *тест компоненте*.

Развој софтверског система код Ларманове методе има јасан логички след (*Слика MERSS*). У фази прикупљања захтева елементи структуре и понашања софтверског система се налазе заједно у случајевима коришћења и они су представљени преко именица, глагола и придева. Структура софтверског система (концепти и атрибути концепата), из фазе анализе, се добија на основу именица и придева који су дефинисани у случајевима коришћења. Системске операције, из фазе анализе, се добијају на основу глагола који су дефинисани у случајевима коришћења. Секвенцни дијаграми, из фазе анализе, се добијају на основу сценарија случајева коришћења.



Слика MERSS: Међувезависност елемената у развоју софтверског система

У фази пројектовања се прави архитектура софтверског система која је тронивојска (кориснички интерфејс, апликациона логика и складиште података). Кориснички интерфејс је дефинисан преко скупа екранских форми. Сценарија коришћења екранских форми је директно повезан са сценаријима случајева коришћења⁵. Пословна логика, која се добија у фази анализе, се преноси у фазу пројектовања и постаје саставни део апликационе логике. Складиште података се пројектује на основу релационог модела. Имплементационе компоненте, из фазе имплементације, треба да реализују компоненте које су добијене у фази пројектовања. Свака од имплементационих компоненти се тестира у фази тестирања.

⁵ Сценарија коришћења екранских форми у суштини представља упутство за крајњег корисника како се користи програм (корисничко упутство).

1.2.2 Однос информационог и софтверског система

Информациони систем (ИС) се прави како би се олакшао рад и управљање неким реалним пословним системом. Пословни систем у најопштијем смислу има своју *структурну и понашање*. Структура пословног система се односи на *организациону структуру* преко које се одређују везе и односи између елемената *пословног система*, док се понашање пословног система односи на *пословне процесе* који одређују токове извршења функција пословног система (нпр. функција продаје, набавке,...).

Структура и понашање пословног система се моделирају преко *модела података* и *модела процеса ИС-а*.

Развој ИС-а подразумева прављење модела процеса и модела података пословног система, који требају да буду реализовани у неком технолошком окружењу (оперативни систем, систем за управљање базом података, програмски језик,...).

Модел процеса се описује помоћу *Структурне Систем Анализе (CCA)*⁶. Структурна систем анализа је представљена преко *дијаграма токова података (ДТП)*. На основу дијаграма тока података могуће је направити *речник података* и дати прецизну спецификацију основних (примитивних) процеса система.

Напомена: Структурна систем анализа се ради у фази анализе ИС-а.

Прва фаза у развоју софтверског система, прикупљање захтева, се описује преко *модела случаја коришћења* (СК) који се добија на основу спецификације основних процеса.

У фази анализе софтверског система, на основу модела СК се одређује *модел података* (структуре) и *модел понашања* софтверског система. Модел података софтверског система се описује помоћу *проширеног модела објекти везе (ПМОВ)*, *релационог модела (РМ)*, *објектног модела (ОМ)*,..., итд. Релациони модел је подржан *SQL* упитним језиком. Релациони модел се може добити на основу ПМОВ-а. Модел понашања софтверског система се описује помоћу *системских операција*. Модел података и модел понашања софтверског система описују *пословну логику* софтверског система.

Напомена: Фаза прикупљања захтева и анализа софтверског система се из перспективе ИС-а називају фаза логичког пројектовања ИС-а.

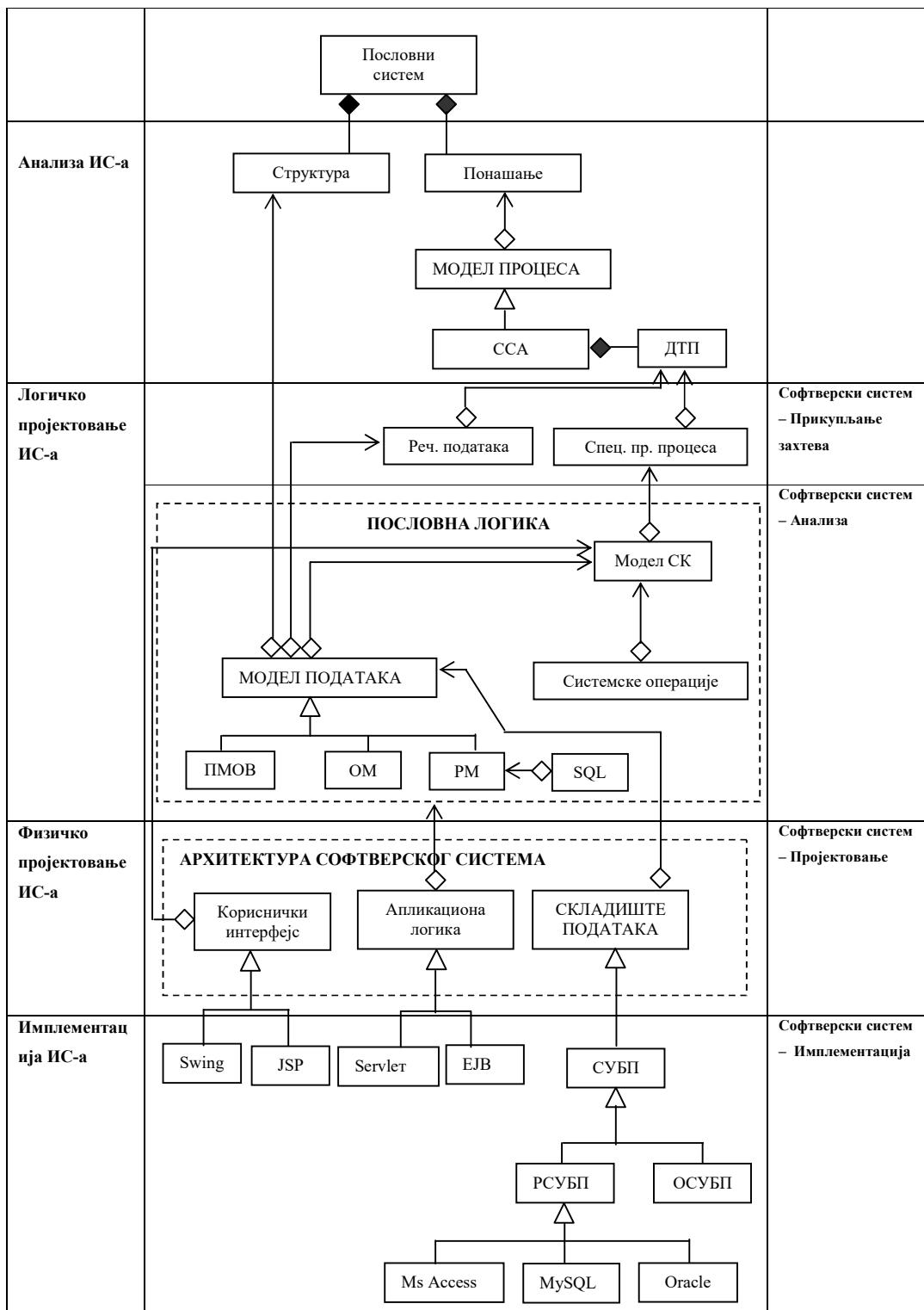
У фази пројектовања софтверског система се дефинише *tronivojska архитектура* која се састоји од: *корисничког интерфејса, апликационе логике и складишта података*.

Пројектовање сценарија коришћења *екранских форми* корисничког интерфејса се ради на основу СК-а. Кориснички интерфејс се може имплементирати коришћењем разних технологија (нпр. *Swing, JSP*,...). Апликациона логика се прави на основу пословне логике софтверског система. Апликациона логика се може реализовати различитим технологијама (*Servlet, EJB*,...).

У складишту података се чувају подаци. Складиште података може бити реализовано преко система за управљање базом података, системом датотека,...,итд. Системи за управљање базом података (СУБП) могу бити: релациони (РСУБП), објектни (ОСУБП),...,итд. Релациони СУБП су: *MS Access, MySQL, Oracle*,..., итд.

Напомена: Фаза пројектовања софтверског система се из перспективе ИС-а назива фаза физичког пројектовања ИС-а. Фаза имплементације софтверског система се из перспективе ИС-а назива фаза имплементације ИС-а.

⁶ Постоје и други модели процеса који овде неће бити разматрани.

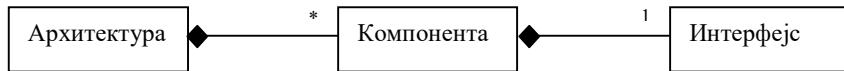


Слика ОИСС: Однос информационог и софтверског система

1.3 Дефиниција пројектовања софтвера

Пројектовање се у контексту софтверског инжењерства дефинише као:

- a) процес дефинисања архитектуре, компоненти, интерфејса и других особина система или компоненте и
- b) резултат тог процеса.



Код објашњења процеса животног циклуса софтвера пројектовање софтвера се састоји од две активности:

- **Пројектовање софтверске архитектуре** (понекад се назива пројектовање највишег нивоа), описује структуру и организацију софтвера на највишем нивоу. На овом нивоу се идентификује различите компоненте.
- **Детаљно пројектовање софтвера** које описује сваку компоненту до нивоа детаљности који је довољан да се она може конструисати.

2. Развој софтверског система

2.1 Прикупљања захтева од корисника

2.1.1 Захтеви (*Requirements*)

Захтеви представљају својства и услове које систем или шире гледајући пројекат мора да задовољи [Ларман]. Постоје различити типови захтева које систем мора да задовољи и они су категоризовани према FURPS+ (**F**unctional - Функционалност, **U**sability - Употребљивост, **R**eliability - Поузданост, **P**erformance - Перформанс, **S**upportability - Подрживост) моделу:

- **Функционалност** представља способност (**capabilities**) система да обезбеди захтеване функције (понашање система). Защита (**security**) система представља једну од основних функција коју систем треба да обезбеди.
- **Употребљивост** представља способност система да се може једноставно користити. То се постиже помоћу разних упутстава и документације који описују начин његовог коришћења.
- **Поузданост** представља способност система да може успешно обрадити проблем (**failure**) који се дешава у току извршења система. У том смислу систем мора да обезбеди начин оправка (**recoverability**) података у случају насиљног прекида рада система. Такође систем треба да омогући предвиђање (**predictability**) могућих понашања система.
- **Перформанс** система се односе на време одзива (**response time**) захтеваних функција, пропусну моћ (**throughput**) мреже кроз коју пролазе подаци, тачност (**accuracy**) извршења функција, могућност коришћења односно расположивост (**availability**) функција система и начин коришћење расположивих ресурса(**resource usage**) система.
- **Подрживост** система се односи на лакоћу његовог прилагођавања (**adaptability**) и одржавања (**Maintainability**), интернационализацију (**internationalization**) у смислу његове прилагодљивости различитим знаковним системима који се користе у свету и начину конфигурисања (**configurability**) система.

Захтеви се често категоризују као функционални и не функционални захтеви. Функционални захтеви дефинишу захтеване функције система, док не функционални захтеви дефинишу све остale захтеве. У том смислу не функционални захтеви (употребљивост, поузданост, перформанс и подрживост система) представљају атрибуте квалитета (**quality attributes**) софтверског система.

У ФУРПС+ моделу знак ‘+’ указује на помоћне захтеве који се односе на:

- **Имплементацију (Implementation)** система – до којих граница се могу користити расположиви ресурси (**resource limitations**). Који се програмски језици (**programming languages**) и алати (**tools**) могу користити. Поред тога имплементациони захтеви се односе и на хардвер (**hardware**) који ће се користити.
- **Интерфејс (Interface)** система – ограничења која постоје у комуникацији система са његовим окружењем (екстерним системима).
- **Операције (Operations)** система – управљање системом и његовим операцијама.
- **Паковање (Packaging)** система – начин физичког организовања система у пакете, који представљају управљиве јединице система.
- **Легалност (Legal)** – могућност употребе система у смислу његове легалности (лиценце и права коришћења система).

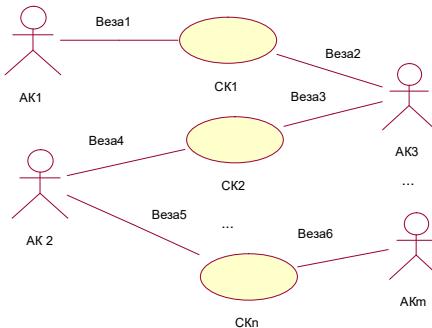
У даљем тексту ми ћемо главни нагласак ставити на разматрање функционалних захтева, од њиховог прикупљања до имплементације. Сматрамо да истовремено објашњење функционалних и не-функционалних захтева, које прати студијски пример, може доста да усложи схватање суштине развоја једног софтверског система из угла његове основне функционалности. У студијском примеру ми ћемо увести неки од не-функционалних захтева који су директно

повезани са функционалношћу система (*поузданост и подрживост система*) и неке од помоћних захтева (*имплементација, операције и паковање система*). Остали не-функционални и помоћни захтеви неће бити разматрани, будући да су они у највећој мери ортогонално постављени у односу на функције система. То значи да они не утичу на схватање и објашњење развоја функција система.

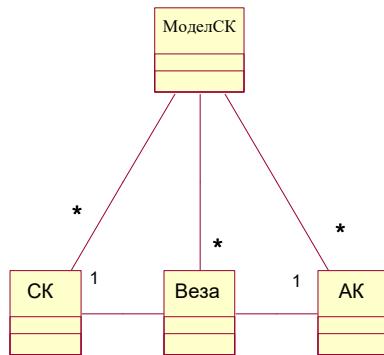
2.1.2 Опис захтева помоћу модела случаја коришћења

Захтеви се код Лармана описују помоћу UML Модела Случаја Коришћења (Use-Case Model).

Деф. ЗА1: Модел СК се састоји од скупа случаја коришћења (СК), актора (АК) и веза између случаја коришћења и актора.



Модел СК се може представити са:



Модел СК је везан за више а) СК, б) веза између СК и АК и ц) АК. Један СК може да има више веза са АК. Један АК може да има више веза са СК. Једна веза се односи на један пар СК-АК.

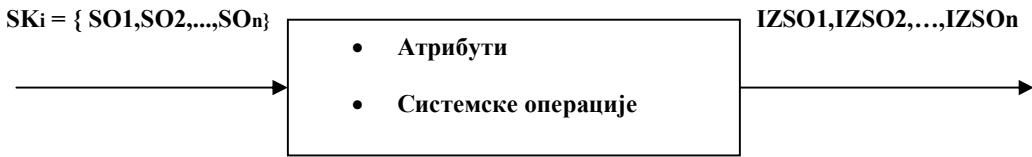
Деф. ЗА2: Случај коришћења описује скуп сценарија (use-case појављивања), односно скуп жељених коришћења система од стране актора.



Сценарио описује једно жељено коришћење система од стране актора. Случај коришћења има један главни и више алтернативних сценарија. Сценарио је описан преко: а) секвенце акција и б) интеракција између актора и система. СК се састоји из главног и алтернативних сценарија.

У току интеракције између актора и софтверског система, актор позива системске операција софтверског система. То значи да се у току неког (једног) сценарија Sk_i случаја коришћења позива једна или више системских операција (SO_1, SO_2, \dots, SO_n).

СОФТВЕРСКИ СИСТЕМ



Као резултат извршења системске операције над атрибутима софтверског система се добијају излазни резултати $IZSO_1, IZSO_2, \dots, IZSO_n$.

Системске операције = $\{SO_1, SO_2, \dots, SO_n\}$ - Основне функције система (атомске функције)

Случајеви коришћења = $\{SK_1, SK_2, \dots, SK_p\}$ - Жељене функције система (молекулске функције)

$SK_i \in \text{Случајеви коришћења}, i = (1..p)$

СК, односно сценарија СК дефинишу жељене функције система. Жељене функције система, када се извршавају, позивају по одређеном редоследу основне функције система.

Једну акцију сценарија изводи или актор или систем. У том смислу:

Актор изводи једну од три врсте акција:

- Актор Припрема Улаз (Улазне Аргументе) за Системску Операцију (АПУСО).
- Актор Позива систем да изврши Системску Операцију (АПСО).
- Актор извршава Несистемску Операцију (АНСО).

Систем изводи две акције у континуитету:

- Систем извршава Системску Операцију (СО):
- Резултат системске операције (Излазни аргументи (ИА)) се прослеђује до актора.

Деф. ЗА3: **Актор** (учесник) представља спољног корисника система. Он поставља захтев систему да изврши једну или више системских операција, по унапред дефинисаном сценарију⁷. Систем одговара на постављени захтев актора, шаљући му вредност излазних аргумента као резултат извршења операција. Актор се обично дефинише као неко или нешто (нпр: људи, рачунарски системи или организациона јединица) што има понашање.

Правила код прикупљања захтева:

Правило ЗА1 (Независност сценарија СК): Сценарија СК не треба да буду у међусобној интеракцији. Она се требају дефинисати као атомска, у смислу да се извршавају у потпуности самостално. На тај начин се олакшава њихов развој и одржавање [JPRS]⁸.

Правило ЗА2 (Glass' law) [EA1]: Недостаци код дефинисања захтева су основни разлог могућег неуспеха у развоју пројекта (програма).

⁷ Захтев за извршење једне или више системских операција се не одиграва континуално него дискретно. То значи да корисник интерактивно позива једну по једну системску операцију, у дискретним временским интервалима. Из наведеног може да се закључи да сценарио описује интерактивно коришћење софтверског система.

⁸ У разговору са проф. Братиславом Петровићем рекао сам да независност сценарија, повећава редудансу у опису понашања система или истовремено знатно олакшава одржавање система. Рекао сам да су редуданса и одржавање обрнуто сразмерни. Проф. Петровић је рекао да је њихов производ вероватно неки кофицијент. **Било би веома интересантно када би неко успео да формално објасни тај однос и да пронађе наведени кофицијент.**

Правило ЗА3 (Boehm's first law) [EA1]: Уколико се не уоче грешке у току дефинисања захтева, исте се веома тешко могу уклонити у каснијим фазама развоја програма.

Правило ЗА4 (Boehm's second law) [EA1]: Прављење прототипова значајно смањује могуће грешке код дефинисања захтева и његовог развоја, нарочито код дефинисања корисничког интерфејса.

Начин представљања модела СК

СК се у почетним фазама развоја софтвера представљају текстуално док се касније они представљају преко секвенцних дијаграма, дијаграма сарадње, дијаграма прелаза стања или дијаграма активности.

Текстуални опис СК има следећу структуру:

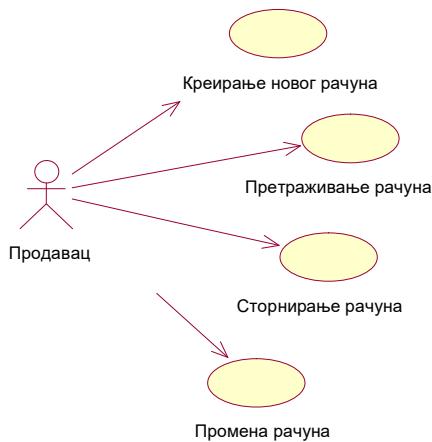
- Назив СК.
- Акторе СК.
- Учеснике СК.
- Предуслови који морају бити задовољени да би СК почeo да сe извршавa.
- Основни сценарио извршења СК.
- Постуслови који морају бити задовољени да би сe потврдило да јe СК успешнo извршен.
- Алтернативна сценарија извршења СК.
- Специјални захтеви.
- Технолошки захтеви.
- Отворена питања.

У нашем примеру имамо следеће СК-а:

1. Креирање новог рачуна.
2. Претраживање рачуна.
3. Сторнирање рачуна.
4. Промена рачуна.

Наведене СК користи Продавац (актор).

Модел СК се може представити преко следећег дијаграма СК:



СК1: Случај коришћења – Креирање новог рачуна

Назив СК

Креирање новог рачуна

Актори СК

Продавац

Учесници СК

Продавац и систем (програм)

Предуслов: Систем је укључен и продавац је улогован под својом шифром. Систем приказује форму за рад са рачуном.

Основни сценарио СК

1. Продавац позива систем да креира нови рачун. (АПСО)
2. Систем креира нови рачун. (СО)
3. Систем приказује продавцу нови рачун и поруку: "Систем је креирао нови рачун". (ИА)
4. Продавац уноси податке у нови рачун. (АПУСО)
5. Продавац контролише да ли је коректно унео податке у нови рачун. (АНСО)
6. Продавац позива систем да запамти податке о рачуну. (АПСО)
7. Систем памти податке о рачуну. (СО)
8. Систем приказује продавцу запамћени рачун и поруку: "Систем је запамтио рачун. (ИА)
9. Продавац позива систем да обради рачун. (АПСО)
10. Систем обрађује рачун.(СО)
11. Систем приказује продавцу обрађен рачун и поруку: "Систем је обрадио рачун".(ИА)

Алтернативна сценарија

- 3.1 Уколико систем не може да креира рачун он приказује продавцу поруку: "Систем не може да креира нови рачун". Прекида се извршење сценарија. (ИА)
- 8.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку "Систем не може да запамти рачун". Прекида се извршење сценарија. (ИА)
- 11.1 Уколико систем не може да обради рачун он приказује продавцу поруку: "Систем не може да обради рачун". (ИА)

СК2: Случај коришћења – Претраживање рачуна

Назив СК

Претраживање рачуна

Актори СК

Корисник

Учесници СК

Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је улогован под својом шифром. Систем приказује форму за рад са рачуном.

Основни сценарио СК

1. Корисник уноси вредност по којој претражује рачун. (АПУСО)
2. Корисник позива систем да нађе рачун по задатој вредности. (АПСО)
3. Систем тражи рачун по задатој вредности. (СО)
4. Систем приказује кориснику податке о рачуну и поруку: "Систем је нашао рачун по задатој вредности". (ИА)

Алтернативна сценарија

- 4.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: "Систем не може да нађе рачун по задатој вредности". (ИА)

СК3: Случај коришћења – Сторнирање рачуна

Назив СК

Сторнирање рачуна

Актори СК

Корисник

Учесници СК

Корисник и **систем** (програм)

Предуслов: Систем је укључен и корисник је улогован под својом шифром. Систем приказује форму за рад са **рачуном**.

Основни сценарио СК

1. Корисник уноси вредност по којој претражује **рачун**. (АПУСО)
2. Корисник позива систем да нађе **рачун** по задатој вредности. (АПСО)
3. Систем тражи **рачун** по задатој вредности. (СО)
4. Систем приказује кориснику **рачун** и поруку: “Систем је нашао **рачун** по задатој вредности”. (ИА)
5. Корисник позива систем да сторнира задати **рачун**. (АПСО)
6. Систем сторнира **рачун**. (СО)
7. Систем приказује кориснику сторниран **рачун** и поруку: “Систем је сторнирао **рачун**”. (ИА)

Алтернативна сценарија

- 4.1 Уколико систем не може да нађе **рачун** он приказује кориснику поруку: “Систем не може да нађе **рачун** по задатој вредности”. Прекида се извршење сценарија. (ИА)
- 7.1 Уколико систем не може да сторнира **рачун** он приказује кориснику поруку: “Систем не може да сторнира **рачун**”.

СК4: Случај коришћења – Промена рачуна

Назив СК

Промена **рачуна**

Актори СК

Продавац

Учесници СК

Продавац и **систем** (програм)

Предуслов: Систем је укључен и продавац је улогован под својом шифром. Систем приказује форму за рад са **рачуном**.

Основни сценарио СК

1. Продавац уноси вредност по којој претражује **рачун**. (АПУСО)
2. Продавац позива систем да нађе **рачун** по задатој вредности. (АПСО)
3. Систем тражи **рачун** по задатој вредности. (СО)
4. Систем приказује продавцу **рачун** и поруку: “Систем је нашао **рачун** по задатој вредности”. (ИА)
5. Продавац уноси (мења) податке о **рачуну**. (АПУСО)
6. Продавац контролише да ли је коректно унео податке о **рачуну**. (АНСО)
7. Продавац позива систем да запамти податке о **рачуну**. (АПСО)
8. Систем памти податке о **рачуну**. (СО)
9. Систем приказује продавцу поруку: “Систем је запамтио **рачун**.” (ИА)
10. Продавац позива систем да обради **рачун**. (АПСО)
11. Систем обрађује **рачун**.(СО)
12. Систем приказује продавцу обрађен **рачун** и поруку: “Систем је обрадио **рачун**.(ИА)

Алтернативна сценарија

- 4.1 Уколико систем не може да нађе **рачун** он приказује продавцу поруку: “Систем не може да нађе **рачун** по задатој вредности”. Прекида се извршење сценарија. (ИА)
- 9.1 Уколико систем не може да запамти податке о **рачуну** он приказује продавцу поруку “Систем не може да запамти **рачун**”. Прекида се извршење сценарија. (ИА)
- 12.1 Уколико систем не може да обради **рачун** он приказује продавцу поруку: “Систем не може да обради **рачун**”. (ИА)

2.2 Анализа

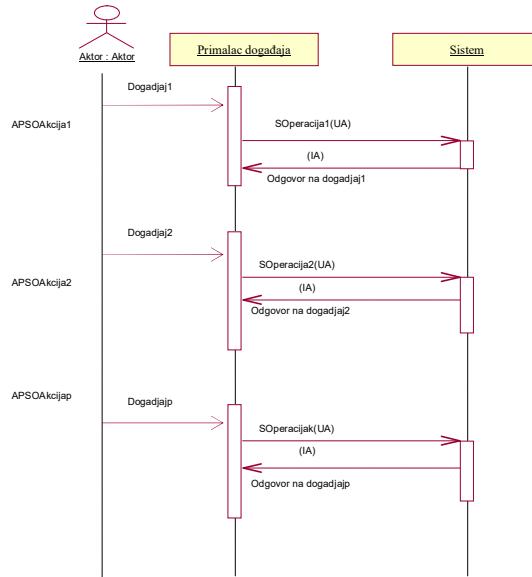
Фаза анализе описује логичку структуру и понашање софтверске система (пословну логику софтверске система). Понашање софтверског система је описано помоћу системских дијаграма секвенци и преко системских операција. Структура софтерског система се описује помоћу концептуалног и релационог модела.

2.2.1 ПОНАШАЊЕ СОФТ. СИСТЕМА - Сист. дијаграми секвенци

Понашање система се може описати преко УМЛ-ових **секвенцних дијаграма** [Larman], односно преко **дијаграма сарадње** [JPRs].

Деф. АН1: Системски дијаграм секвенци приказује, за издвојени сценарио СК, догађаје у одређеном редоследу, који успостављају интеракцију између актора и софтверског система.

Деф. АН2: Догађај који направи актор је побуда за позив системске операције. То значи да актор не позива системску операцију непосредно већ то чини преко посредника (примаоца дугађаја). Позив системске операције указује на интеракцију између актора и система. За дугађај који представља побуду за позив СО се често каже да је то **системски дугађај**.



Дугађаје праве актори (нпр. клик на дугме, које се налази на екранској форми), у оквиру APSO акција, над примаоцем дугађаја (нпр. дугме). Прималац дугађаја прихвата дугађај и позива системску операцију (нпр. дугме прими клик (дугађај) и покреће методу која позива системску операцију) која се налази на страни система. Након извршења системске операције систем враћа неки резултат као одговор на дугађај (то може да буде сигнал о успешности извршења операције и/или неки податак).

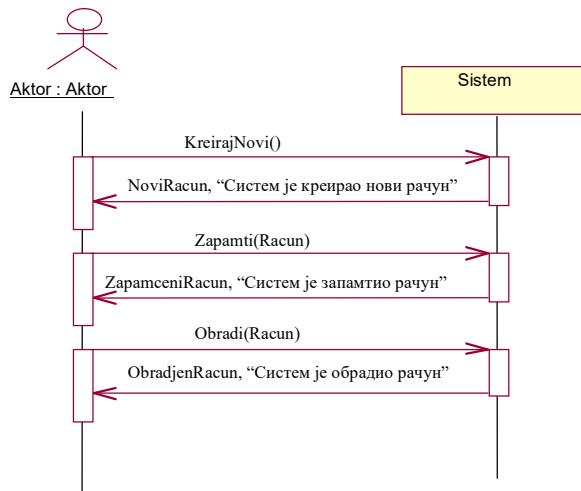
Као резултат анализе сценарија СК добијају се захтеви за извршење системских операција. За сваку системску операцију се праве уговори(контракти).

За сваки СК, прецизније речено за сваки сценаријо СК, праве се системски дијаграми секвенци и то само за АПСО и ИА акције сценарија.

ДС1: Дијаграми секвенци случаја коришћења – Креирање новог рачуна

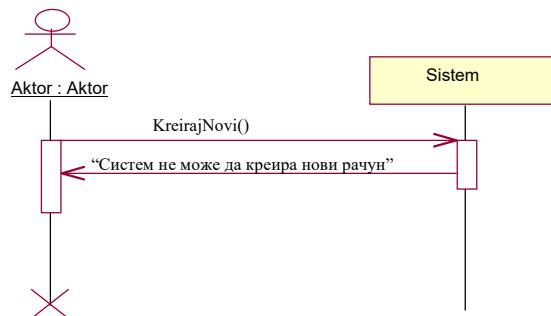
1. Продавац позива систем да креира нови рачун. (АПСО)
2. Систем приказује продавцу нови рачун и поруку: "Систем је креирао нови рачун". (ИА)
3. Продавац позива систем да запамти податке о рачуну. (АПСО)
4. Систем приказује продавцу запамћени рачун и поруку: "Систем је запамтио рачун. (ИА)
5. Продавац позива систем да обради рачун. (АПСО)
6. Систем приказује продавцу обрађен рачун и поруку: "Систем је обрадио рачун".(ИА)

Из наведеног може да се закључи да се на системском дијаграму секвенци не виде АПУСО, СО и АНСО акције.

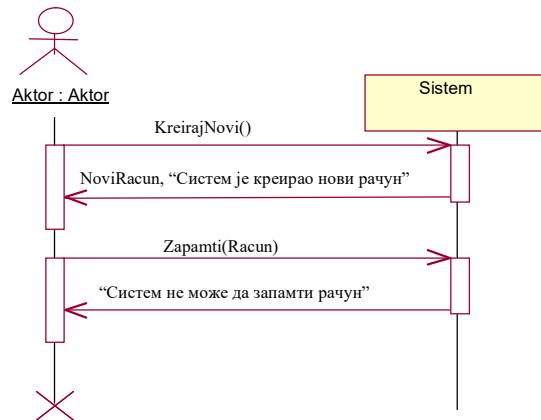


Алтернативна сценарија

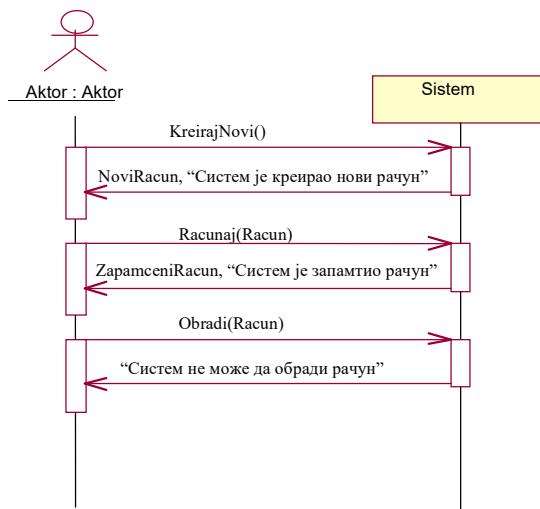
- 2.1 Уколико систем не може да креира рачун он приказује продавцу поруку: "Систем не може да креира нови рачун". Прекида се извршење сценарија. (ИА)



4.1 Уколико **систем** не може да запамти податке о **рачуну** он приказује **продавцу** поруку “**Систем не може да запамти рачун**”. Прекида се извршење сценарија. (ИА)



6.1 Уколико **систем** не може да обради **рачун** он приказује **продавцу** поруку: “**Систем не може да обради рачун**”. (ИА)

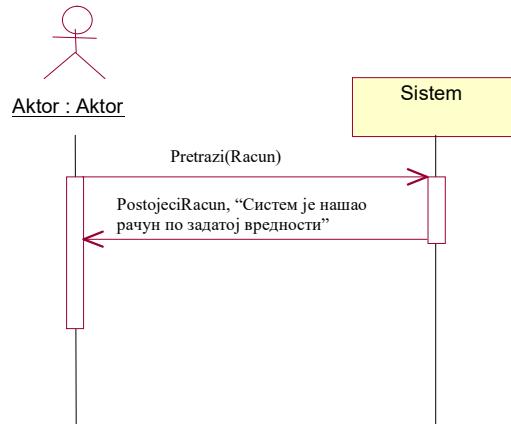


Са наведених секвенцних дијаграма уочавају се 3 системске операције које треба пројектовати:

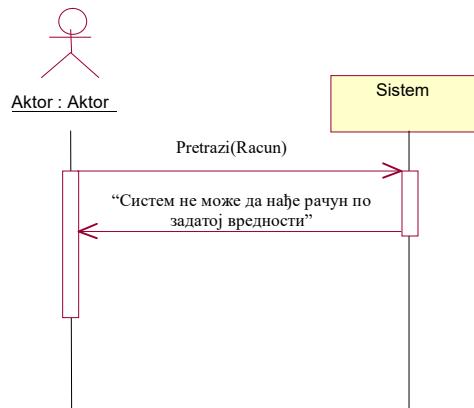
1. *signal KreirajNovi (Racun);*
2. *signal Zapamti(Racun);*
3. *signal Obradi(Racun);*

ДС2: Дијаграми секвенци случаја коришћења – Претраживање рачуна

1. Корисник позива систем да нађе рачун по задатој вредности. (АПСО)
2. Систем приказује кориснику податке о рачуну и поруку: "Систем је нашао рачун по задатој вредности". (ИА)

**Алтернативна сценарија**

- 2.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: "Систем не може да нађе рачун по задатој вредности". (ИА)



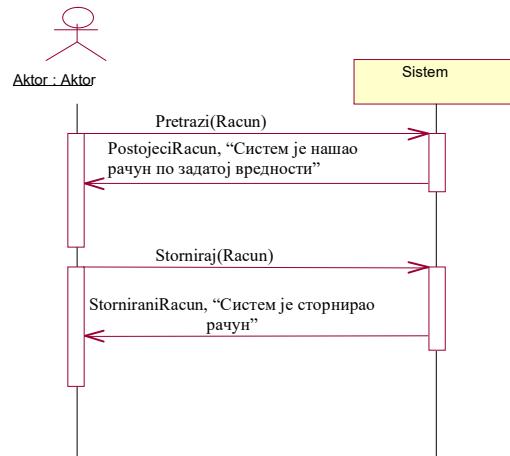
Са наведених секвенцних дијаграма уочава се још једна системска операција коју треба пројектовати:

I. signal Pretrazi (Racun);

ДСЗ: Дијаграми секвенци случаја коришћења – Сторнирање рачуна

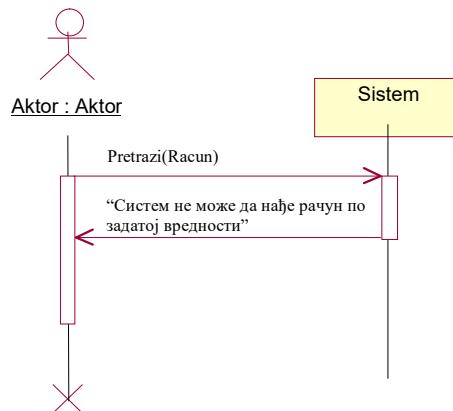
Основни сценарио СК

1. Корисник позива систем да нађе рачун по задатој вредности. (АПСО)
2. Систем приказује кориснику рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)
3. Корисник позива систем да сторнира задати рачун. (АПСО)
4. Систем приказује кориснику сторниран рачун и поруку: "Систем је сторнирао рачун". (ИА)

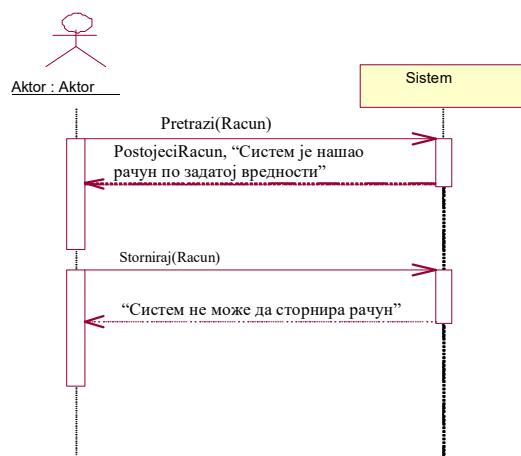


Алтернативна сценарија

- 2.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: "Систем не може да нађе рачун по задатој вредности". Прекида се извршење сценарија. (ИА)



- 4.1 Уколико систем не може да сторнира рачун он приказује кориснику поруку: "Систем не може да сторнира рачун".

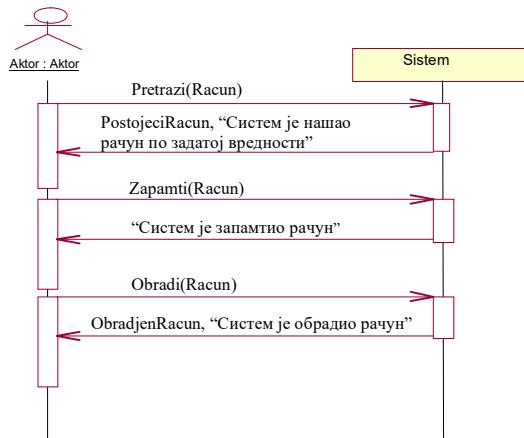


Са наведених секвенцних дијаграма уочава се још једна системска операција коју треба пројектовати:

1. signal **Storniraj (Racun);**

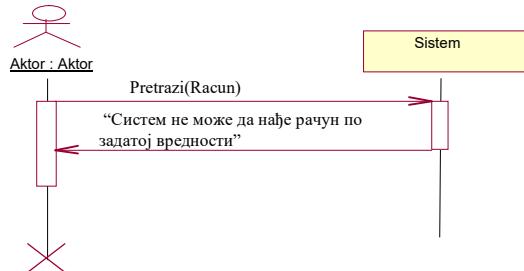
ДС4: Дијаграми секвенци случаја коришћења – Промена рачуна

1. Продавац позива систем да нађе рачун по задатој вредности. (АПСО)
2. Систем приказује продавцу рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)
3. Продавац позива систем да запамти податке о рачуну. (АПСО)
4. Систем приказује продавцу поруку: "Систем је запамтио рачун." (ИА)
5. Продавац позива систем да обради рачун. (АПСО)
6. Систем приказује продавцу обрађен рачун и поруку: "Систем је обрадио рачун." (ИА)

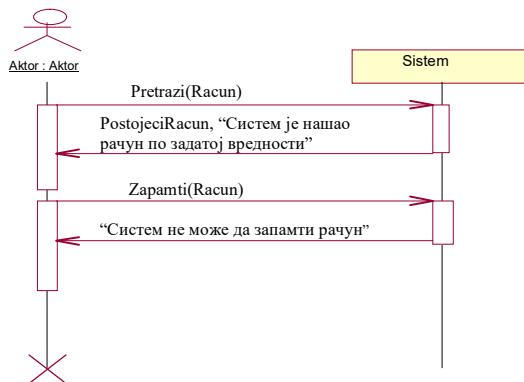


Алтернативна сценарија

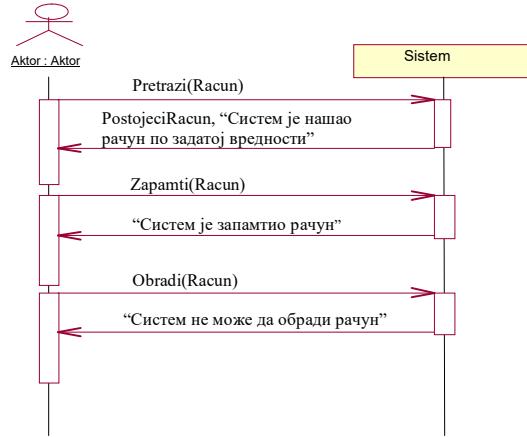
- 2.1 Уколико систем не може да нађе рачун он приказује продавцу поруку: "Систем не може да нађе рачун по задатој вредности". Прекида се извршење сценарија. (ИА)



- 4.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку "Систем не може да запамти рачун". Прекида се извршење сценарија. (ИА)



6.1 Уколико **систем** не може да обради **рачун** он приказује **продавцу** поруку: “**Систем** не може да обради **рачун**”. (ИА)



Како резултат анализе сценарија добијено је укупно 5 системских операција које треба пројектовати:

1. *signal KreirajNovi (Racun);*
2. *signal Zapamti(Racun);*
3. *signal Obradi(Racun);*
4. *signal Pretrazi (Racun);*
5. *signal Storniraj (Racun);*

2.2.2 Дефинисање уговора о системским операцијама

За сваку од уочених системских операција праве се уговори.

Деф. АН3: Системска операција описује понашање софтверског система. Системска операција има свој потпис, који садржи име методе и опционо улазне и/или излазне аргументе.

Деф. АН4: Уговори се праве за системске операције и они описују њено понашање. Уговори описују шта операција треба да ради, без објашњења како ће то да ради. Један уговор је везан за једну системску операцију.

Уговори се сastoјe из следећих секција:

- **Операција** – име операције и њени улазни и излазни аргументи
- **Веза са СК** – имена СК у којима се позива системска операција
- **Предуслов** – пре извршења системске операције морају бити задовољени одређени предуслови (систем мора бити у одговарајућем стању).
- **Постуслови** – после извршења системске операције у систему морају бити задовољени одређени постуслови (систем мора бити у одговарајућем стању или се поништава резултат операције).

Деф. АН5: Постулови СО указују на то шта треба да се деси (ефекти извршења СО), након извршења СО, а не како ће то да се деси.

Постулови се изражавају у прошлом времену, како би се нагласило да је објекат дошао у ново стање, а не да ће доће у ново стање. Нпр. *Ставка рачуна је креирана*. Не би било добро да се напише: *Креирање ставке рачуна*.

Деф. АН6: Предуслови СО указују на то шта је требало да се деси, како би СО могла да се изврши, а не како се то десило.

1. Уговор UG1: *KreirajNovi*

Операција: KreirajNovi (*Racun*):signal;

Веза са СК: СКП31

Предуслови: Вредносна и структурна ограничења над објектом *Racun* морају бити задовољена.

Постулови: Направљен је нови рачун.

2. Уговор UG2: *Zapamti*

Операција: Zapamti(*Racun*):signal;

Веза са СК: CK1, CK4

Предуслови: Ако је рачун обрађен или сторниран не може се извршити системска операција.

Вредносна и структурна ограничења над објектом *Racun* морају бити задовољена.

Постулови:

- Израчуната је вредност сваке од ставки рачуна.
- Израчуната је укупна вредност рачуна.

3. Уговор UG3: *Obradi*

Операција: Obradi(*Racun*):signal;

Веза са СК: CK1, CK4

Предуслови: Ако је рачун обрађен или сторниран не може се извршити системска операција.

Вредносна и структурна ограничења над објектом *Racun* морају бити задовољена.

Постулови:

- Израчуната је вредност сваке од ставки рачуна.
- Израчуната је укупна вредност рачуна.
- Рачун је обрађен.

4. Уговор UG4: *Pretraži*

Операција: Pretraži (*Racun*):signal;

Веза са СК: CK2, CK3, CK4

Предуслови:

Постулови:

5. Уговор UG5: *Storniraj*

Операција: *Storniraj(Racun):signal;*

Веза са СК: *CK3*

Предуслови: *Ако је рачун сторниран не може се извршити системска операција.*

Постуслови: *Рачун је сторниран*

2.2.3 ограничења при извршењу системских операција

Системске операције се могу састоји од више: а) **операција одржавања базе података** (убаци, избаци и промени) и/или б) **операција извештавања** (прикази).

У току извршења системске операције над структуром софтверског система (односно над базом података) подаци (објекти у оперативној меморији и слогови у бази података) морају да остану конзистентни, односно морају да буду задовољена вредносна и структурна ограничења дефинисана над подацима.

Вредносна ограничења се односе на дозвољене вредности атрибути доменских класа (табела) и она се деле на:

а) проста вредносна ограничења - ограничења везана за домен (тип) атрибути и вредност атрибути.

б) сложена вредносна ограничења - ограничења везана за међузависност атрибути.

Структурна ограничења су дефинисана преко кардиналности пресликања између доменских класа (табела).

При извршењу операција **убаци** и **промени** објекат (слог) проверавају се и вредносна и структурна ограничења. Ове провере се раде у одељку **предуслови** код уговора за системске операције. У одељку **постуслови** се наводи резултат операција убаци (нпр. *Направљен је нови рачун*), и промени (нпр. *Рачун је сторниран, Израчуната је укупна вредност рачуна,...*).

Убаци (insert)

Предуслови: *Вредносна и структурна ограничење морају бити задовољена.*

Постуслови: *Наводи се резултат операције.*

Промени (update)

Предуслови: *Вредносна и структурна ограничење морају бити задовољена.*

Постуслови: *Наводи се резултат операције.*

При извршењу операције **обриши** објекат (слог) проверавају се структурна ограничења. Ова провера се ради у одељку **предуслови** или у одељку **постуслови** код уговора за системске операције. У одељку **постуслови** се наводи резултат операција обриши (нпр. *Обрисан је предмет*).

Избаци (delete)

Предуслови: *Структурна ограничење морају бити задовољена.*

Постуслови: *Наводи се резултат операције.*

При извршењу операције **извештавања** не проверавају се вредносна и структурна ограничења. Код уговора за системске операције у одељцима **постуслови** и **предуслови** ништа се не наводи.

Прикази (select)

Предуслови: /

Постуслови: /

2.2.4 СТРУКТУРА СОФТ СИСТЕМА - Концептуални (доменски) модел

Структура софтверског система се описује помоћу концептуалног модела. Наводимо дефиниције концептуалног модела и његових елемената.

Дефиниција АН7: Концептуални модел описује концептуалне класе домена проблема. Концептуални модел садржи концептуалне класе (доменске објекте) и асоцијације између концептуалних класа. Често се за концептуалне моделе каже да су то **доменски модели** или **модели објектне анализе**.

Дефиниција АН8: Концепти (концептуалне класе) представљају атрибуте⁹ софтверског система. То значи да концепти описују структуру софтверског система. Концептуалне класе састоје се од атрибута, који описују особине класе. Концептуалне класе треба разликовати од софтверских класа.

Дефиниција АН9: Атрибути представљају особине која се придржују до концептуалних класа. Сваки од атрибута је везан за одређени тип податка.

Атрибут има конкретну вредност за конкретно појављивање концептуалне класе.

Дефиниција АН10: Асоцијација је веза између концептуалних класа. Сваки крај асоцијације представља улогу концепта који учествује у асоцијацији. Улога садржи име, пресликавање и навигацију.

Име улоге је засновано на формату: ИмеКонцКласе1 – Глагол – ИмеКонцКласе2, где глагол описује однос између концептуалних класа у датом контексту.

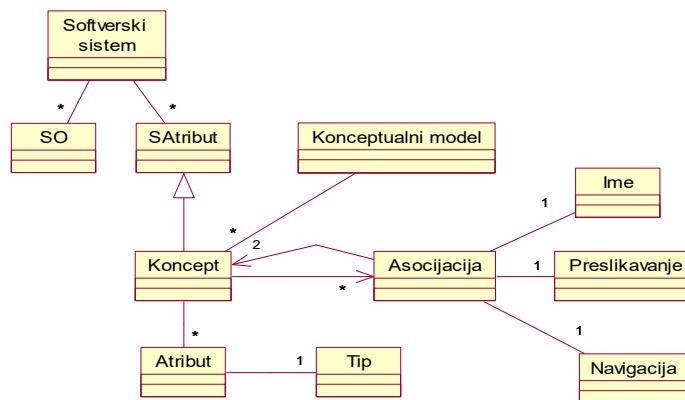
Пресликавање дефинише колико много појављивања концептуалне класе А може бити придржано једном појављивању концептуалне класе Б.

Навигација указује на једносмерне везе између концептуалних класа.

Између концептуалних класа може постојати више асоцијација.

Објашњење дефиниција концептуалног модела и његових елемената

Софтверски систем се састоји од атрибута (САтрибут) и системских операција (СО). Концепти представљају реализацију атрибута софтверског система.

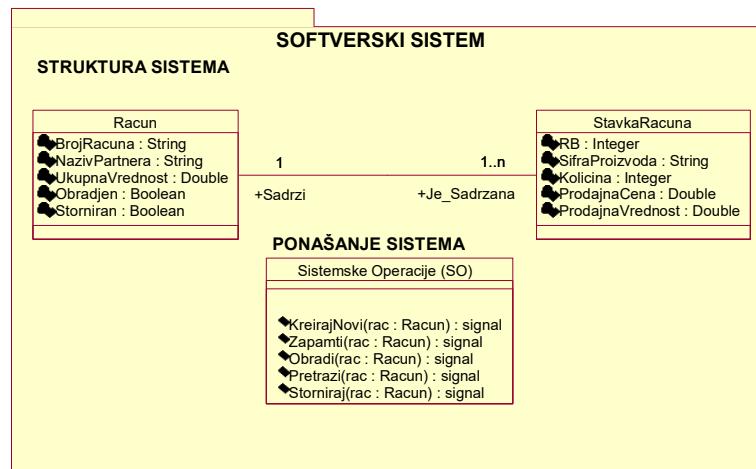


⁹ Концепти се могу **уочити** из UA који се проследују до softverskog система. Меđutim треба нагласити да UA нису концепти. UA су изван softverskog sistema и они представљају улаз у softverski sistem. Концепти су унутар softverskog sistema и они представљају структуру softverskog sistema.

Конкретан пример концептуалног модела

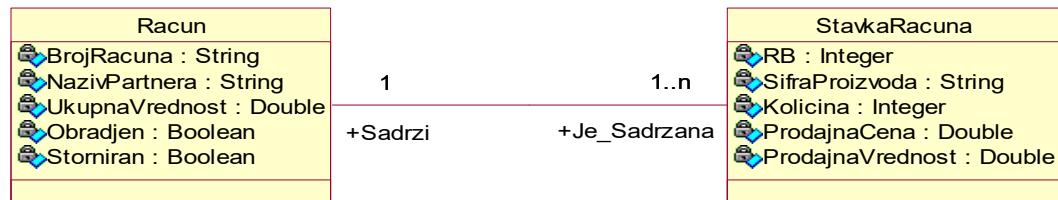


Као резултат анализе сценарија СК и прављења концептуалног модела добија се **логичка структура и понашање софтверског система**:



2.2.5 СТРУКТУРА СОФТ. СИСТЕМА - Релациони модел

На основу концептуалног модела може се направити релациони модел, који ће да представља основу за пројектовање релационе базе података[Ullman].



На основу датог концептуалног модела (Racun, StavkeRacuna) прави се **релациони модел**:

Racun(BrojRacuna, NazivPartnera, UkupnaVrednost, Obradjen, Stomiran);

StavkaRacuna(BrojRacuna, RB, SifraProizvoda, Kolicina, ProdajnaCena, ProdajnaVrednost)

| Табела Racun | | Просто вредносно ограничење | | Сложено вредносно ограничење | | Структурно ограничење |
|-----------------|----------------|-----------------------------|-------------------|--------------------------------|---|---|
| Atributi | Име | Тип атрибута | Вредност атрибута | Међузав. атрибута једне табеле | Међузав. атрибута више табела | |
| Atributi | BrojRacuna | String | not null | | | INSERT / UPDATE CASCADES StavkeRacuna DELETE CASCADES StavkeRacuna |
| | NazivPartnera | String | | | | |
| | UkupnaVrednost | Double | (default:0) | | UkupnaVrednost= SUM (StavkaRacuna.ProdajnaVrednost) | |
| | Obradjen | Boolean | (default: false) | | | |
| | Stomiran | Boolean | (default: false) | | | |

Код сложених објеката (као што је Racun) треба узети следеће ограничење у обзир:

UPDATE Racun ----> DELETE StavkeRacuna (у бази) ----> INSERT StavkeRacuna (из оперативне меморије)

| Табела StavkaRacuna | | Просто вредносно ограничење | | Сложено вредносно ограничење | | Структурно ограничење |
|---------------------|------------------|-----------------------------|-------------------|--|-------------------------------|--|
| Atributi | Име | Тип атрибута | Вредност атрибута | Међузав. атрибута једне табеле | Међузав. атрибута више табела | |
| Atributi | BrojRacuna | String | Not null | | | INSERT RESTRICTED Racun UPDATE RESTRICTED Racun DELETE / |
| | RB | Integer | Not null and > 0 | | | |
| | SifraProizvoda | String | | | | |
| | Kolicina | Integer | >0 (default:0) | | | |
| | ProdajnaCena | Double | >0 (default:0) | | | |
| | ProdajnaVrednost | Double | (default:0) | ProdajnaVrednost = Kolicina*ProdajnaCena | | |

2.3 Пројектовања

Фаза пројектовања описује физичку структуру и понашање софтверског система (архитектуру софтверског система). Пројектовање архитектуре софтверског система обухвата пројектовање корисничког интерфејса, апликационе логике и складишта података. Пројектовање корисничког интерфејса обухвата пројектовање екранских форми и контролера корисничког интерфејса. У оквиру апликационе логике се пројектују контролер апликационе логике, пословна логика и брокер базе података. Пројектовање пословне логике обухвата пројектовање логичке структуре и понашања софтверског система.

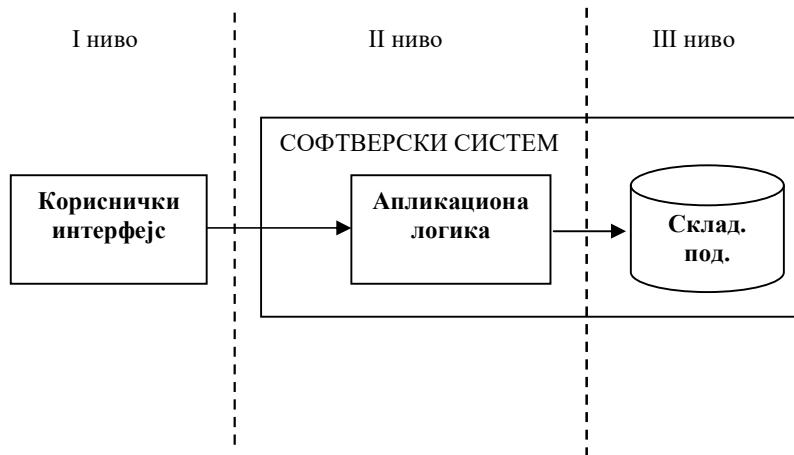
АРХИТЕКТУРА СОФТВЕРСКОГ СИСТЕМА

Пре него што кренемо на пројектовање структуре и понашања софтверског система потребно је да се дефинише архитектура софтверског система. Ми ћемо користити класичну тронивојску архитектуру [TK78,Garther 95].

Дефиниције и правила архитектуре софтверског система,

Деф. PRAR1: Тронивојска архитектура се састоји из следећих ниво:

1. **Корисничког интерфејса** који представља улазно – излазну репрезентацију софтверског система.
2. **Апликационе логике** која описује структуру и понашање софтверског система.
3. **Складишта података** који чува стање атрибута софтверског система.



Слика ТНА: Тронивојска архитектура

Правило PRpARH1: Апликационе логике се пројектује независно од корисничког интерфејса и обрнуто.

Правило PRpARH2: Апликационе логике може да има различите улазно-излазне репрезентације.

Правило PRpARH3 (Model-View Separation Principle): Апликационе логике (модел) нема знања о томе где се налази кориснички интерфејс (поглед).

Деф. ПРАР2: На основу тронивојске архитектуре су направљени савремени апликациони сервери.

Деф. ПРАР3: Апликациони сервери су одговорни да обезбеде сервисе који ће да омогуће реализацију апликационе логике софтверског система. Сваки апликациони сервер се састоји из три основна дела:

1. део за комуникацију са клијентим (контролер)
2. део за комуникацију са неким складиштем података (база података или датотечни систем)
3. део који садржи пословну логику



Деф. ПРАР4: Контролер је одговоран да прихвати захтев за извршење системске операције од клијента и да га проследи до пословне логике која је одговорна за извршење системске операције.

Деф. ПРАВ5: Пословна логика је описана са структуром (доменским класама) и понашањем (системским операцијама).

Деф. ПРАР6: Брокер базе података је одговоран за комуникацију између пословне логике и складишта података.

У даљем тексту ћемо пројектовати сваки од наведених елеманата тронivoјске архитектуре:

- контролер
 - пословна логика – доменске класе
 - пословна логика – системске операције
 - базе података
 - датабасе брокер
 - складиште података
 - кориснички интерфејс

Из наведеног можемо да закључимо да смо у фазама прикупљања захтева и анализе дали спецификацију структуре и понашања софтверског система, односно **спецификацију пословне логике софтверског система** (Slika ASSPL).

Из наведеног можемо да закључимо да смо у фазама прикупљања захтева и анализе дали спецификацију структуре и понашања софтверског система, односно **спецификацију пословне логике софтверског система**.



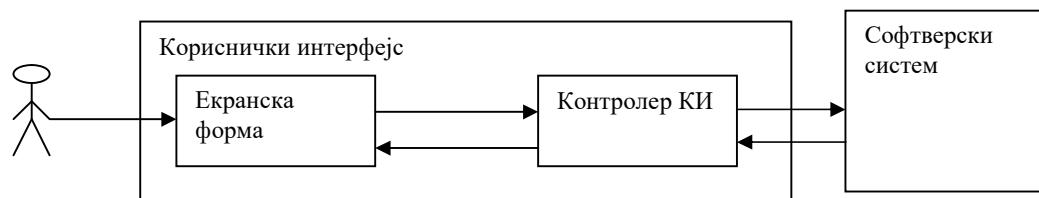
Слика АССПЛ: Архитектура софт. система – Пословна логика

2.3.1 Пројектовање корисничког интерфејса

Кориснички интерфејс, сходно Деф. Лаз1, представља реализацију улаза и/или излаза софтверског система. Пре пројектовања корисничког интерфејса објаснићемо делове корисничког интерфејса (његову структуру), начин њихове међусобне комуникације и начин комуникације корисничког интерфејса са софтверским системом (Слика СКИ).

Кориснички интерфејс се састоји од:

1. Екранске форме која је одговорна да:
 - а) приhvата податке које уноси актор,
 - б) приhvата догађаје које прави актор,
 - ц) позива контролера графичког интерфејса, прослеђујући му прихваћене податке
 - д) приказује податке које је добио од контролера графичког интерфејса.
2. Контролера корисничког интерфејса који је одговоран да:
 - а) приhvati податке које шаље екранска форма,
 - б) конвертује податке (који се налазе у графичким елементима) у објекат који представља улазни аргумент СО која ће бити позвана,
 - ц) шаље захтев за извршење СО до апликационог сервера (софтверског система),
 - д) приhvata објекат (излаз) софтверског система који настаје као резултат извршења СО и
 - е) конвертује објекат у податке графичких елемената.



Слика СКИ: Структура корисничког интерфејса

2.3.1.1 Пројектовање екранске форме

Екранска форма треба, за наведени пример, да има следећи изглед:

| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| | | | | |

Кориснички интерфејс је дефинисан преко скупа екранских форми. Сценарија коришћења екранских форми је директно повезан са сценаријима случајева коришћења.

Постоје два аспекта пројектовања екранске форме:

- Пројектовање сценарија СК који се изводе преко екранске форме.
- Пројектовање метода екранске форме.

Пројектовање сценарија СК

СК1: Случај коришћења – Креирање новог рачуна

Назив СК

Креирање новог рачуна

Актори СК

Продавац

Учесници СК

Продавац и систем (програм)

Предуслов: Систем је укључен и **продавац** је улогован под својом шифром. Систем приказује форму за рад са **рачуном**.

The screenshot shows a Windows-style application window titled 'Racun'. At the top right are standard window controls (minimize, maximize, close). Below the title are four buttons: 'Kreiraj' (highlighted in blue), 'Zapamti', 'Obradi', and 'Storniraj'. The main area has several input fields and checkboxes. On the left, there's a 'Broj racuna:' field with a dropdown arrow and a 'Pretrazi' button with a magnifying glass icon. Below it is a 'Naziv partnera:' input field. To the right are two checkboxes: 'Obradjen' and 'Storniran'. In the center is a table with columns: Redni broj, Sifra proizvoda, Kolicina, Prodajna cena, and Prodajna vred... (Prodajna vrednost). The bottom right contains a 'Ukupna vrednost:' label and a text input field showing '0'.

Основни сценаријо СК

1. Продавац позива систем да креира нови рачун. (АПСО)

Опис акције: Продавац кликом на дугме "Kreiraj" позива системску операцију **kreirajNovi (Racun)** која прави нови рачун.

2. Систем креира нови рачун. (СО)

3. Систем приказује продавцу нови рачун и поруку: "Систем је креирао нови рачун". (ИА)

The screenshot shows the same 'Racun' window as before, but with a message box overlaid. The message box has a title 'Poruka:' and the text 'Sistem je kreirao novi racun...' (System has created a new bill...). The rest of the window is visible below the message box.

4. Продавац уноси податке у нови рачун. (АПУСО)

The screenshot shows a Windows application window titled "Racun". The interface includes tabs for "Kreiraj" (Create), "Zapamti" (Save), "Obradi" (Process), and "Storniraj" (Cancel). A status bar at the top right shows checkboxes for "Obradjen" (Processed) and "Storniran" (Cancelled). The main area contains fields for "Broj racuna:" (Bill number: 0001) and "Naziv partnera:" (Partner name: Pera Peric). Below these are tables for item details and a summary table.

| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| 1 s3 | | 5 | 120 | 600 |
| 2 s5 | | 2 | 250 | 500 |

| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| 1 s3 | | 5 | 120 | 600 |
| 2 s5 | | 2 | 250 | 500 |

Ukupna vrednost: 0

5. Продавац контролише да ли је коректно унео податке у нови рачун. (АНСО)
6. Продавац позива систем да запамти податке о рачуну. (АПСО)
Опис акције: Продавац кликом на дугме "Zapamti" позива системску операцију **Zapamti (Racun)** која памти нови рачун.
7. Систем памти податке о рачуну. (СО)
8. Систем приказује продавцу запамћени рачун и поруку: "Систем је запамтио рачун". (ИА)

The screenshot shows the same "Racun" application window. The "Zapamti" tab is now highlighted. The status bar shows the "Obradjen" checkbox is unchecked. The summary table shows the total value as 1.100.

| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| 1 s3 | | 5 | 120 | 600 |
| 2 s5 | | 2 | 250 | 500 |

Ukupna vrednost: 1.100

9. Продавац позива систем да обради рачун. (АПСО)
Опис акције: Продавац кликом на дугме "Obradi" позива системску операцију **Obradi (Racun)** која обрађује нови рачун.
10. Систем обрађује рачун.(СО)
11. Систем приказује продавцу обрађен рачун и поруку: "Систем је обрадио рачун".(ИА)

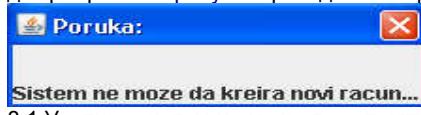
The screenshot shows the "Racun" application window. The "Obradi" tab is now highlighted. The status bar shows the "Obradjen" checkbox is checked. The summary table shows the total value as 1.100.

| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| 1 s3 | | 5 | 120 | 600 |
| 2 s5 | | 2 | 250 | 500 |

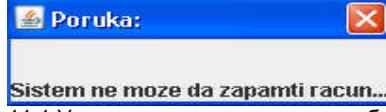
Ukupna vrednost: 1.100

Алтернативна сценарија

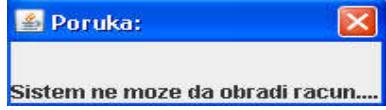
3.1 Уколико **систем** не може да креира **рачун** он приказује **продавцу** поруку: “**Систем** не може да креира нови **рачун**”. Прекида се извршење сценарија. (ИА)



8.1 Уколико **систем** не може да запамти податке о **рачуни** он приказује **продавцу** поруку “**Систем** не може да запамти **рачун**”. Прекида се извршење сценарија. (ИА)



11.1 Уколико **систем** не може да обради **рачун** он приказује **продавцу** поруку: “**Систем** не може да обради **рачун**”. (ИА)



На сличан начин за наведени пример унећемо још 2 рачуна.

СК2: Случај коришћења – Претраживање рачуна**Назив СК**

Претраживање **рачуна**

Актори СК

Корисник

Учесници СК

Корисник и **систем** (програм)

Предуслов: Систем је укључен и **корисник** је улогован под својом шифром. Систем приказује форму за рад са **рачуном**.

| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| 1 | s1 | 1 | 100 | 100 |

Ukupna vrednost:
100

Основни сценарио СК

1. **Корисник** уноси вредност по којој претражује **рачун**. (АПУСО)
Опис акције: Корисник уноси вредност у погље под називом *Pretrazi*.

2. Корисник позива систем да нађе рачун по задатој вредности. (АПСО)

Опис акције: Корисник након уноса вредности у поље под називом *Pretrazi* притиска тику *<Enter>* и позива системску операцију *Pretrazi (Racun)*.

3. Систем тражи рачун по задатој вредности. (СО)

4. Систем приказује кориснику податке о рачуну и поруку: "Систем је нашао рачун по задатој вредности". (ИА)

Алтернативна сценарија

4.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: "Систем не може да нађе рачун по задатој вредности". (ИА)



СК3: Случај коришћења – Сторнирање рачуна

Назив СК
Сторнирање рачуна

Актори СК
Корисник

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је улогован под својом шифром. Систем приказује форму за рад са рачуном.

Основни сценарио СК

1. Корисник уноси вредност по којој претражује рачун. (АПУСО)
Опис акције: Корисник уноси вредност у поље под називом Pretrazi.

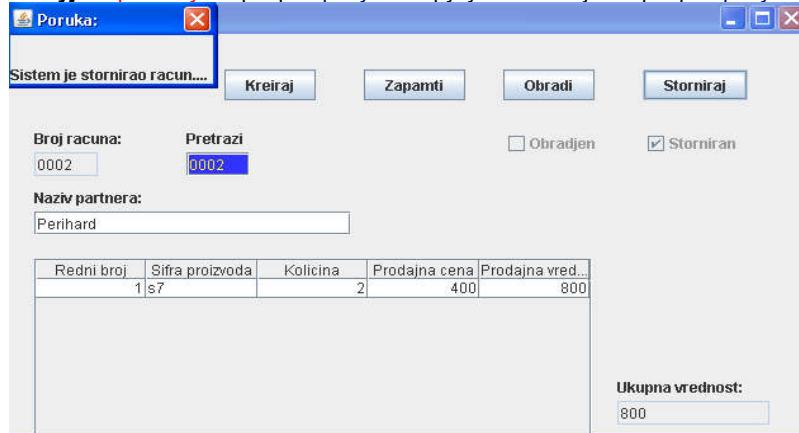
| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| 1 | s7 | 2 | 400 | 800 |

2. Корисник позива систем да нађе рачун по задатој вредности. (АПСО)
Опис акције: Корисник након уноса вредности у поље под називом Pretrazi притиска тапку <Enter> и позива системску операцију Pretrazi (Racun).
3. Систем тражи рачун по задатој вредности. (СО)
4. Систем приказује кориснику рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)

| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| 1 | s7 | 2 | 400 | 800 |

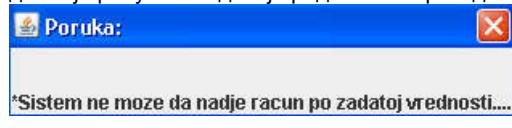
5. Корисник позива систем да сторнира задати рачун. (АПСО)
Опис акције: Корисник кликом на дугме "Storniraj" позива системску операцију Storniraj (Racun).
6. Систем сторнира рачун. (СО)

7. Систем приказује кориснику сторниран рачун и поруку: "Систем је сторнирао рачун". (ИА)

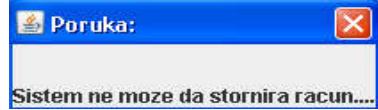


Алтернативна сценарија

4.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: "Систем не може да нађе рачун по задатој вредности". Прекида се извршење сценарија. (ИА)



7.1 Уколико систем не може да сторнира рачун он приказује кориснику поруку: "Систем не може да сторнира рачун".



СК4: Случај коришћења – Промена рачуна

Назив СК

Промена рачуна

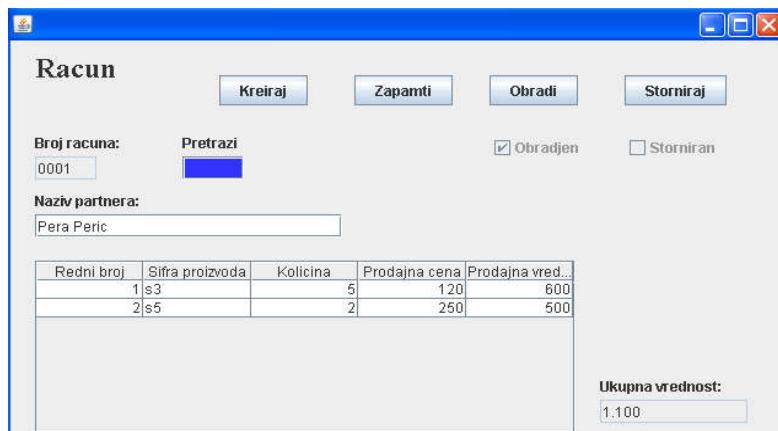
Актори СК

Продавац

Учесници СК

Продавац и систем (програм)

Предуслов: Систем је укључен и продавац је улогован под својом шифром. Систем приказује форму за рад са рачуном.



Основни сценарио СК

1. Продавац уноси вредност по којој претражује рачун. (АПУСО)
Опис акције: Продавац уноси вредност у поље под називом Pretrazi.

The screenshot shows the 'Racun' application window. At the top, there are four buttons: 'Kreiraj' (Create), 'Zapamti' (Save), 'Obradi' (Process), and 'Storniraj' (Cancel). Below these are two input fields: 'Broj racuna:' containing '0001' and 'Pretrazi' containing '0003'. A circled 'Pretrazi' field indicates it is the active input. To the right are two checkboxes: 'Obradjeno' (checked) and 'Storniran' (unchecked). A table below shows bill details for 'Pera Peric':

| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| 1 | s3 | 5 | 120 | 600 |
| 2 | s5 | 2 | 250 | 500 |

On the right side, there is a label 'Ukupna vrednost:' followed by a text box containing '1.100'.

2. Продавац позива систем да нађе рачун по задатој вредности. (АПСО)
3. Систем тражи рачун по задатој вредности. (СО)
4. Систем приказује продавцу рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)

The screenshot shows the 'Poruka' application window with the message 'Sistem je nasao racun po zadatoj vrednosti...' (System found the bill by the specified value...). It has four buttons: 'Zapamti' (Save), 'Obradi' (Process), and 'Storniraj' (Cancel). Below this is a copy of the 'Racun' application window from the previous step, showing the same search results for 'Meridian Invest'.

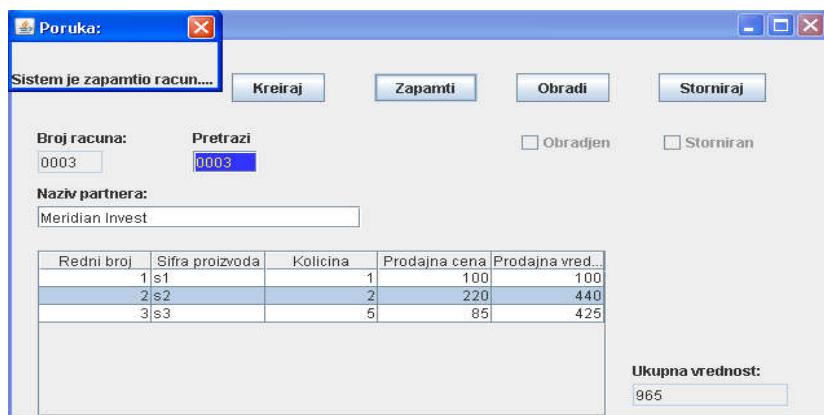
5. Продавац уноси (мења) податке о рачуну. (АПУСО)

The screenshot shows the 'Racun' application window again. The 'Pretrazi' field now contains '0003'. The table data has been modified:

| Redni broj | Sifra proizvoda | Kolicina | Prodajna cena | Prodajna vred... |
|------------|-----------------|----------|---------------|------------------|
| 1 | s1 | 1 | 100 | 100 |
| 2 | s2 | 2 | 220 | 0 |
| 3 | s3 | 5 | 85 | 0 |

The 'Ukupna vrednost:' field on the right is now '100'.

6. Продавац контролише да ли је коректно унео податке о рачуну. (АНСО)
7. Продавац позива систем да запамти податке о рачуну. (АПСО)
Опис акције: Продавац кликом на дугме "Zapamti" позива системску операцију **Zapamti (Racun)**.
8. Систем памти податке о рачуну. (СО)
9. Систем приказује продавцу запамћени рачун и поруку: "Систем је запамтио рачун." (ИА)

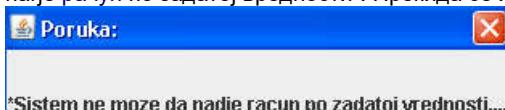


10. Продавац позива систем да обради рачун. (АПСО)
Опис акције: Продавац кликом на дугме "Obradi" позива системску операцију **Obradi (Racun)** која обрађује нови рачун.
11. Систем обрађује рачун.(СО)
12. Систем приказује продавцу обрађен рачун и поруку: "Систем је обрадио рачун.(ИА)"

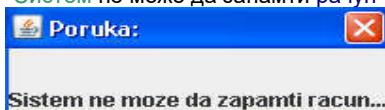


Алтернативна сценарија

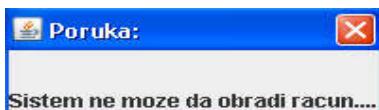
- 4.1 Уколико систем не може да нађе рачун он приказује продавцу поруку: "Систем не може да нађе рачун по задатој вредности". Прекида се извршење сценарија. (ИА)



- 9.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку "Систем не може да запамти рачун". Прекида се извршење сценарија. (ИА)



- 12.1 Уколико систем не може да обради рачун он приказује продавцу поруку: "Систем не може да обради рачун". (ИА)



Пројектовање метода екранске форме

Постоје две класе које се користе у пројектовању метода екранске форме: **OpstaEkranskaForma** и **EkranskaFormaRacun**. Абстрактна класа OpstaEkranskaForma садржи опште методе које су независне од екранске форме која ће да репрезентује неку доменску класу. Док класа EkranskaFormaRacun, садржи методе које приказују екранску форму која репрезентује конкретну доменску класу (у нашем примеру рачун) са свим припадајућим графичким елементима. Класа EkranskaFormaRacun је повезана са класом KontrolerKIRacun којој прослеђује графички објекат (при иницијализацији) и захтев за извршење системске операције.

```

abstract class OpstaEkranskaForma extends ... // (navodi se ime klase koja omogućava kreiranje
                                                // ekranse forme)
{
    ... // navode se opšte metode ekranse forme.
    abstract OpstiDomenskiObjekat kreirajObjekat();
}

class EkranskaFormaRacun extends OpstaEkranskaForma
{ KontrolerKIRacun kkir;

    // Glavni program
    public static void main(String args[])
    { EkranskaFormaRacun EF = new EkranskaFormaRacun();
    }

    // 1. Konstruktor ekranse forme
    public EkranskaFormaRacun ()
    { krerajKomponenteEkranskeForme();           // 1.1
        pokreniMenadzeraRasporedaKomponeti();     // 1.2
        postaviIimeForme();                        // 1.3
        postaviTextFieldBrojRacuna();              // 1.4
        postaviTextFieldNazivPartnera();            // 1.5
        postaviTextFieldUkupnaVrednost();            // 1.6
        postaviTextFieldPretrazivanje();             // 1.7
        postaviCheckBoxObradjeni();                 // 1.8
        postaviCheckBoxStorniran();                 // 1.9
        postaviDugmeKreiraj();                     // 1.10
        postaviDugmeObradi();                      // 1.11
        postaviDugmeStorniraj();                   // 1.12
        postaviDugmeZapamti();                    // 1.13
        postaviTabelu();                           // 1.14
        inicializacijaKontrolera();                // 1.15
    }

    ...
    void postaviDugmeKreiraj()
    { ...
        // Kada se klikne na dugne poziva se:
        String signal = kkir.SOKreirajNovi();
        ...
    }

    void postaviDugmeObradi()
    { ...
        // Kada se klikne na dugne poziva se:
        String signal = kkir.SOObrađi();
        ...
    }
}

```

```
void postaviDugmeStorniraj()
{ ...
    // Kada se klikne na dugme poziva se:
    String signal = kkir.SOStorniraj();
    ...
}

void postaviDugmeZapamti()
{ ...
    // Kada se klikne na dugme poziva se:
    String signal = kkir.SOZapamti();
    ...
}

void postaviTabelu()
{ ...
    // Kada se pritisne neka od tipki na tabeli poziva se:
    String signal = kkir.pritisakTipke(evt);
    ...
}

// 1.15 Inicijalizacija KontroleraKI
// Pri inicijalizaciji, kontroler dobija referencu na graficki objekat (this).
void inicijalizacijaKontrolera()
{ kkir = new KontrolerKIRacun (this); }

OpstiDomenskiObjekat kreirajObjekat() {return new Racun();}

}
```

2.3.1.2 Пројектовање контролера корисничког интерфејса

Контролер корисничког интерфејса треба пројектовати тако да има општи део (*OpstiKontrolerKI*) који је независтан од екранске форме преко које се извршава сценаријо случаја коришћења и конкретни део који је везан за домен екранске форме (*KontrolerKIRacun*).

Општи контролер:

- a) успоставља везу између екранске форме и апликационе логике.
- b) прихвата од екранске форме захтев за извршење системске операције.
- c) креира доменски објекат.
- d) прослеђује захтев за извршење системске операције и доменске објекте до апликационог сервера (апликационе логике).
- e) прихвата доменске објекте и сигнале (о успешности извршења СО) које је вратио апликациони сервер као резултат извршења системске операције.

Конкретни контролер:

- a) прихвата од екранске форме графичке објекте.
- b) конвертује графичке објекте у доменске објекте који ће бити прослеђени преко мреже до апликационог сервера.
- c) конвертује доменске објекте у графичке објекте и прослеђује их до екранске форме.

```
abstract class OpstiKontrolerKI
```

```
{ AplikacionaLogika al;
  String signal;
  OpstiDomenskiObjekat odo;
  OpstaEkanskaForma oef;

// a) успоставља везу између екранске форме и апликационе логике преко сокета.
  OpstiKontrolerKI()
  { UspostaviVezuIzmedjuEkranskeFormeIAplikacioneLogike(); }

  public String pritisakTipke(KeyEvent evt)
  { OdredjujeSeNacinObradiTipkiPriRaduSaTabelom(); }
```

```
// b) прихвата од екранске форме захтев за извршење системске операције.
  public String SOPretrazi()
  { // c) креира доменски објекат.
    odo = oef.kreirajObjekat();
    KonvertujGrafickiObjekatUDomenskiObjekat();
    signal = pozivSO("Pretrazi");
    KonvertujDomenskiObjekatUGrafickiObjekat();
    return signal;
  }
```

```
  public String SOKreirajNovi()
  { odo = oef.kreirajObjekat();
    signal = pozivSO("kreirajNovi");
    KonvertujObjekatUGrafickeKomponente();
    return signal;
  }
```

```
  public String SOZapamti()
  { odo = oef.kreirajObjekat();
    KonvertujGrafickiObjekatUDomenskiObjekat ();
    signal = pozivSO("Zapamti");
    KonvertujDomenskiObjekatUGrafickiObjekat();
    return signal;
  }
```

```
  public String SOStorniraj()
  { odo = oef.kreirajObjekat();
    KonvertujGrafickiObjekatUDomenskiObjekat();
    signal = pozivSO("Storniraj");
    KonvertujDomenskiObjekatUGrafickiObjekat();
```

```
        return signal;
    }

    public String SOObradi()
    {
        odo = oef.kreirajObjekat();
        KonvertujGrafickiObjekatUDomenskiObjekat();
        signal = pozivSO("Obradi");
        KonvertujDomenskiObjekatUGrafickiObjekat();
        return signal;
    }

    // d) прослеђује захтев за извршење системске операције и доменске објекте до апликационог
    // сервера (апликационе логике).
    // е) прихвата доменске објекте и сигнале (о успешности извршења СО) које је вратио
    // апликациони сервер као резултат извршења системске операције.
    String pozivSO(String nazivSO)
    {
        signal = PozivAplikacionogServera(odo);
        return signal;
    }

    abstract public void KonvertujGrafickiObjekatUDomenskiObjekat();
    abstract public void KonvertujDomenskiObjekatUGrafickiObjekat();
}

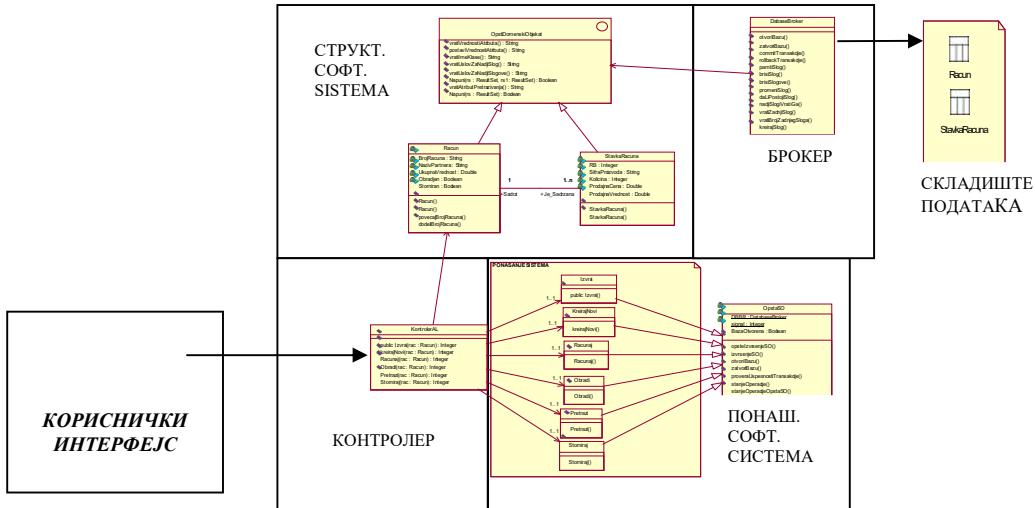
class KontrolerKIRacun extends OpstiKontrolerKI
{
    // a) прихвата од екранске форме графичке објекте.
    KontrolerKIRacun(EkranskaFormaRacun efr) {oef = efr;}

    // b) конвертује графичке објекте у доменске објекте који ће бити прослеђени преко мреже до
    // апликационог сервера.
    public void KonvertujGrafickiObjekatUDomenskiObjekat()
    {
        Racun rac = (Racun) odo;
        EkranskaFormaRacun efr = (EkranskaFormaRacun) oef;
        KonvertujeElementeGrafickogObjektauAtributeDomeneskogObjekta();
        KonvertujeRedoveTabeleStavkeRacunaUNizObjekataStavkeRacuna();
    }

    // c) конвертује доменске објекте у графичке објекте и прослеђује их до екранске форме.
    public void KonvertujDomenskiObjekatUGrafickiObjekat()
    {
        Racun rac = (Racun) odo;
        EkranskaFormaRacun efr = (EkranskaFormaRacun) oef;
        KonvertujeAtributeDomeneskogObjektaUElementeGrafickogObjekta();
        KonvertujeNizObjekataStavkeRacunaURedoveTabeleStavkeRacuna();
    }
}
```

Након пројектовања корисничког интерфејса добија се следећи дијаграм класа:

Кориснички интерфејс у контексту архитектуре софтверског система може се представити на следећи начин:



2.3.2 Пројектовање апликационе логике

2.3.2.1 Контролер апликационе логике

Контролер апликационе логике треба да подигне серверски сокет који ће да ослушају мрежу. Када клијент (клијентски сокет) успостави конекцију са контролером (серверским сокетом), тада контролер треба да генерише нит која ће успоставити двосмерну везу са клијентом (указну и излазну). Слање и примање података од клијента се остварује преко сокета. Клијент шаље захтев за извршење неке од СО до одговарајуће нити (коју смо назвали “нит клијента”), која је повезана са тим клијентом. “Нит клијента” прима захтев и даље га преусмерава до класа које су одговорне за извршење СО. Након извршења СО резултат се враћа до апликационе логике, односно до “нити клијента”, која тада резултат шаље назад до клијента.

Дајемо приказ пројектоване класа *KontrolerAL* и *NitKlijenta*.

```
class KontrolerPL // Kontroler poslovne logike
{
    void main(String[] args)
    {
        serverskiSoket = podizanjeServerskogSoketa();
        for(...)

        { klijentskiSoket = serverskiSoket.osluskivanjeMreze();
          NitKlijenta.kreiranjeNitiKlijenta(klijentskiSoket);

        }
    }

}

class NitKlijenta {

    void kreiranjeNitiKlijenta (klijentskiSoket)
    {
        povezivanjeNitiSaKlijentskimSoketom();
        Ulaz ul = kreiranjeUlazaPrekoKlijentskogSoketa();
        Izlaz iz = kreiranjeIzlazaPrekoKlijentskogSoketa();

        ImeOperacije = ul.PrihvatiimeOperacijeOdKlijenta();
OpstiDomenskiObjekat odo = ul.PrihvatiDomenskiObjekatOdKlijenta();

        if (ImeOperacije = "kreirajNovi") signal = KreirajNovi.kreirajNovi(odo);
        if (ImeOperacije = "Pretrazi") signal = Pretrazi.Pretrazi(odo);
        if (ImeOperacije = "Zapamti") signal = Racunaj.Zapamti(odo);
        if (ImeOperacije = "Obradi") signal = Obradi.Obradi(odo);
        if (ImeOperacije = "Storniraj") signal = Storniraj.Storniraj(odo);

        izl.SlanjeDomenetskogObjektaDoKlijenta(rac);
        izl.SlanjeSignalaDoKlijenta(signal);
    }
}
```

2.3.2.2 Пословна логика

2.3.2.2.1 Пројектовање понашања софтверског система – системске операције

Препорука ПР1: У почетку пројектовања СО треба треба направити концептуалне реализације (решења) за сваку СО. Концептуалне реализације требају да буду директно повезане са логиком проблема.

Препорука ПР2: Може се предпоставити да се подаци (стање софтверског система) чувају у бази. У том смислу могу се позвати неке од основних операција базе (insert,update,delete,find,...), без улажења у начин њихове реализације.

Препорука ПР3: Аспекти реализације који се односе на конекцију са базом, перзистентност и трансакције треба избећи у почетку пројектовања СО. У каснијим фазама наведени аспекти требају ортогонално да се повежу са пројектованим решењима СО, како би се логика решења проблема независно од њих развијала.

Препорука ПР4: Концептуалне реализација се могу описати преко објектног псевдокода, дијаграма сарадње [Larman,JPRS], секвенцних дијаграма¹⁰ [Larman,JPRS], дијаграма активности, дијаграма прелаза стања или дијаграма структура [Budgen].

За сваки од уговора пројектује се концептуално решење.

Projektovanje konceptualnih rešenja SO

1. Уговор UG1: *KreirajNovi*

Операција: *KreirajNovi* (*OpstiDomenskiObjekat*):signal;

Веза са СК: СКП31

Предуслови:

Постуслови: Направљен је нови доменски објекат..

```
class KreirajNovi extends OpstaSO
{
    public static String kreirajNovi(OpstiDomenskiObjekat odo)
    {
        KreirajNovi kn = new KreirajNovi();
        OpstaSO.transakcija = true;
        return OpstaSO.opstelzvrsenjeSO(odo,kn);
    }

    // Prekrivanje metode klase OpstaSO
    boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
    {
        if (!BBP.kreirajSlog(odo))
            BBP.dodajPorukuMetode("Sistem ne moze da kreira " + odo.vratiNazivNovogObjekta() + ".");
        return false;
    }
    BBP.dodajPorukuMetode("Sistem je kreirao " + odo.vratiNazivNovogObjekta() + ".");
    return true;
}
}
```

¹⁰ I sekvenčni i dijagram saradnje čuvaju исту семантику интеракције између објеката. Разлика између њих сеogleда у начину њиховог представљања.

Dijagrami saradnje opisuju интеракцију између објеката у графичком или мрежном формату, у коме објекти могу да се поставе било где на диграму.

Sekvenčni dijagram описује интеракцију у fence(ograda) формату, у коме се сваки следећи објекат додаје десно у односу на предходни објекат.

Navedeni dijagrami имају одређене предности и недостатке:

- **Sekvenčni dijagram:**
 - предност: јасно се приказује секвенаца порука у времену, једноставно описивање.
 - недостатак: када се додаје нови објекат заузима простор у десно.
- **Dijagram saradnje:**
 - предност: једноставније се додају нови објекти на диграму. Болje се илуструју комплексне гране, циклуси и конкурентно понашање.
 - недостатак: Тешкоће код посматранаја секвенце акција. Слоženija notacija.

2. Уговор UG2: *Zapamti*

Операција: *Racunaj(OpstiDomenskiObjekat):signal;*

Веза са СК: *CK1, CK4*

Предуслови: Ако је доменски објекат обрађен или сторниран не може се извршити системска операција. Просто вредносно ограничење над доменским објектом мора бити задовољено.

Постуслови: Запамћен је доменски објекат.

```
class Zapamti extends OpstaSO
{
    public static String Zapamti(OpstiDomenskiObjekat odo)
    { Zapamti r = new Zapamti();
        OpstaSO.transakcija = true;
        return OpstaSO.opstelzvrsenjeSO(odo,r);
    }

    // Prekrivanje методе класе OpstaSO
    boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
    { if (!Preduslov(odo))
        { BBP.dodajPorukuMetode("Sistem ne moze da zapamti " + odo.vratiNazivObjekta() +
        ". " +
                    odo.vratilmeKlase() + " je vec obradjen ili storniran.");
            return false;
        }

        if (!odo.vrednosnaOgranicenja())
        { BBP.dodajPorukuMetode("Sistem ne moze da zapamti " + odo.vratiNazivObjekta() +
        ". " +
                    Naruseno je vrednosno ogranicenje.");
            return false;
        }

        if (!BBP.brisiSlog(odo))
        { BBP.dodajPorukuMetode("Sistem ne moze da zapamti " + odo.vratiNazivObjekta() +
        ". " +
                    return false;
        }

        if (!BBP.pamtiSlozeniSlog(odo))
        { BBP.dodajPorukuMetode("Sistem ne moze da zapamti " + odo.vratiNazivObjekta() +
        ". " +
                    return false;
        }

        BBP.dodajPorukuMetode("Sistem je zapamtio " + odo.vratiNazivObjekta() + ".");
        return true;
    }

    private boolean Preduslov(OpstiDomenskiObjekat odo)
    { if ((BBP.vratiLogickuVrednostAtributa(odo,"Obradjeni") == true) ||
        (BBP.vratiLogickuVrednostAtributa(odo,"Storniran") == true))
        { return false;
        }
        return true;
    }
}
```

3. Уговор UG3: *Obradi*

Операција: *Obradi(OpstiDomenskiObjekat):signal;*

Веза са СК: *CK1, CK4*

Предуслови: Ако је доменски објекат обрађен или сторниран не може се извршити системска операција. Просто вредносно ограничење над доменским објектом мора бити задовољено.

Постуслови: Доменски објекат је обрађен.

```
class Obradi extends OpstaSO
{
    public static String Obradi(OpstiDomenskiObjekat odo)
    { Obradi ob = new Obradi();
        OpstaSO.transakcija = true;
        return OpstaSO.opstelzvrsenjeSO(odo,ob);
    }

    // Prekrivanje методе класе OpstaSO
    boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
    { if (!Preduslov(odo))
        { BBP.dodajPorukuMetode("Sistem ne moze da obradi " + odo.vratiNazivObjekta() + ".
            Dokument je vec obradjen ili storniran.");
            return false;
        }

        if (!odo.vrednosnaOgranicenja())
        { BBP.dodajPorukuMetode("Sistem ne moze da obradi " + odo.vratiNazivObjekta() + ".
            Naruseno je vrednosno ogranicenje.");
            return false;
        }

        odo.Obradi();

        if (!BBP.brisiSlog(odo))
        { BBP.dodajPorukuMetode("Sistem ne moze da obradi " + odo.vratiNazivObjekta() +
            ".");
            return false;
        }

        if (!BBP.pamtiSlozeniSlog(odo))
        { BBP.dodajPorukuMetode("Sistem ne moze da obradi " + odo.vratiNazivObjekta() +
            ".");
            return false;
        }

        BBP.dodajPorukuMetode("Sistem je obradio " + odo.vratiNazivObjekta() + ".");
        return true;
    }

    private boolean Preduslov(OpstiDomenskiObjekat odo)
    { if ((BBP.vratiLogickuVrednostAtributa(odo,"Obradjen") == true) ||
        (BBP.vratiLogickuVrednostAtributa(odo,"Storniran") == true))
        { return false;
        }
        return true;
    }
}
```

4. Уговор UG4: *Pretraži*

Операција: *Pretraži* (*OpstiDomenskiObjekat*):signal;

Веза са СК: CK2, CK3, CK4

Предуслови:

Постуслови:

```
class Pretrazi extends OpstaSO
{
    public static String Pretrazi(OpstiDomenskiObjekat odo)
    { Pretrazi p = new Pretrazi();
        OpstaSO.transakcija = false;
        return OpstaSO.opstelzvrsenjeSO(odo,p);
    }

    // Prekrivanje metode klase OpstaSO
    boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
    { signal = BBP.nadjislogiVratiGa(odo);
        if (!signal)
            { BBP.dodajPorukuMetode("Sistem ne moze da nadje " + odo.vratiNazivObjekta() + " po
zadatoj vrednosti.");
                return false;
            }
        BBP.dodajPorukuMetode("Sistem je nasao " + odo.vratiNazivObjekta() + " po zadatoj
vrednosti.");
        return true;
    }
}
```

5. Уговор UG5: *Storniraj*

Операција: *Storniraj* (*OpstiDomenskiObjekat*):signal;

Веза са СК: CK3

Предуслови: Ако је доменски објекат сторниран не може се извршити системска операција.

Постуслови: Доменски објекат је сторниран

```
class Storniraj extends OpstaSO
{
    public static String Storniraj(OpstiDomenskiObjekat odo)
    { Storniraj st = new Storniraj();
        OpstaSO.transakcija = true;
        return OpstaSO.opstelzvrsenjeSO(odo,st);
    }

    // Prekrivanje metode klase OpstaSO
    boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
    { if (!Preduslov(odo))
        { BBP.dodajPorukuMetode("Sistem ne moze da stornira " + odo.vratiNazivObjekta() + ". " +
odo.vratilimeKlase() + " je vec storniran.");
            return false;
        }
        odo.Storniraj();
        if (!BBP.brisiSlog(odo))
            { BBP.dodajPorukuMetode("Sistem ne moze da stornira " + odo.vratiNazivObjekta() + ".");
                return false;
            }
        if (!BBP.pamtiSlozeniSlog(odo))
            { BBP.dodajPorukuMetode("Sistem ne moze da stornira " + odo.vratiNazivObjekta() + ".");
                return false;
            }
        BBP.dodajPorukuMetode("Sistem je stornirao " + odo.vratiNazivObjekta() + ".");
        return true;
    }
}
```

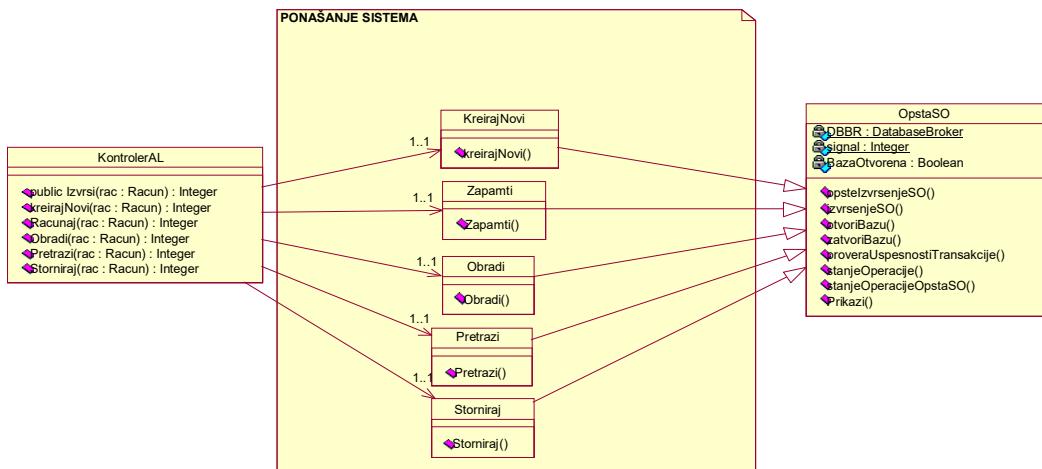
```
private boolean Preduslov(OpstiDomenskiObjekat odo) // 94,77
{ // 1. Ako je racun storniran nad njim se ne moze izvrsiti operacija racunanja stavki.
  if (BBP.vratiLogickuVrednostAtributa(odo, "Storniran") == true)
  { return false; }
  return true;
}
```

Након пројектовања сваке од СО прелази се на пројектовање класе која је одговорна за конекцију са базом и за контролу извршења трансакције. Метода која обезбеђује наведене захтеве се зове *opstelIzvrsenjeSO()*.

Свака од СО треба да наследи класу *OpstaSO* како би могла да се повеже са базом и како би се њено извршење пратило као трансакција:

```
class X extends OpstaSO
{
  public static String X(OpstiDomenskiObjekat odo)
  { X x = new X();
    OpstaSO.transakcija = true;
    return OpstaSO.opstelIzvrsenjeSO(odo,x);
  }
  ..
}
```

Клase које су одговорне за SO наследују класу OpstaSO (Slika OSO).



Slika OSO: Клase које су одговорне за SO наследују класу OpstaSO

Наводимо класу OpstaSO:

```
abstract class OpstaSO
{
  static BrokerBazePodataka BBP;
  static boolean signal;
  static boolean BazaOtvorena = false;
  static boolean transakcija = false;
  // Ova metoda je sinhronizovana kako bi se onemogućilo da 2 ili više klijenata u isto vreme izvršavaju
  // SO koja je pod transakcijom.
```

```
synchronized static String opstizvrsenjeSO(OpstiDomenskiObjekat rac, OpstaSO os)
{ if (!os.otvoriBazu()) return os.vratiPorukuMetode();

if (los.izvrsenjeSO(rac) && transakcija)
{ signal = os.rollbackTransakcije();
return os.vratiPorukuMetode();
}

if (transakcija) os.commitTransakcije();
return os.vratiPorukuMetode();
}

abstract boolean izvrsenjeSO(OpstiDomenskiObjekat rac);

boolean otvoriBazu()
{ if (BazaOtvorena == false)
{ BBP = new BrokerBazePodataka();
BBP.isprazniPoruku();
signal = BBP.otvoriBazu("RACUN");
if (!signal) return false;
}
BBP.isprazniPoruku();
BazaOtvorena = true;
return true;
}

boolean commitTransakcije()
{ return BBP.commitTransakcije();}

boolean rollbackTransakcije()
{ return BBP.rollbackTransakcije();}

String vratiPorukuMetode()
{ System.out.println(BBP.vratiPorukuMetode());
return BBP.vratiPorukuMetode();
}
```

2.3.2.2 Пројектовање структуре софтверског система

На основу концептуалних класа праве се софтверске класе структуре (Слика АССКС).

Концептуалне класе:



Софтверске класе структуре:

```

class Racun
{
    String BrojRacuna;
    String NazivPartnera;
    Double UkupnaVrednost;
    boolean Obradjen;
    boolean Storniran;
    StavkaRacuna []sracun;

    Racun()
    {
        BrojRacuna = "";
        NazivPartnera = "";
        UkupnaVrednost = new Double(0);
        Obradjen = false;
        Storniran = false;
        sracun = null;
    }
}

class StavkaRacuna
{
    Integer RB;
    String SifraProizvoda;
    Integer Kolicina;
    Double ProdajnaCena;
    Double ProdajnaVrednost;
    Racun rac;

    StavkaRacuna(Racun rac1)
    {
        RB = new Integer(0);
        SifraProizvoda = new String("");
        Kolicina = new Integer(0);
        ProdajnaCena = new Double(0);
        ProdajnaVrednost = new Double(0);
        rac = rac1;
    }
}
  
```

2.3.2.3 Брокер базе података

Пре пројектовања датабасе брокера наводи се његова дефиниција и дефиниција свих оних концепата који су потребни да би се јасно схватила комуникација између датабасе брокера и базе података.

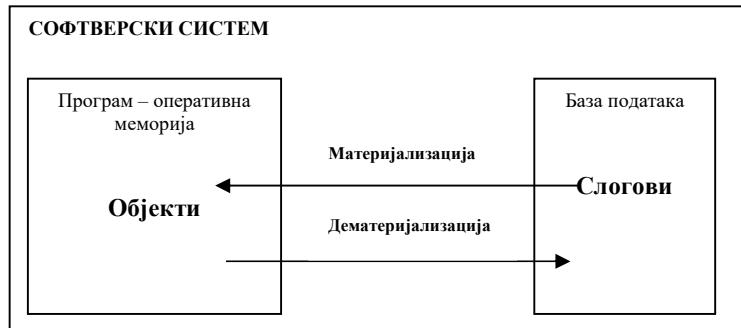
Дефиниција брокера базе података

Деф ПРПО1(Г. Баоц): Објекат је **перзистентан** уколико настави да постоји и након престанка рада програма који га је створио.

Деф ПРПО2: Објекат је **перзистентан** уколико се може **материјализовати** и **дематеријализовати** (Слика МД).

Деф ПРПО3: Материјализација представља процес трансформације слогова из базе података у објекте програма¹¹.

Деф ПРПО4: Дематеријализација (Пасивизација) представља процес трансформације објекта из програма у слогове базе података.



Слика МД: Материјализација и дематеријализација објекта

Деф ПРПО5: Перзистентни оквир је скуп интерфејса и класа који омогућава перзистентност објектима различитих класа (Перзистентни оквир омогућава **перзистентни сервис**¹² објектима). Он се може проширити са новим интерфејсима и класама.

Деф ПРПО6: Перзистентни оквири су засновани на **Холивудском принципу**: “*Don't call us, we'll call you*”. То значи да кориснички дефинисане класе прихватају поруке од предефинисаних класа оквира.

Деф ПРПО7: Уколико у оквиру неке **трансакције**, операције мењају стање перзистентних објекта, оне не чувају то стање одмах у бази података. Уколико се жели запамтити стање које је настало као резултат извршених операција над перзистентним објектима позива се **commit операција**. Уколико се не жели запамтити стање које је настало као резултат извршених операција над перзистентним објектима позива се **rollback операција**. Commit и rollback операције представљају **трансакционе операције**.

Пример брокера базе података

У нашем примеру ми смо пројектовали перзистентни оквир (класа BrokerBazePodataka¹³) који ће да реализује следеће методе:

1. *int otvoriBazu(String imeBaze)*
2. *int commitTransakcije()*
3. *boolean rollbackTransakcije()*
4. *boolean pamtiSlog(OpstiDomenskiObjekat)*
5. *boolean brisiSlog(OpstiDomenskiObjekat)*
6. *boolean promeniSlog(OpstiDomenskiObjekat)*
7. *boolean daLiPostojiSlog(OpstiDomenskiObjekat odo)*
8. *boolean kreirajSlog(OpstiDomenskiObjekat)*
9. *boolean nadjiSlogiVratiGa(Opjekat, Objekat)*
10. *boolean vratiLogickuVrednostAtributa(OpstiDomenskiObjekat odo, String nazivAtributa)*
11. *boolean pamtiSlozeniSlog(OpstiDomenskiObjekat odo)*

¹¹ Термине материјализација и дематеријализација смо објаснили у ужем смислу као процесе који трансформишу објекте програма у слогове базе података. У ширем смислу би се подразумевало да објекти могу бити сачувани у било ком перзистентном складишту података.

¹² Уколико **перзистентни сервис** омогућава памћење објекта у релационој бази података за њега се каже да је он **Object-Relation сервис пресликавања (mapping service)**.

¹³ Брокер базе података [Larman] патерн представља једну могућу реализацију перзистентног оквира. Брокер базе података [Larman] патерн је одговоран за материјализацију, дематеријализацију и кеширање објекта у меморији. Он се често назива и **Database Mapper** патерн.

Dajemo detaljno objašnjenje navedenih metoda.

```
class DatabaseBroker
{
    static Connection con;
    static Statement st;

    /*Уговор DB1: otvoriBazu(String imeBaze) : boolean
    Постуслов: Успостављена је веза (конекција) са базом података. Уколико је успешно остварена веза са базом података метода враћа вредност true и памти поруку: „Uspostavljena je konekcija sa bazom podataka”, иначе метода враћа false и памти поруку у зависности од следећих ситуација:
    а) ако драјвер није учитан метода памти поруку: „Drajver nije ucitan”
    б) ако се десила грешка код конекције метода памти поруку: „Greška kod konekcije”
    с) ако се десила грешка код заштите базе података. метода памти поруку: „Greška zaštite” */

    public boolean otvoriBazu(String imeBaze)
    {
        String Urlbaze;
        try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Urlbaze = "jdbc:odbc:" + imeBaze;
            con = DriverManager.getConnection(Urlbaze);
            con.setAutoCommit(false); // Ako se ovo ne uradi nece moci da se radi rollback.
        } catch(ClassNotFoundException e)
        {
            porukaMetode = "Drajver nije ucitan:" + e; return false;}
        catch(SQLException esql)
        {
            porukaMetode = "Greška kod konekcije:" + esql; return false;}
        catch(SecurityException ese)
        {
            porukaMetode = "Greška zastite:" + ese; return false;}
        porukaMetode = "Uspostavljena je konekcija sa bazom podataka."; return true;
    }

    /*Уговор DB2: commitTransakcije() : boolean
    Постуслов: Све операције које су мењале стање базе, од задњег позива commit или rollback transakcije, су успешно извршене (промене су успешно запамћене у бази). У том случају метода враћа true и памти поруку: „Uspešno uraden commit transakcije”, иначе враћа false и памти поруку: “Nije uspešno urađen commit transakcije”. */

    public boolean commitTransakcije()
    {
        try { con.commit();
            } catch(SQLException esql)
            {
                porukaMetode = porukaMetode + "\nNije uspesno uradjen commit transakcije " + esql;
                return false;
            }
        porukaMetode = porukaMetode + "\nUspesno uradjen commit transakcije ";
        return true;
    }

    /*Уговор DB3: rollbackTransakcije() : boolean
    Постуслов: Ефекти свих операција које су мењале стање базе, од задњег poziva commit или rollback трансакције, су поништени (промене нису запамћене у бази). У том случају метода враћа true и поруку: “Uspesno uradjen rollback transakcije”, иначе враћа false и поруку: „Nije uspešno uradjen rollback transakcije“.

    */

    public boolean rollbackTransakcije()
    {
        try{ con.rollback();
            } catch(SQLException esql)
            {
                porukaMetode = porukaMetode + "\nNije uspesno uradjen rollback transakcije" + esql;
                return false;
            }
        porukaMetode = porukaMetode + "\nUspesno uradjen rollback transakcije";
        return true;
    }
```

/*Уговор DB4: pamtiSlog(OpstiDomenskiObjekat): boolean

Постуслов: Извршено је памћење доменског објекта у базу података (материјализација). Уколико је метода успешно извршена враћа труе и памти поруку: "Успешно запамћен slog u bazi", иначе враћа false и памти поруку: „Nije uspešno zapamćen slog u bazi”.

Напомена: Наведена метода је генеричка јер омогућава памћење објекта било које класе у бази, уколико те класе наслеђује класу *OpstiDomenskiObjekat*.

```
public boolean pamtiSlog(OpstiDomenskiObjekat odo)
{
    String upit;
    try{ st = con.createStatement();
        upit = "INSERT INTO " + odo.vratilmeKlase() +
               " VALUES (" + odo.vratiVrednostiAtributa() + ")";
        st.executeUpdate(upit);
        st.close();
    } catch(SQLException esql)
    {
        porukaMetode = porukaMetode + "\nNije uspesno zapamcen slog u bazi. " + esql;
        return false;
    }
    porukaMetode = porukaMetode + "\nUspesno zapamcen slog u bazi. ";
    return true;
}
```

ОБЈАШЊЕЊЕ ПОСТУПКА ПРОЈЕКТОВАЊА ГЕНЕРИЧКЕ МЕТОДЕ

Уколико би метода била специфично везана за конкретну доменску класу нпр. Račun она би имала следећи изглед (болдовали смо зависне делове методе од доменске класе) :

```
public boolean pamtiSlog(Racun rac)
{
    String upit;
    try{ st = con.createStatement();
        upit = "INSERT INTO " + rac.vratilmeKlaseRacun() +
               " VALUES (" + rac.vratiVrednostiAtributaRacun() + ")";
        st.executeUpdate(upit);
        st.close();
    } catch(SQLException esql)
    {
        porukaMetode = porukaMetode + "\nNije uspesno zapamcen slog u bazi. " + esql;
        return false;
    }
    porukaMetode = porukaMetode + "\nUspesno zapamcen slog u bazi. ";
    return true;
}

class Racun
{
...
public String vratiVrednostiAtributaRacuna()
{
    return ""+ BrojRacuna + ", " + NazivPartnera + ", " + UkupnaVrednost.doubleValue() + ", " + Obradjeni + ", " +
           Storniran;
}
public String vratilmeKlaseRacun() { return "Racun";}
```

Уколико би користили наведени приступ морали би за сваку доменску класу да имплементирају методу *pamtiSLog()*, као и све остале методе које ћемо ниже навести (*brisniSlog()*, *promeniSlog()*,...) у класи BrokerBazePodataka. То није практично јер би број операција BrokerBazePodataka растао са појавом нових доменских класа. Због тога смо сваку од ниже наведених метода пројектовали као генеричку методу. На примеру методе *pamtiSlog()* објаснићемо општи поступак за пројектовање било које од ниже наведених генеричких метода.

Поступак пројектовања генеричке методе:

1. Одредити специфичне класе (*Racun*) у методи (*PamtiSlog*) која треба да постане генерална. Именовати специфичне класе у општем смислу (нпр. класу Račun са класом *OpstiDomenskiObjekat*). Такође методе специфичних класа именовати у општем смислу, ако је њихов назив повезан са нечим што је специфично за класу којој оне припадају (нпр. назив методе *vratilmeKlaseRacun()* са називом *VratilmeKlase()*).

```
public boolean pamtiSlog(OpstiDomenskiObjekat odo)
{
    String upit;
    try{ st = con.createStatement();
        upit = "INSERT INTO " + odo.vratilmeKlase() +
               " VALUES (" + odo.vratiVrednostiAtributa() + ");
```

```

        st.executeUpdate(upit);
        st.close();
    } catch(SQLException esql)
    {
        porukaMetode = porukaMetode + "\nNije uspesno zapamcen slog u bazi. " + esql;
        return false;
    }
    porukaMetode = porukaMetode + "\nUspesno zapamcen slog u bazi. ";
    return true;
}

```

2. Направити генералну класу (апстрактну класу или интерфејс) за специфичне класе.

```

interface OpstiDomenskiObjekat
{
    vratIvrednostiAtributa();
    vratIimeKlase();
}

```

Из наведеног може да се закључи да метода *pamtiSlog()*, може да прихвати различите доменске објекте преко параметра, ако доменски објекти наследе класу *OpstiDomenskiObjekat* и имплементирају њене методе *vratIzraz()* и *vratImeKlase()*.

У том смислу доменска класа Раџун и СтавкаРачуна ће добити следећи изглед:

```

class Racun implements OpstiDomenskiObjekat
{
    ...
    public String vratIvrednostiAtributa()
    {
        return ""+ BrojRacuna + ", " + NazivPartnera + ", " + UkupnaVrednost.doubleValue() + ", " + Obradjeni + ", " +
               Storniran;
    }
    public String vratIimeKlase()
    {
        return "Racun";
    }
}

class StavkaRacuna implements OpstiDomenskiObjekat
{
    ...
    public String vratIvrednostiAtributa()
    {
        return "" + rac.BrojRacuna + ", " + RB.intValue() + ", " + SifraProizvoda + ", " + Kolicina.intValue() + ", " +
               ProdajnaCena.doubleValue() + ", " + ProdajnaVrednost.doubleValue();
    }
    public String vratIimeKlase() { return "StavkaRacuna"; }
}

```

Код доменских класа смо болдовали све делове програма који су промењени како би они могли да буду прихваћени као параметар методе *pamtiSlog()*.

КРАЈ ОБЈАШЊЕЊА ПОСТУПКА ПРОЈЕКТОВАЊА ГЕНЕРИЧКЕ МЕТОДЕ

*/*Уговор DB5: brisiSlog(OpstiDomenskiObjekat odo) : boolean*

Постуслов: Обрисан је слог у бази података (такав објекат се не може више материјализовати). Уколико је метода успешно извршена враћа true и поруку: „Uspešno obrisan slog u bazi.“, иначе враћа false и памти поруку: “Nije uspešno obrisan slog u bazi.”.

Напомена: Наведена метода је генерицка јер омогућава да се преко ње мозе обрисати објекат било које класе која наслеђује класу *OpstiDomenskiObjekat*.

/*

```

public boolean brisiSlog(OpstiDomenskiObjekat odo)
{
    String upit;
    try { st = con.createStatement();
        upit ="DELETE * FROM " + odo.vratIimeKlase() + " WHERE " + odo.vratIUslovZaNadjiSlog();
        st.executeUpdate(upit);
        st.close();
    } catch(SQLException esql)
    {
        porukaMetode = porukaMetode + "\nNije uspesno obrisan slog u bazi: " + esql;
        return false;
    }
    porukaMetode = porukaMetode + "\nUspesno obrisan slog u bazi.";
    return true;
}

```

/*Уговор DB6: **promeniSlog**(OpstiDomenskiObjekat odo) : boolean
Постуслов: Промењен је слог у бази података по задатом услову. Уколико је метода успешно извршена враћа true и памти поруку: "Uspešno promenjen slog u bazi podataka", иначе враћа false и памти поруку: "Nije uspešno promenjen slog u bazi podataka".

Напомена: Наведена метода је генерицка јер омогућава промену објекта било које класе која наслеђује класу *OpstiDomenskiObjekat*. */

```
public boolean promeniSlog(OpstiDomenskiObjekat odo)
{ String upit;
try { st = con.createStatement();
upit = "UPDATE " + odo.vratiImeKlase() +
" SET " + odo.postaviVrednostiAtributa() +
" WHERE " + odo.vratiUslovZaNadjiSlog();
System.out.println("PROMENI SLOG" + upit);
st.executeUpdate(upit);
st.close();
} catch(SQLException esql)
{ porukaMetode = porukaMetode + "\nNije uspesno promenjen slog u bazi podataka: " +
esql;
return false;
}
porukaMetode = porukaMetode + "\nUspesno promenjen slog u bazi podataka: ";
return true;
}
```

/*Уговор DB7: **daLiPostojiSlog**(OpstiDomenskiObjekat odo) : boolean

Постуслов: Уколико постоји слог у бази података метода враћа true и памти поруку: "Slog postoji u bazi podataka." иначе враћа false и памти поруку: „Slog postoji u bazi podataka.“

Напомена: Наведена метода је генерицка јер омогућава претраживање објекта било које класе која наслеђује класу *OpstiDomenskiObjekat*. */

```
public boolean daLiPostojiSlog(OpstiDomenskiObjekat odo)
{ String upit;
ResultSet RSslogovi;
try { st = con.createStatement();
upit = "SELECT *" +
" FROM " + odo.vratiImeKlase() +
" WHERE " + odo.vratiUslovZaNadjiSlog();
RSslogovi = st.executeQuery(upit);
boolean signal = RSslogovi.next();
RSslogovi.close();
st.close();
if (signal == false)
{ porukaMetode = porukaMetode + "\nSlog ne postoji u bazi podataka. ";
return false; // Slog ne postoji u bazi.
}
} catch(SQLException esql)
{ porukaMetode = porukaMetode + "\nNije uspesno pretrazena baza: " + esql;
return false;
}
porukaMetode = porukaMetode + "\nSlog postoji u bazi. ";
return true;
}
```

/*Уговор DB8: **kreirajSlog**(OpstiDomenskiObjekat odo): boolean

Постуслов: Креира нови слог у бази података метода. Уколико је метода успешно враћа true и памти поруку: "Slog postoji u bazi podataka." иначе враћа false и памти поруку: „Slog postoji u bazi podataka.“

Напомена: Наведена метода је генерицка јер омогућава креирање објекта било које класе која наслеђује класу *OpstiDomenskiObjekat*. */

```
public boolean kreirajSlog(OpstiDomenskiObjekat odo)
{ String upit;
ResultSet rs;
upit = "SELECT Max(" + odo.vratiAtributPretrazivanja() + ") AS Max" +
" FROM " + odo.vratiImeKlase();
try { st = con.createStatement();
```

```

rs = st.executeQuery(upit);

if (rs.next() == false)
    odo.postaviPocetniBroj();
else
    odo.povecajBroj(rs);

upit = "INSERT INTO " + odo.vratilmeKlase() +
       " VALUES (" + odo.vratiVrednostiAtributa() + ")";
st.executeUpdate(upit);
st.close();
} catch(SQLException esql)
{
    porukaMetode = porukaMetode + "\nNe moze da se kreira novi slog: " + esql;
    return false;
}
porukaMetode = porukaMetode + "\nKreiran je novi slog: ";
return true;
}

/*Уговор DB9: nadjiSlogiVratiGa(OpstiDomenskiObjekat odo, OpstiDomenskiObjekat odo1) : integer
*/
public boolean nadjiSlogiVratiGa(OpstiDomenskiObjekat odo)
{
    ResultSet RS;
    String nazivVezanogObjekta;
    int brojStavki;
    String upit;
    Statement st;
    try {
        st= con.createStatement(resultSet.TYPE_SCROLL_INSENSITIVE,resultSet.CONCUR_READ_ONLY);
        upit = "SELECT *" + " FROM " + odo.vratilmeKlase() +
               " WHERE " + odo.vratiUslovZaNadjiSlog();
        RS = st.executeQuery(upit);
        System.out.println("Upit-nadji: " + upit);
        boolean signal = RS.next();
        if (signal == false)
            { porukaMetode = porukaMetode + "\nNe postoji slog u bazi podataka.";
              return false;
            }
        porukaMetode = porukaMetode + "\nUspesno je procitan slog iz baze podataka.";

        if (odo.Napuni(RS)) // if
        { for (int j=0;j<odo.vratiBrojVezanihObjekata();j++)
          { OpstiDomenskiObjekat vez = odo.vratiVezaniObjekat(j);
            if (vez == null)
                { porukaMetode = porukaMetode + "\nNe postoji vezani objekat a navedeno je
                  da postoji.";
                  return false;
                }
            else // else1
                { upit = "SELECT COUNT(*) as brojStavki" + " FROM " + vez.vratilmeKlase() +
                  " WHERE " + vez.vratiUslovZaNadjiSlogove();
                  RS = st.executeQuery(upit);
                  if (RS.next() == false)
                      { porukaMetode = porukaMetode + "\nNe postoje slogovi vezanog objekta";
                        return true;
                      }
                  brojStavki = RS.getInt("brojStavki");
                  odo.kreirajVezaniObjekat(brojStavki,j);
                  upit = "SELECT *" + " FROM " + vez.vratilmeKlase() +
                         " WHERE " + vez.vratiUslovZaNadjiSlogove();
                  RS = st.executeQuery(upit);
                  int brojSloga = 0;
                  while(RS.next())
                      { odo.Napuni(RS,brojSloga,j);
                        brojSloga++;
                      }
                }
            }
        }
    }
}

```

```

        }
        porukaMetode = porukaMetode + "\nUspesno su procitani slogovi vezanog objekta";
    } // end else1
} // end for
} // end if1
RS.close();
st.close();
} catch(Exception e)
{
    porukaMetode = porukaMetode + "\nGreska kod citanja sloga iz baze podataka." + e;
    return false;
}
return true;
}/*
*/
/*Уговор DB10: vradiLogickuVrednostAtributa(OpstiDomenskiObjekat odo, String nazivAtributa) : boolean
*/
public boolean vradiLogickuVrednostAtributa(OpstiDomenskiObjekat odo, String nazivAtributa)
{
    String upit;
    ResultSet rs;
    boolean s = false;
    try { st = con.createStatement();
        upit = " SELECT " +
            " FROM " + odo.vratiImeKlase() +
            " WHERE " + odo.vratiUslovZaNadjiSlog();
        System.out.println("upit: " + upit);
        rs = st.executeQuery(upit);
        rs.next();
        s = KonverterTipova.Konvertuj(rs, s, nazivAtributa);
        rs.close();
        st.close();
    } catch(Exception e)
    {
        porukaMetode = porukaMetode + "\nGreska kod citanja logicke vrednosti atributa sloga." + e;
        return false;
    }
    return s;
}

```

Уговор DB11: pamtiSlozeniSlog(OpstiDomenskiObjekat odo): boolean

```

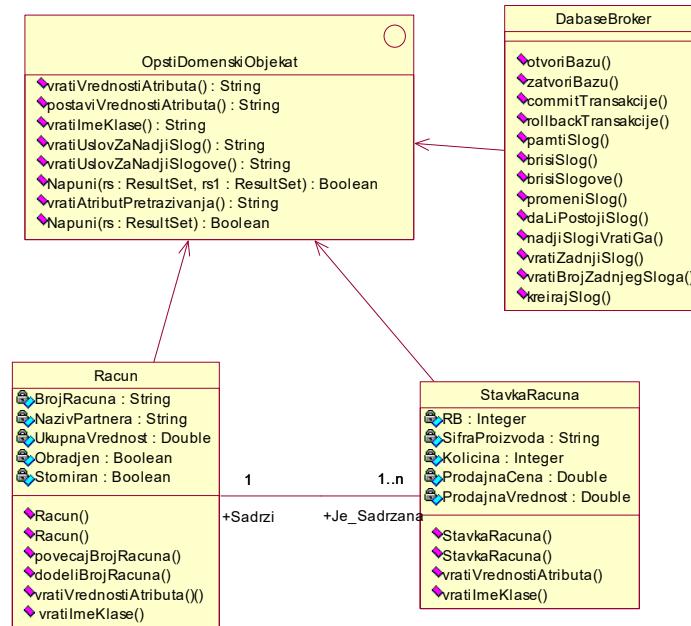
public boolean pamtiSlozeniSlog(OpstiDomenskiObjekat odo)
{
    String upit;
    try { st = con.createStatement();
        upit = " INSERT INTO " + odo.vratiImeKlase() +
            " VALUES (" + odo.vratiVrednostiAtributa() + ")";
        st.executeUpdate(upit);
        for(int j=0;j<odo.vratiBrojVezanihObjekata();j++)
        {
            OpstiDomenskiObjekat vez;
            for(int i=0; i < odo.vratiBrojSlogovaVezanogObjekta(j);i++)
            {
                vez = odo.vratiSlogVezanogObjekta(j,i);
                upit = " INSERT INTO " + vez.vratiImeKlase() +
                    " VALUES (" + vez.vratiVrednostiAtributa() + ")";
                st.executeUpdate(upit);
            }
        }
        st.close();
    } catch(SQLException esql)
    {
        porukaMetode = porukaMetode + "\nNije zapamcen slozeni slog: " + esql;
        return false;
    }
    porukaMetode = porukaMetode + "\nZapamcen je slozeni slog.";
    return true;
}

```

U procesu pravljenja generičkih metoda DatabaseBroker klase dobili smo metode interfejsa OpstiDomenskiObjekat:
// Operacije navedenog interfejsa je potrebno da implementira svaka od domenskih klasa,
// koja zeli da joj bude omogucena komunikacija sa Database broker klasom.

```
interface OpstiDomenskiObjekat
{ String vratiVrednostiAtributa();
String postaviVrednostiAtributa();
String vratiImeKlase();
String vratiUslovZaNadjiSlogove();
String vratiUslovZaNadjiSlogove();
boolean Napuni	ResultSet rs, ResultSet rs1);
String vratiAtributPretrazivanja();
boolean Napuni	ResultSet rs);
}
```

Kao rezultat projektovanja klase DatabaseBroker i interfejsa OpstiDomenskiObjekat dobijaci se sledeći dijagrami klasa (Slika DBBR, ASSDBBR)



Slika DBBR: Database broker klasa se povezuje sa klasom OpstiDomenskiObjekat

2.3.3 Пројектовање складишта података

На основу софтверских класа структуре пројектовали смо табеле (складишта података) релационог система за управљање базом података (Слика АССТБП).

Table: Racun
Columns

| Name | Type | Size |
|----------------|-----------------|------|
| BrojRacuna | Text | 10 |
| NazivPartnera | Text | 50 |
| UkupnaVrednost | Number (Double) | 8 |
| Obradjen | Yes/No | 1 |
| Storniran | Yes/No | 1 |

Table Indexes

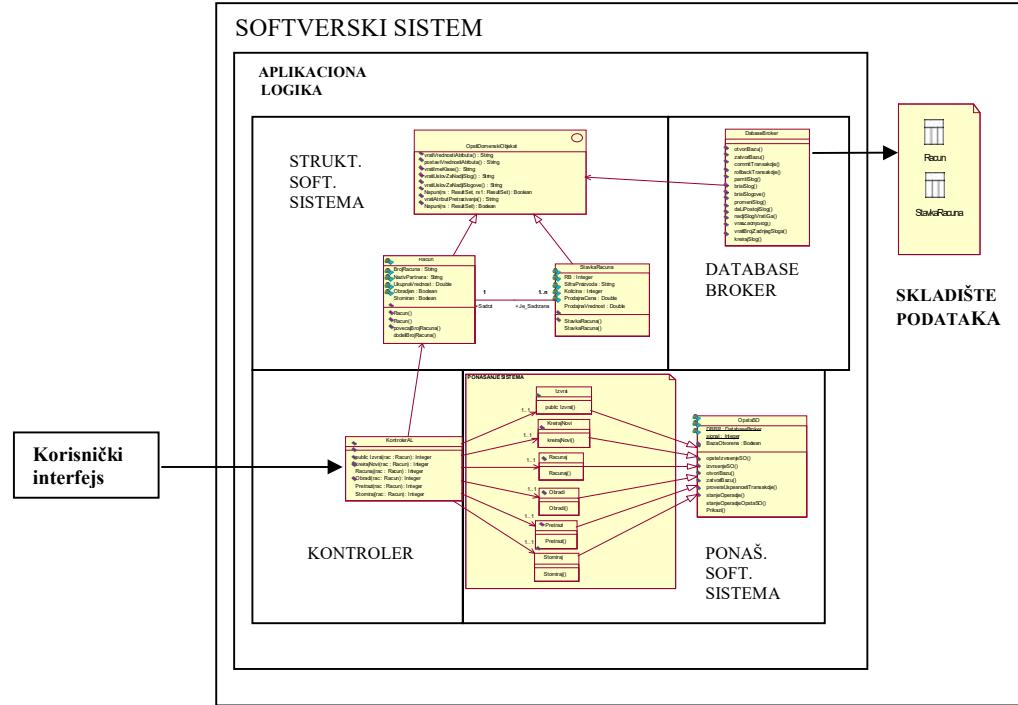
| Name | Number of Fields |
|-----------------------|----------------------------|
| PrimaryKey Fields: | 1 BrojRacuna, Ascending |

Table: StavkaRacunaColumns

| Name | Type | Size |
|------------------|------------------|------|
| BrojRacuna | Text | 10 |
| RB | Number (Integer) | 2 |
| SifraProizvoda | Text | 20 |
| Kolicina | Number (Integer) | 2 |
| ProdajnaCena | Number (Double) | 8 |
| ProdajnaVrednost | Number (Double) | 8 |

Table Indexes

| Name | Number of Fields |
|-----------------------|---|
| PrimaryKey Fields: | 2 BrojRacuna, Ascending RB, Ascending |



Слика АССТБП: Архитектура софт. система након пројектовања табела базе података

IV ДЕО – РЕЛАЦИОНИ УПИТНИ ЈЕЗИК

(Прављење табела и постављање упита у језику Transact-SQL)

Саша Д. Лазаревић

САДРЖАЈ

| | |
|--|-----------|
| 1. МОДЕЛ ПОДАТАКА | 3 |
| 2. БАЗА ПОДАТКА. СИСТЕМ ЗА УПРАВЉАЊЕ БАЗАМА ПОДАТКА | 5 |
| 3. РЕЛАЦИОНИ МОДЕЛ ПОДАТАКА | 6 |
| 3.1. Концепција релационог модела података | 7 |
| 3.1.1. Независност: логичка и физичка | 7 |
| 3.1.2. Једноставност: табела као једина структура података | 7 |
| 3.1.3. Декларативност: упитни језик | 7 |
| 3.2. Структурална компонента релационог модела података | 7 |
| 3.2.1. Релација | 7 |
| 3.2.2. Домени атрибута | 9 |
| 3.2.3. Кључеви релације | 9 |
| 3.2.4. Схема релационе базе и релациони база | 10 |
| 3.2.5. NULL вредност | 11 |
| 3.3. Операциона компонента релационог модела података | 12 |
| 3.4. Интегритетна компонента релационог модела података | 12 |
| 3.4.1. Ограничења | 12 |
| 3.4.2. Правила интегритета | 14 |
| 3.5. Кодова релациониа правила | 16 |
| 4. РЕЛАЦИОНИ УПИТНИ ЈЕЗИК: STRUCTURED QUERY LANGUAGE (SQL)..... | 19 |
| 4.1. Врсте SQL наредби | 19 |
| 4.2. SQL-SCHEMA STATEMENTS | 20 |
| 4.2.1. SQL типови података | 20 |
| 4.2.2. Домени | 20 |
| 4.2.3. Табеле и колоне | 21 |
| 4.2.4. Индекси | 21 |
| 4.2.5. Схеме | 22 |
| 4.3. SQL-DATA-MANIPULATION STATEMENTS | 22 |
| 4.4. SQL-DATA-RETRIEVE STATEMENTS | 25 |
| 4.4.1. Поступак извршавања SQL упита | 25 |
| 4.4.2. Синтакса наредбе SELECT | 26 |
| 5. РЕФЕРЕНЦЕ | 27 |
| ЛИТЕРАТУРА | 27 |
| ИНТЕРНЕТ | 27 |
| 6. ПРИЛОГ | 28 |
| 6.1. Концептуализација | 28 |
| 6.1.1. Вербални модел | 28 |
| 6.1.2. Концептуални модел | 28 |
| 6.1.2.1. Структура | 28 |
| 6.1.2.2. Ограничења | 29 |
| 6.2. Спецификација | 30 |
| 6.2.1. Схема релационе базе података: базне релације | 30 |
| 6.2.2. Схема релационе базе података: изведене релације | 30 |
| 6.2.3. Правила интегритета | 30 |
| 6.3. Имплементација | 30 |
| 6.3.1. DDL | 30 |
| 6.3.2. DML | 32 |
| 6.3.3. RDB | 33 |
| 6.3.4. ADS | 34 |
| -- 1. УПИТИ НАД ЈЕДНОМ ТАБЕЛОМ КОЈИМА СЕ ПРИКАЗУЈЕ ЊЕН НЕИЗМЕЊЕН САДРЖАЈ | 37 |
| -- 2. УПИТИ НАД ЈЕДНОМ ТАБЕЛОМ КОЈИМА СЕ ПРИКАЗУЈЕ ЊЕН ИЗМЕЊЕН САДРЖАЈ | 48 |
| -- 3. УПИТИ НАД ДВЕ ИЛИ ВИШЕ ТАБЕЛА | 59 |

1. Модел података

Апликативни софтвер је рачунарско¹ решење проблема из реалног света; он је (на рачунару извршив) модел разматраног система. За спецификацију и имплементацију тог модела потребни су одређени формални концепти и механизми. У знаменитом спису Едгара Ф. Кода (Edgar F. Codd) *Data Models in Database Management*² први пут је уведен појам **модела података** као формалног система који се користи за спецификацију и имплементацију модела разматраног система³. Циљ модела података је да омогући једнозначну и једноставну спецификацију и имплементацију:

- Ентитета из разматраног система (конструкција, ажурирање, деструкција),
- Њихових узајамних релација (повезивање, превезивање, развезивање) и
- Дозвољених трансформација (поступци и правила промене стања, начини извођења нових ентитета из постојећих).

Сваки модел података састоји се од⁴:

- (1) Структурне компоненте – колекција концепата за дефинисање структуре система (стања система). Поред основних концепата, структурна компонента садржи и конструктор(е) нових, сложених концепата. Ова компонента модела података служи за опис статичких особина разматраног система, које су споро или ретко изменљиве, те зато релативно независне од времена (и простора). Концепти структурне компоненте пресликавају се у структуру будуће базе података.

Пример. У пекарству основне концепте представљају: брашно, квасац, со, вода, млеко, јаја, шећер, цем... Од њих се граде сложенији концепти као што су: тесто, надев, прелив...

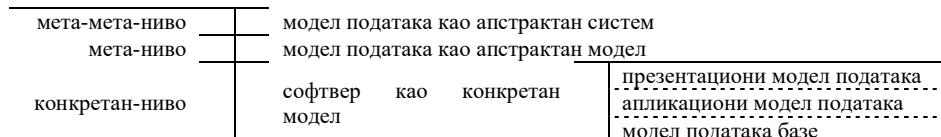
Пример. У програмирању основне концепте представљају домен, тип, променљива (варијабла), вредност, скалар, вектор, матрица, листа...

- (2) Операционе компоненте – колекција оператора (тј. операција) за манипулацију концептима структуре (промена стања, приказ стања). Ова компонента модела података служи за опис динамичких особина разматраног система, којима се исказује начин рада и промене стања разматраног система. Оператори ове компоненте модела података пресликавају се у операције над базом података. Наиме, сваки систем током постојања подложен је променама стања. Те промене се у моделу (тј. у бази података) исказују путем измене података о концептима структурне компоненте. Да би база података представљала верну слику разматраног система, морају се дефинисати операције које мењају садржај базе података и доводе га у сагласност са стварним стањем разматраног система. Секвенце тих операција чине:
 - (i) програме за ажурирање који мењају садржај базе података сагласно променама у разматраном систему и
 - (ii) програме за извештавање који приказују садржај базе података, тј. тренутно стање система.

¹ Рачунар = хардвер + системски софтвер

² Први пут објављеном у *Proceedings of Workshop on Data Abstraction, Databases, and Conceptual Modelling* (Michael L. Brodie and Stephen N. Yilles, eds.), Pingree Park, Colo., June 1980. Касније више пута објављиван, са мањим изменама, у различитим часописима.

³ Извесну недоумицу може да изазове несрћено изабран израз **модел података**, јер он, у зависности од контекста у којем се нађе, представља бар два појма: први, то је **апстрактни систем** који обезбеђује фундаменталне концепте за моделовање било ког система, и други, то је модел којим је описан разматрани систем (тј. модел података разматраног система), а настало је употребом концепата дефинисаних у апстрактном систему; зато се овај модел може назвати **апстрактни модел** разматраног система. Ту није крај могућим недоумицама, јер се исти израз користи у оквиру софтвера (који је једини оперативни, тј. **конкретни модел**), где се прави разлика између **презентационог модела података**, **апликационог модела података** и **модела података базе**. Све у свему, да би се избегле недоумице, потребно је уочити три међусобно повезана, али посебна нивоа:



⁴ Lazarević Branislav i dr., *Baze podataka*, FON, Beograd, 2003. Стр. 14-22, 39-44 и 88-93.

Програми представљају имплементацију процеса који се извршавају у разматраном систему.

Пример. У пекарству операције могу бити: направи_тесто, меси_тесто, посоли_тесто, одмери_тесто_за_векну_хлеба, обликуј_векну_хлеба, испечи_хлеб... Од наведених операција могу се направити процеси: направи_бели_хлеб, направи_црни_хлеб...

Пример. У програмирању постоје аритметички оператори (сабирање, одузимање, множење, дељење), логички оператори (конјункција, дисјункција, негација), скуповни оператори (припадност скупу), стринг оператори (конкатенација)... Они се могу користити за реализацију операција сабирања, одузимања, множења...

- (3) Интегритетне компоненте – служи за утврђивање валидних података у моделу података. Састоји се од две целине: колекције ограничења и колекције правила интегритета.

Колекцијом ограничења се искључују недозвољене вредности концепата структуре, односно обезбеђује појављивање само дозвољених вредности (конзистентност⁵ стања система). Деле се на:

- (i) ограничења модела података (ОМП) – општа ограничења која су својствена за сам модел података,
- (ii) ограничења разматраног система (тзв. пословна правила) – посебна ограничења која су својствена само за разматрани систем:
 - ограничења стања система (ограничења на могуће вредности података у бази) (ОСС),
 - ограничења промене стања, тј. прелаза из једног стања у друго стање система (ограничења на могуће промене вредности података у бази) (ОПС);

Следећи корак је дефинисање правилá интегритета која дефинишу основну динамику⁶ система. Свако правило интегритета чини тројка која се састоји од *операције* која се извршава (којим се мења стање система), *ограничења* које мора бити уважено (да би се очувала конзистентност стања) и *акције* која се предузима уколико је дошло до нарушувања ограничења (а којом се систем преводи или враћа у конзистентно стање). Понекада се правилу интегритета додаје и четврти члан: *тренутак* испитивања ограничења, који може да буде: (1) *пре* извршења операције или (2) *после* извршења операције. Формално, правило интегритета је уређена четворка⁷:

$$\text{ПИ} = \langle \text{Операција}, \text{Ограничение}, \text{Акција} [, \text{Тренутак}] \rangle.$$

Ограничение представљају основу за специфицирање правила интегритета, која се потом пресликавају у услове интегритета⁸ будуће базе података.

Пример. У пекарству ограничења могу бити:

- О-1: Датум употребе квасца мора да буде пре датума истека његовог рока трајања. (ОСС)
- О-2: За пшенични бели хлеб користи се оштро бело брашно, тип 500, добијено од тврде пшенице. (ОСС)
- О-3: За пшенични црни хлеб користи се оштро црно брашно, тип 1100, добијено од тврде пшенице. (ОСС)
- О-4: Векна белог хлеба типа „Сава“ тешка је 600 гр. (ОСС)
- О-5: У све дуготрајне хлебове додаје се адитив АЕ-306 у следећем односу: на 10 кг теста додаје се 3 мл адитива. (ОСС)
- О-6: Хлеб за отпис је сваки краткотрајни хлеб који је старији од три дана. (ОПС)

Пример. У пекарству правила интегритета могу бити:

- ПИ-1: <направи_тесто, О-1, А-1, Пре>
- ПИ-2: <одмери_тесто_за_векну_хлеба, О-4, А-2, После>

⁵ Конзистентан = непротивречан, доследан, у складу са. У нашем контексту, то значи да се у бази података могу појавити само они подаци који су у складу са чињеничним стањем разматраног система.

⁶ Односно “понашање” система.

⁷ Средње заграде указују на опциону компоненту. Уколико је ова компонента изостављена, тренутак испитивања ограничења је после извршења операције.

⁸ Услов интегритета је механизам којим се у бази података обезбеђује да подаци о стању разматраног система буду усаглашени са: (1) ограничењима разматраног система (који су семантичког карактера) и (2) ограничењима самог модела података (који су техничког карактера).

Где су акције:

А-1: Одби операцију и пријави надређеном разлог.

А-2: Изврши исправку тежине теста.

Модел података је основа за развој софтвера, а посебно базе података, која је (врло често) најважнији део сваког софтвера. Формално-логички он је аксиоматски систем у којем се могу формулисати разне хипотезе и теорије о разматраном систему; информатичко-програмерски он је (семантички богат) језик за моделовање којим се могу конструисати рачунарски извршиви модели, тј. софтвери. Формално, модел података је уређена тројка:

$$МП = < Ст, Оп, Ин >$$

где су:

Ст – структурна комонента,

Оп – операциона компонента и

Ин – интегритетна компонента.

2. База података. Систем за управљање базама података

База података (database, DB) је збирка узајамно повезаних података, који су трајно меморисани (похрањени) са контролисаном редундансом⁹, да би оптимално служила различитим апликацијама¹⁰. Подаци су меморисани одвојено и независно од апликационих програма који их користе. За додавање нових података и модификовање или претраживање постојећих података користе се заједнички или контролисани приступи. Са аспекта (физичке) организације података, база података је скуп интегрисаних датотека којима апликације приступају на контролисан начин, преко система за управљање базама података (СУБП). Са архитектуралног становишта, постоје три начина представљања једне базе података:

- екстерна репрезентација – састоји се из скupa апликационих подсхема¹¹ (нпр. погледи) или апликационог интерфејса¹² (нпр. ускладиште процедуре и функције); садржи подмоделе података приложођене корисницима базе¹³, а опционо и скуп операција које се могу извршити над тим подацима; скуп логичких подмодела базе података; ова репрезентација базе података је делимична¹⁴ и привремена¹⁵;
- концептуална репрезентација – логичка схема целокупне базе података, општа логичка структура базе података која приказује концептуални модел разматраног система; логички модел базе података; ова репрезентација базе података је целовита¹⁶ и привремена¹⁷;
- интерна репрезентација – физичка структура базе података (битови и бајтови, физички слогови и датотеке, индекси и кластери, блокови, стазе, плоче, цилиндри и дискови), физички модел базе података који приказује начин похрањивања и приступа подацима на екстерној меморији; ова репрезентација базе података је целовита и трајна¹⁸ и једина стварно физички постојећа.

⁹ Вишеструко понављање истих података.

¹⁰ Апликациони програм (краће: апликација) је рачунарски програм који се користи за одређену врсту послова и служи за непосредно задовољење коначних потреба крајњег корисника; то је програм који је написан да реши одређени проблем, произведе одређени извештај или ажурира одређене податке. На пример: (1) програм који креира платни списак је апликација која обрађује финансијске податке о платама запослених; (2) програм за обраду текста (нпр. *MS Word*) је апликација за унос, обликовање и штампање текста. Насупрот апликацијама, оперативни систем, драјвер или компајлер могу бити од суштинског значаја за ефикасно коришћење рачунарског система, али ни један од наведених програма не учествује у непосредном задовољавању коначних потреба крајњих корисника.

¹¹ Опис података који су апликацији расположиви за коришћење.

¹² Интерфејс за програмирање апликација (Application Programming Interface, API); опис процедуралних и функцијских потпрограма који су апликацији расположиви за коришћење.

¹³ Апликациони програмери, аналитичари података, крајњи корисници.

¹⁴ Приказује само део базе података.

¹⁵ Траје само док траје апликациони програм који је користи.

¹⁶ Приказује целу базу података.

¹⁷ Траје само док се извршава СУБП (и који двосмерно пресликава физички модел БП у концептуални модел БП).

¹⁸ Постоји док постоји и меморијски медијум на којем су подаци записани у бинарном облику.

Систем за управљање базама података (database management system, DBMS) је програмски производ (софтвер) који је посредник између апликационих програма и крајњих корисника, са једне стране, и записа базе података на екстерној меморији, са друге стране. Обично се заснива на неком теоријском моделу података и у складу са њим¹⁹:

- 1) омогућава ефикасно креирање, коришћење и мењање базе података:
 - дефинисање структуре, операција и правила интегритета базе података,
 - селекцију и модификацију (упис, промена и брисање) садржаја базе података;
- 2) поседује механизме за:
 - управљање трансакцијама,
 - заштиту од неовлашћеног приступа подацима,
 - заштиту од уништења података,
 - обезбеђења ефикасног коришћења базе података,
 - управљање дистрибуираним деловима базе података.

Систем база података (database system, DBS) = Систем за управљање базама података + Базе података

3. Релациони модел података

У другој половини шездесетих година прошлог века појавили су се мрежни и хијерархијски системи за управљање базама података (Database Management System, DBMS). После почетних великих очекивања, већ почетком седамдесетих година, уочени су недостаци ових система за управљање базама података²⁰:

- недовољна развојеност логичког од физичког аспекта базе података,
- комплексне структуре података,
- коришћење процедуралних и навигационих језика.

Настојање да се реше ови недостаци имало је за последицу дефинисање релационог модела података²¹. Данас је релациони модел (РМ) најпопуларнији модел података и основа за имплементацију најзначајнијих комерцијалних СУБП. Његове битне особине су:

- једноставност - једина структура података у РМ је табела; операције над табелама су једноставне и дају за резултат нову табелу (хомогеност);
- декларативност – једноставна структура и једноставне операције добра су основа за дефинисање декларативног релационог језика²²; такође, највећи број ограничења је могуће реализовати декларативно;
- фундираност – целокупан РМ је формално-математички заснован: одређене врсте табела се могу третирати као математичке релације и искористити добро познати формализми за развој релационог језика, релационих база и на њима заснованог софтвера; тиме се успоставља општи теоријски и методолошки оквир који развој софтвера преводи из вештине у науку.

¹⁹ Могин Павле, Луковић Иван, *Принципи база података*, ФТН и МП “Stylos”, Нови Сад, 1996. Стр. 62-77.

²⁰ Могин Павле, наведено дело, стр. 8-9, 55-77.

²¹ За почетак теоријске формулатије релационог модела података се спис Едгара Ф. Кода: *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, који је објављен 1970. године. Током скоро две следеће деценије настављен је интензиван теоријски рад на прецизном формулисању релационог модела.

²² Који се најчешће, погрешно, назива упитним језиком, јер је постављање упита само једна од могућности овог језика.

3.1. Концепција релационог модела података

3.1.1. Независност: логичка и физичка

3.1.2. Једноставност: табела као једина структура података

3.1.3. Декларативност: упитни језик

3.2. Структурална компонента релационог модела података

Концепти²³: Домен, атрибут, релација (релациона схема, релациона променљива и релациона вредност), кључ (кандидат за кључ, надкључ, примарни, алтернативни, спољни), база (схема релационе базе и инстанца релационе базе), NULL ознака (вредност)

3.2.1. Релација

Скуп обједињава мноштво предмета у једну целину. Предмети морају да имају бар једну заједничку особину, које је основ њиховог обједињавања. Унутар скупа предмети су међусобно различити, што значи да је сваки члан скupa јединствен. Чланови скupa се називају елементима скupa; могу се навести набрањањем (екстензија скupa) или навођењем особине коју морају да задовоље (интензија скupa).

Пример. Скуп свих самогласника у српском језику може се описати на два начина:

1) екстензија скупа: $S = \{E, I, A, O, U\}$

2) интензија скупа: $S = \{x \mid x \in \mathcal{A} \wedge \mathcal{O}(x)\}$, где је: \mathcal{A} српски алфабет, а $\mathcal{O}(x)$ је особина: „ x је самогласник“

Пример. Скуп свих непарних природних бројева који су мањи од 10.

1) екстензија скупа: $A = \{1, 3, 5, 7, 9\}$

2) интензија скупа: $A = \{x \mid x \in \mathcal{N} \wedge x < 10\}$, где је \mathcal{N} скуп природних бројева

Декартов производ скупова A и B је скуп свих могућих уређених парова $\langle a, b \rangle$ тако да је $a \in A$ и $b \in B$:
$$A \times B = \{ \langle a, b \rangle \mid a \in A, b \in B \}.$$

Уопштено, нека су дати скупови D_1, D_2, \dots, D_n . Декартов производ ових скупова је скуп свих могућих уређених n -торки $\langle d_1, d_2, \dots, d_n \rangle$ тако да је $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$:

$$D_1 \times D_2 \times \dots \times D_n = \{ \langle d_1, d_2, \dots, d_n \rangle \mid d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n \}.$$

Пример. Декартов производ скупова $A = \{1, 3, 5, 7, 9\}$ и $B = \{2, 6, 10\}$ је скуп свих могућих уређених парова, таквих да је први елемент у пару из скупа A , а други елемент у пару је из скупа B . У овом случају, укупан број уређених парова је $5 \times 3 = 15$:

$$A \times B = \{ \langle 1, 2 \rangle, \langle 1, 6 \rangle, \langle 1, 10 \rangle, \langle 3, 2 \rangle, \langle 3, 6 \rangle, \langle 3, 10 \rangle, \langle 5, 2 \rangle, \langle 5, 6 \rangle, \langle 5, 10 \rangle, \langle 7, 2 \rangle, \langle 7, 6 \rangle, \langle 7, 10 \rangle, \langle 9, 2 \rangle, \langle 9, 6 \rangle, \langle 9, 10 \rangle \}.$$

Релација R дефинисана на скуповима A и B јесте подскуп Декартовог производа тих скупова, који садржи све оне уређене парове $\langle a, b \rangle$ који задовољавају услов релације Q :

$$A \times B \supseteq R = \{ \langle a, b \rangle \mid a \in A \wedge b \in B \wedge a Q b \}.$$

Пример. Дата су два скупа: $A = \{1, 3, 5, 7, 9\}$ и $B = \{2, 6, 10\}$. Дефинишими неколико релација над наведеним скуповима:

$$R_1: a = b/2, \text{ где: } a \in A, b \in B$$

$$R_2: a < b, \text{ где: } a \in A, b \in B$$

$$R_3: a = 3 * b, \text{ где: } a \in A, b \in B$$

Наведене релације важе за неке парове Декартовог производа скупова A и B , а за неке не важе. Релација је скуп свих они парови за које важи услов наведен у релацији. Претходно наведеним интензијама релације одговарају следеће екстензије релација:

²³ Ово поглавље у целости је засновано на: [1] стр. 53-61 и [5] стр. 52-73, 79-105. Целокупна структура овог поглавља преузета је из [1], као и најважније дефиниције.

$$R_1 = \{ <1, 2>, <3, 6>, <5, 10> \}$$

$$R_2 = \{ <1, 2>, <1, 6>, <1, 10>, <3, 6>, <3, 10>, <5, 6>, <5, 10>, <7, 10>, <9, 10> \}$$

$$R_3 = \{ \emptyset \}$$

Дакле, релација је скуп парова (ако се ради о бинарној релацији) за које је однос који одређује релација задовољен. Важи $R \subseteq A \times B$ и то онај подскуп који задовољава услов задате релације.

Уопшено, **релација R** дефинисана на скуповима D_1, D_2, \dots, D_n јесте подскуп Декартовог производа тих скупова, који садржи све оне н-торке $\langle d_1, d_2, \dots, d_n \rangle$ које задовољавају услов релације Q:

$$D_1 \times D_2 \times \dots \times D_n \supseteq R = \{ \langle d_1, d_2, \dots, d_n \rangle \mid d_1 \in D_1 \wedge d_2 \in D_2 \wedge \dots \wedge d_n \in D_n \wedge Q(d_1, d_2, \dots, d_n) \}.$$

Мање формално, релација R дефинисана на скуповима D_1, D_2, \dots, D_n јесте подскуп Декартовог производ наведених скупова: $R \subseteq D_1 \times D_2 \times \dots \times D_n$. Подскуп R садржи оне н-торке Декартовог производа које задовољавају задату релацију.

Домен релације. Ако је релација $R \subseteq D_1 \times D_2 \times \dots \times D_n$, онда се скупови D_1, D_2, \dots, D_n називају доменима релације R.

Степен релације. Ако је релација $R \subseteq D_1 \times D_2 \times \dots \times D_n$, онда је степен релације R једнак n. Дакле, број домена на којима је дефинисана нека релација јесте степен релације. Разликујемо унарне (на једном домену), бинарне (на два домена) и н-арне релације.

Кардиналност релације је број н-торки у релацији.

Атрибут(и) релације. Дати су скупови A, B, C и релација R:

$$A = \{1622, 1611, 1522\}$$

$$B = \{\text{Јова, Ана}\}$$

$$C = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

$$A \times B \times C \supseteq R = \{ \langle 1611, \text{Јова}, 3 \rangle, \langle 1622, \text{Ана}, 5 \rangle, \langle 1522, \text{Ана}, 7 \rangle \}$$

Како је релација скуп уређених н-торки, редослед елемената у једној н-торки је битан. Међутим, ако вредностима елемената у н-торкама придржимо имена домена, редослед елемената у н-торкама више неће имати значаја. Доделићемо скуповима и релацији семантичка имена:

$$A = \text{БИ} \quad \text{број индекса}$$

$$B = \text{Име} \quad \text{име студента}$$

$$C = \text{Сем} \quad \text{уписани семестар}$$

$$R = \text{СТУДЕНТ} \quad \text{студент факултета}$$

Сада је могуће претходно уведену релацију записати на следећи начин:

$$\text{СТУДЕНТ} = \{ \langle \text{БИ:1611, Име:Јова, Сем:3} \rangle, \langle \text{БИ:1622, Сем:5, Име:Ана} \rangle, \langle \text{Сем:7, Име:Ана, БИ:1522} \rangle \}$$

Атрибут релације се формално може дефинисати као пар **назив домена, вредност домена**. На пример, **Име, Јова** или **БИ, 1611**. Концепт атрибута омогућава представљање релације као табеле. Релација СТУДЕНТ се може представити у табеларној форми:

| СТУДЕНТ | | | ≤ Назив релације | Схема релације (интензија релације) |
|---------|------|-----|---------------------|--|
| БИ | Име | Сем | ≤ Атрибути релације | |
| 1611 | Јова | 3 | ≤ н-торка #1 | Вредност релације или Р-вредност |
| 1622 | Ана | 5 | ≤ н-торка #2 | (екstenзија релације) |
| 1522 | Ана | 7 | ≤ н-торка #3 | |

Екстензија релације је скуп свих н-торки разматране релације, односно табеларни приказ свих њених врста. Интензија релације је генерализација екстензије, она је временски непроменљива и састоји се од назива релације и атрибута релације.

Пошто је релација скуп, а свака табела није скуп, неопходно је одредити услове које табела мора да задовољи да би била релација:

- 1) Све врсте у табели морају да буду различите.
- 2) Редослед врста у табели је произвољан.
- 3) Редослед колона у табели је произвољан.
- 4) Све вредности поља (тј. вредности атрибута) у табели су атомске, односно није дозвољено да вредности неких поља у табели буду табеле²⁴.

Ако табела задовољава наведене услове, онда је она релација у првој нормалној форми (1НФ).

3.2.2. Домени атрибута

Сада се домени могу дефинисати као скупови из којих атрибути релације узимају своје вредности. Уобичајено је да се домени поделе на:

- Предефинисане домене – домене који су већ имплементирани и постоје у релационим језицима база података. На пример, у Transact-SQL-у постоје: цели бројеви (int, smallint, bigint, tinyint, bit), низови знакова²⁵ (char, varchar, text; nchar, nvarchar, ntext), децимални бројеви (decimal или numeric; money, smallmoney; float, real), датуми (datetime, date, time, datetime2, datetimeoffset, smalldatetime), низови бајтова²⁶ (binary, varbinary, image), просторни типови (spatial types) и посебни типови (cursor, timestamp, hierarchyid, uniqueidentifier, sql_variant, xml, table). Предефинисани домени у релационим језицима су еквивалентни предефинисаним (уграђеним) типовима података у програмским језицима.
- Семантичке домене – домене које програмер (database developer) креира употребом предефинисаних или претходно креираних семантичких домена. Дефиниција семантичког домена састоји се из навођења основног (“носећег”, underlaying) типа података и ограничења; на тај начин се новокреираном домену додељује одређено значење²⁷, због чега се и називају семантички. Семантички домени у релационим језицима су еквивалентни апстрактним типовима података²⁸ у програмским језицима.

Атрибуте релације би увек требало дефинисати над семантичким доменима, јер само тако програмер/СУБП може имати пуну контролу у извршавању операција над базом података. Два атрибута у моделу су семантички еквивалентна само ако су дефинисани над истим доменом.

3.2.3. Кључеви релације

Кључ релације је непразан скуп атрибута чије вредности јединствено одређују једну н-торку. Кључ може бити прост, уколико се састоји из једног атрибута, или сложен, уколико се састоји из два или више атрибута. На пример, у релацији СТУДЕНТ атрибут БИ је прост кључ (не постоје два студента која имају исти број индекса, тј. не постоје две н-торке релације СТУДЕНТ са истом вредношћу атрибута БИ). Формално, кључ релације R је таква колекција K њених атрибута која задовољава следећа два услова:

- 1) **особина јединствености:** не постоје било које две н-торке са истом вредношћу атрибута из K.
- 2) **особина нередундантности:** ако се изостави било који атрибут из K, губи се особина јединствености.

Она колекција атрибута која задовољава само особину јединствености назива се **надкључ** релације. На пример, колекција атрибута (БИ, Име, Сем, Смер#) јесте надкључ релације СТУДЕНТ, јер задовољава само услов јединствености. У једној релацији може постојати више различитих колекција атрибута које задовољавају дефиницију кључа. На пример, ако у релацију СТУДЕНТ додамо атрибут МБ (матични број), онда ће и он задовољавати дефиницију кључа. Све такве колекције се називају **кандидати за кључ**. Један од кандидата који се изабере да служи за идентификацију н-торки релације назива се **примарни кључ**. Остали неизабрани кандидати се тада називају **алтернативним кључевима**. Примарни кључ се у релационој схеми подвлачи. На пример, релациона схема за релације СТУДЕНТ и СМЕР:

СТУДЕНТ (БИ, Име, Сем, Смер#)
СМЕР (С#, Назив)

²⁴ Нису дозвољене табеле у табели.

²⁵ Низ знакова = ниска, стринг (string)

²⁶ Бинарне ниске

²⁷ “Уграђује” се одређено знање о разматраном систему у домен.

²⁸ Које, такође, креира програмер (application programmer).

Атрибути релације који су кључеви или су делови сложених кључева (било примарних, било алтернативних) називају се **кључни** атрибути. Остали атрибути релације су **некључни** атрибути. На пример, у претходним релационим схемама некључни атрибути су: Име, Сем и Назив. А какав је атрибут Смер#?

Свака релација може садржати атрибут(е) који је(су) у некој другој релацији примарни кључ. Такав атрибут(и) се назива **спољни кључ**²⁹. Вредност спољног кључа користи се за повезивање са вредношћу примарног кључа неке друге релације. У релационом моделу, везе између релација успостављају се преко вредности спољних кључева³⁰. У горе наведеном примеру, атрибут Смер# је спољни кључ; преко њега се релација СТУДЕНТ повезује са релацијом СМЕР. Спољни кључ СТУДЕНТ.Смер# референцира примарни кључ СМЕР.С#. Формално, спољни кључ се може одредити на следећи начин: Нека је релација R1 базна релација (*референцирајућа релација*). Спољни кључ SK у релацији R1 јесте подскуп њених атрибута такав да задовољава следећа два услова:

- 1) *егзистенција референциране релације*: постоји базна релација R2 која садржи кандидат за кључ KK и
- 2) *егзистенција референцирајуће вредности*: свака вредност SK у релацији R1 једнака је некој вредности KK у релацији R2 или је NULL³¹ вредност.

На пример, дате су следеће релације:

| СТУДЕНТ | | | | СМЕР | |
|----------------|------------|------------|--------------|-------------|--------------------------|
| БИ | Име | Сем | Смер# | C# | Назив |
| 1611 | Јова | 3 | 10 | 10 | Софтверско инжењерство |
| 1622 | Ана | 5 | NULL | 20 | Информациони системи |
| 1522 | Ана | 7 | 20 | 30 | Информационе технологије |
| 1533 | Ема | 7 | 10 | | |

Егзистенција референцирајуће релације:

Референцирајућа релација (R1): СТУДЕНТ

Спољни кључ (R1.SK): СТУДЕНТ.Смер#

Егзистенција референциране релације:

Референцирана релација (R2): СМЕР

Кандидат за кључ (R2.KK): СМЕР.С#

Егзистенција референцирајуће вредности:

{ СТУДЕНТ.Смер# } \subseteq { СМЕР.С# } ³²

3.2.4. Схема релационе базе и релациона база

Схема релационе базе (relational database scheme, RDS) дефинише логичку структуру базе података као скупа интензија релација. На пример, за концептуални модел наведен у прилогу, схема релационе базе података је:

| | |
|---------|---|
| ОДСЕК | (<u>О#</u> , Назив) |
| СМЕР | (<u>С#</u> , Назив, Одсек#) |
| СТУДЕНТ | (<u>БИ</u> , Име, Презиме, СредњеИме, ДатумРођења, Слика, ГодинаСтудија, ВозачкаКатегорија, Кредит, Стипендија, Смер#) |
| ПРЕДМЕТ | (<u>П#</u> , Назив, Врста, ЕСПБ, Одсек#) |
| ИСПИТ | (<u>БИ</u> , <u>П#</u> , ДатумПолагања, Оцена) |

Релациона база података (relational database, RDB) је:

- физички: скуп интегрисаних датотека са минималном редундансом³³ које су међусобно повезане³⁴ и променљиве у времену;
- концептуално: скуп временски променљивих релација³⁵;

²⁹ Или: страни кључ.

³⁰ Зато се релациони модел назива вредносно-оријентисани модел података.

³¹ Видети поглавље 3.2.5.

³² Прецизније: { СТУДЕНТ.Смер# } \subseteq { СМЕР.С# } \cup { NULL }

³³ Постоји редунданса спољних и примарних кључева.

³⁴ Преко примарних и спољних кључева.

- апликативно: скуп табела у којима се трајно чувају ажурабилни подаци о ентитетима из разматраног система.

На пример, за концептуални модел наведен у прилогу, део релационе базе података је:

| ПРЕДМЕТ | | | | СМЕР | |
|---------|---------------------|-------|------|------|--------------------------|
| П# | Назив | Врста | ЕСПБ | C# | Назив |
| 1 | Програмирање I | ОБВ | 6 | 10 | Софтверско инжењерство |
| 2 | Математика I | ОБВ | 6 | 20 | Информациони системи |
| 3 | Базе података | ОБВ | 6 | 30 | Информационе технологије |
| 4 | Економија | ОБВ | 8 | 40 | Управљање производњом |
| 5 | Увод у ИС | ОБВ | 6 | 50 | Управљање пословањем |
| 6 | Математика II | ОБВ | 6 | 60 | Управљање кадровима |
| 7 | Основе организације | ОБВ | 6 | | |
| 8 | Социологија | ИЗБ | 5 | | |
| 9 | Психологија | ИЗБ | 5 | | |

Релације у некој бази података могу да се поделе на базне и изведене. **Базна релација** је релација која се не може извести из осталих релација у релационој бази података, а у бази података се чува њена интензија и екстензија. **Изведена релација** (поглед, приказ) је релација која се може извести из скупа базних или изведенских релација, преко операција које се дефинишу над релацијама; у бази података се чува само њена интензија и поступак извођења, али не и подаци.

Упоредни приказ израза (термина) релационе, табеларне и датотечне обраде података:

| РЕЛАЦИОНА | ТАБЕЛАРНА | ДАТОТЕЧНА |
|---------------|-----------------------|------------------------|
| Домен | Скуп вредности | Тип |
| Атрибут | Колона | Поље |
| Н-торка | Врста | Запис / Слог / Рекорд |
| Примарни кључ | Идентификатор (врсте) | Јединствени кључ |
| Релација | Табела | Датотека података |
| Индекс | Индекс | Индексна датотека |
| Степен | Број колона | Број поља у запису |
| Кардиналност | Број врста | Број записа у датотеци |

3.2.5. NULL вредност

У случају недостатка податка о вредности атрибута, у базу података се уместо недостајуће вредности уписује NULL вредност. На пример, у табели СТУДЕНТ за н-торку:

< БИ: 1622, Име: Ана, Сем: 5, Смер#: NULL >

вредност NULL у атрибути Смер# значи да је **тренутно непозната вредност** за шифру смера. Drugim речима, када се сазна вредност атрибута за разматрану н-торку (вредност атрибута Смер# за студента чији је БИ = 1622) та ће вредност заменити NULL вредност, тј. биће уписана у базу података.

Прецизније, NULL вредност и није права вредност, већ је треба схватити као ознаку специјалне намене, маркер који указује на тренутно непознату вредност, чувар места (placeholder) за будућу вредност атрибута. У бази података ознака NULL има специфичан бинарни код, различит од свих других бинарних кодова. Зато се ознака NULL може користити уместо било које тренутно непознате вредности, за атрибуте било ког типа.

Проблеми са ознаком NULL могу настати у случају када је неки атрибут **неприменљиво својство** за сва појављивања објекта представљених н-торкама разматране релације. На пример, нека је релацији СТУДЕНТ додат атрибут ВозКат који значи возачку категорију коју студент има:

³⁵ Појам релације се у овој дефиницији користи двојако: као схема релације (интензија; апстрактна структура података) и као вредност релације (екстензија; физички постојећи, конкретни подаци). Зато се израз “временски променљивих” користи да укаже на чињенице да се током времена: (1) схема релације може мењати и (2) вредности атрибута релације могу мењати.

СТУДЕНТ

| БИ | Име | ВозКат | Смер# |
|------|------|--------|-------|
| 1611 | Јова | NULL | 10 |
| 1622 | Ана | Б | NULL |
| 1522 | Ана | NULL | 20 |
| 1533 | Ема | А | 10 |

Међутим, атрибут ВозКат применљив је само на студенте који имају возачку категорију. Студенти који имају БИ = 1611 и БИ = 1522 немају возачку категорију, па је зато:

- 1) атрибут ВозКат за њих непримениљиво својство,
- 2) вредност атрибута ВозКат за оба студента NULL.

Овде настаје проблем, јер у овом случају није могуће разлучити да ли је ознака NULL *тренутно непозната вредност* или *непримениљиво својство*. Означавање две семантички различите ситуације са једном ознаком NULL доводи до забуне. Да би се забуна избегла ознака NULL се користи само у првом случају (*тренутно непозната вредност*).

Генерално решење за атрибуте који су *непримениљива својства* је да се релације пројектују тако да су сви атрибути применљива својства за све н-торке у релацији. У претходном примеру то се може постићи декомпозицијом табеле СТУДЕНТ на две нове релације:

СТУДЕНТ_1 (БИ, Име, Смер#) и
СТУДЕНТ_ВОЗАЧ (БИ, ВозКат).

3.3. Операциона компонента релационог модела података

Операције релационог модела могуће је исказати коришћењем релационе алгебре. Операције над релацијама могу се разврстати у:

- Конвенционалне скуповне операције: унија, пресек, разлика и Декартов производ;
- Специјалне релационе операције: селекција, пројекција, спајање и делење;
- Додатне релационе операције: полуспајање, хоризонтално скаларно рачунање у релационој алгебри (*extend*), вертикално скаларно рачунање у релационој алгебри (*summarize*), операције поређења релација (једнакост и неједнакост релација, подскуп и надскуп релације, прави подскуп и прави надскуп релације);
- Операције ажурирања релације: *insert* (додавање нове н-торке у релацију), *delete* (брисање н-торке из релације) и *update* (промена вредности атрибута);
- Операције са NULL ознакама.

Пошто су операције релационог модела интуитивно јасне, овде се неће приступити њиховом формалном и детаљном објашњавању³⁶.

3.4. Интегритетна компонента релационог модела података

3.4.1. Ограничења

Ограничења релационог модела података. Сама структура релационог модела диктира два *описа ограничења*:

- (i) Ограниччење на вредности примарног кључа,
- (ii) Ограначење на вредности спољног кључа.

Ограничења разматраног система (пословна правила). У релационом моделу *посебна ограничења* је могуће поделити на:

- (i) ограничења стања система (ограничења на могуће вредности података у бази), а која могу бити:
 - 01: ограничења за домене – којима се одређује које вредности постоје у домену (дозвољене вредности за домен); на пример, домен ОЦЕНА садржи вредности од 5 до 10;
 - 02: ограничења за атрибуте – којима се одређују дозвољене вредности неког атрибута независно од вредности других атрибута у бази (дозвољене вредности за атрибут); на пример, атрибут Оцена релације ПОЛОЖЕНИ_ИСПИТ може да садржи вредности од 6 до 10 (мада је сам атрибут Оцена дефинисан над доменом ОЦЕНА):

³⁶ Такви описи могу се наћи у: [1] стр. 62-83 и [5] стр. 139-218.

ПОЛОЖЕНИ_ИСПИТ. Оцена *is type* ОЦЕНА *and check value in [6..10];*

03: ограничења за релације – којима се одређује вредност једног атрибута релације на основу вредности другог атрибута (других атрибута) у једној релацији (међузависност вредности атрибута једне релације); на пример, ако је ставка рачуна представљена следећом релационом схемом:

СТАВКА(Рачун#, С#, Производ, Количина, ЈединичнаЦена, Вредност)

онда се последњи атрибут у претходној релационој схеми израчунава по обрасцу:

СТАВКА.Вредност = СТАВКА.Количина * СТАВКА.ЈединичнаЦена;

04: ограничења за базу – којима се одређује вредност једног атрибута релације на основу вредности атрибута из више релацији (међузависност вредности атрибута више релација); на пример, ако је дата релациониа схема:

РАЧУН(Р#, ..., Укупно)

СТАВКА(Рачун#, С#, Производ, Количина, ЈединичнаЦена, Вредност)

онда се вредност атрибута Укупно израчунава на следећи начин:

РАЧУН.Укупно = *SUM*(СТАВКА.Вредност *WHERE* РАЧУН.Р# = СТАВКА.Рачун#)

- (ii) ограничења промене стања, тј. прелаза из једног стања у друго стање система (ограничења на могуће промене вредности података у бази) – којима се одређује да ли је могућ прелаз из једног скупа вредности атрибута у други скуп вредности; на пример, нека су дате следеће релације:

| СТУДЕНТ | | | | СМЕР | |
|---------|------|-----|-------|------|------------------------|
| БИ | Име | Сем | Смер# | C# | Назив |
| 1611 | Јова | 3 | 10 | 10 | Софтверско инжењерство |
| 1622 | Ана | 5 | NULL | 20 | Информациони системи |
| 1522 | Ана | 7 | 30 | 60 | Управљање кадровима |
| 1533 | Ема | 7 | 10 | | |

- задато следеће правило: Студенти са смера ‘Управљање кадровима’ не могу да пређу на смер ‘Информациони системи’, а ни на смер ‘Софтверско инжењерство’. Обрнуто је дозвољено.

Онда би спецификација претходног правила могла да се обави на следећи начин³⁷:

```

CONSTRAINT [ПрелазСаСмераНаСмер]
    ON [СТУДЕНТ]
BEGIN
    LET &УК is type as SCALAR( [СМЕР.С#] );
    LET &СИИС is type as TABLE( [СМЕР.С#] );

    &УК := SELECT [С#] FROM [СМЕР]
        WHERE [НАЗИВ] LIKE 'Управљање кадровима';
    &СИИС := SELECT [С#] FROM [СМЕР]
        WHERE [НАЗИВ] LIKE 'Софтверско инжењерство'
        OR [НАЗИВ] LIKE 'Информациони системи';

    IF ( OLD[СТУДЕНТ.Смер#] = &УК ) AND ( NEW[СТУДЕНТ.Смер#] IN &СИИС )
        THEN Rejection;
END.

```

³⁷ У хипотетичком језику.

На основу претходно дефинисаних ограничења могу се дефинисати правила интегритета.

3.4.2. Правила интегритета

Присетимо се да је правило интегритета³⁸ уређена четвртка³⁹:

ПИ = <Операција, Ограниччење, Акција [, Тренутак]>.

Правила интегритета дефинишу дозвољена стања и дозвољене прелазе система из једног у друго стање. У релационом моделу правила интегритета се исказују дефинисањем ограничења на вредности атрибута и акцијама које се предузимају када нека операција ажурирања наруши раније утврђено ограничење.

Правила интегритета релационог модела података:

1. интегритет ентитета (интегритет примарног кључа),
2. референцијални интегритет.

Правила интегритета разматраног система (пословна правила):

1. правила интегритета стања система,
 - (i) интегритет домена,
 - (ii) интегритет атрибута,
 - (iii) интегритет релације,
 - (iv) интегритет базе;
2. правила интегритета промене стања, тј. прелаза из једног стања у друго стање система.

Интегритет ентитета (интегритет примарног кључа): ниједан атрибут који је примарни кључ или део примарног кључа неке базне релације не може да има NULL вредност.

Примарни кључеви у базним релацијама идентификују један објекат у скупу објеката (нпр. један предмет у релацији ПРЕДМЕТ или једног студента у релацији СТУДЕНТ). Због такве њихове улоге примарни кључеви не могу имати NULL вредности. Постојање NULL вредности у примарном кључу (целом или било ком његовом делу) нарушило би саму дефиницију кључа и то како особину јединствености, тако и особину нередундантности.

Ограниччење које дефинише интегритет ентитета могу да наруше операције убаџивања нове н-торке у релацију или операције промене вредности. У оба случаја једино могућа акција је одбијање (*reject* = одбацивати, одбијати; *rejection* = одбацивање, одбијање) операције.

Референцијални интегритет: Ако нека базна релација R1 поседује спољни кључ R1.SK који ову релацију повезује са неком другом базном релацијом R2 преко примарног кључа R2.PK, тада свака вредност R1.SK мора бити било једнака некој вредности R2.PK или бити NULL вредност. Релације R1 и R2 не морају бити различите. Коришћењем релационе алгебре претходно ограничење референцијалног интегритета може се записати на следећи начин:

$$\pi_{\text{SK}}(R1) \subseteq \pi_{\text{PK}}(R2) \cup \{ \text{NULL} \}^{40}$$

Референцијални интегритет обезбеђује коректно повезивање н-торки релација у релационом моделу⁴¹. Нарушавањем референцијалног интегритета укидају се везе између н-торки релација и читава релационија база података се распада. Да до тога не би дошло, дефинише се правило референцијалног интегритета. Нарушавање овог правила може се десити у два случаја:

1. У релацији која садржи спољни кључ⁴² вредности атрибута које чине спољни кључ нису одговарајуће. То се може десити приликом:
 - а. Убаџивања нове н-торке у релацију,
 - б. Измене вредности атрибута који чине спољни кључ.

³⁸ Ово поглавље у целости је засновано на: [1] стр. 87-94 и [5] стр. 110-134. Целокупна структура овог поглавља преузета је из [1], као и најважније дефиниције.

³⁹ Средње заграде указују на опциону компоненту.

⁴⁰ Пројекција релације R1 по спољном кључу SK увек је подскуп пројекције релације R2 по примарном кључу PK проширене елементом NULL.

⁴¹ Видети 3.2.3. за пример.

⁴² Референцирајућа релација.

2. У релацији која садржи примарни кључ⁴³ вредности атрибута које чине примарни кључ нису одговарајуће. То се може десити приликом:
 - а. Избацивања н-торке из релације,
 - б. Измене вредности атрибута који чине примарни кључ.

За случајеве 1.а. и 1.б. једино могућа акција је одбијање операције (*rejection*) која је изазвала нарушавање ограничења. У случајевима 2.а. и 2.б. уколико дође до нарушавања референцијалног интегритета могуће су следеће акције:

- *Rejection (Restrict)*: одбија се операција која је нарушила ограничење референцијалног интегритета;
- *Cascade*: преноси се извршавање одговарајуће операције⁴⁴ и на релацију у којој се налази спољни кључ, да би се на тај начин задовољило ограничење референцијалног интегритета. Другим речим, покреће се⁴⁵ извршење истоветне операције и на референцирајућој релацији (“продужено” избацивање или “продужена” измена).
- *SetNull*: замењује се вредност спољног кључа са NULL вредношћу за све н-торке којима одговара избачени или промењени спољни кључ.
- *SetDefault*: замењује се вредност спољног кључа са неком унапред одређеном (default) вредношћу за све н-торке којима одговара избачени или промењени спољни кључ.
- *NoAction*: обавља се захтевано ажурирање базе података без икаквих додатних акција⁴⁶.

Правила интегритета разматраног система (пословна правила) изражавају посебна ограничења и одговарајуће акције при њиховом нарушавању. Ове врсте акција се не могу схематизовати и унапред одредити као што је то урађено за референцијални интегритет. Зато свако нарушавање ограничења које је својствено разматраном систему захтева специфичну акцију, која се имплементира преко тригера⁴⁷.

Интегритет домена одређује скуп дозвољених вредности за домен и акцију која се предузима уколико разматрана вредност није у том скупу. Ограниччење за домен могу да наруше операције убаџивања нове н-торке у релацију или операције промене вредности атрибута постојеће н-торке. У оба случаја једино могућа акција је одбијање операције.

$\text{ПИ}_{\text{домен}} = <\text{Домен}, [\text{Операција}],>$ ⁴⁸ Ограниччење, [*Rejection*,] Пре >,
где је: Операција = { *Update*, *Insert* }.

Дефинисање правила интегритета за домен јесте друго име за креирање семантичког домена.

Интегритет атрибута одређује скуп дозвољених вредности за атрибут и акцију која се предузима уколико разматрана вредност није у том скупу. Ограниччење за атрибут могу да наруше операције убаџивања нове н-торке у релацију или операције промене вредности атрибута постојеће н-торке. У оба случаја једино могућа акција је одбијање (*rejection*) операције.

$\text{ПИ}_{\text{атрибут}} = <\text{Атрибут}, \text{Домен}, [\text{Операција}],>$ ⁴⁹ Ограниччење, [*Rejection*,] Пре >,
где је: Операција = { *Update*, *Insert* }.

Интегритет релације одређује:

- ограничење којим се исказује на који начин је вредност једног атрибута у релацији повезана (условљена) са вредностима других атрибута у истој релацији и
- акцију која се предузима уколико је та повезаност (условљеност) нарушена.

⁴³ Референцирана релација.

⁴⁴ Избацивање за избацивање и промена вредности за промену вредности.

⁴⁵ “окида се”

⁴⁶ Опрез! Ово је опасно и прихватљиво је само у ретким случајевима!

⁴⁷ Тригер (trigger) је процедура која се извршава кад год је задовољен услов за њено активирање. Он се не може извршити као што се извршавају остале процедуре, непосредним позивом, већ само када се деси неки догађај или нека операција ажурирања базе (*Update*, *Insert*, *Delete*). Претходно дефинисане акције *Rejection (Restrict)*, *Cascade*, *SetNull* и *SetDefault* могу се схватити као специјализовани тригери.

⁴⁸ Навођење операције и акције је у начелу непотребно, јер је акција увек иста: било да је операција *Update* или *Insert*, уколико је ограничење нарушено акција је увек *Rejection*.

⁴⁹ Види претходну напомену.

Ограниччење повезаности вредности атрибута у једној релацији могу да наруше операције: убацивања нове н-торке, брисање постојеће н-торке или промене вредности атрибута постојеће н-торке. У свим случајевима могућа је једна од следећих акција:

- одбијања (*rejection*) извршења операције која је нарушила интегритет⁵⁰ или
- израчунавање нове вредности зависног атрибута и ажурирање зависног атрибута новосрачунатом вредношћу⁵¹.

$\text{ПИ}_{\text{релације}} = <\text{Релација}, \text{Атрибут}, \text{Операција}, \text{Ограниччење}, \text{Акција}, \text{Тренутак}>$,
где је: Операција = { *Update, Insert, Delete* }.

У *Ограниччењу* и *Акцији* дозвољен је приступ и промена података само у *Релацији* специфицираној у $\text{ПИ}_{\text{релације}}$.

Интегритет базе одређује било које сложено ограничење на вредности атрибута у бази података, ограничење које повезује вредности атрибута из више релација. Прецизније, интегритет базе одређује:

- ограничење којим се исказује на који начин је вредност једног атрибута у релацији повезана (условљена) са вредностима других атрибута у осталим релацијама и
- акцију која се предузима уколико је та повезаност (условљеност) нарушена.

Ограниччење повезаности вредности атрибута у бази могу да наруше операције: убацивања нове н-торке, брисање постојеће н-торке или промене вредности атрибута постојеће н-торке. У свим случајевима могућа је једна од следећих акција:

- одбијања (*rejection*) извршења операције која је нарушила интегритет⁵² или
- израчунавање нове вредности зависног атрибута и ажурирање зависног атрибута новосрачунатом вредношћу⁵³.

$\text{ПИ}_{\text{базе}} = <\text{Релација}, \text{Атрибут}, \text{Операција}, \text{Ограниччење}, \text{Акција}, \text{Тренутак}>$,
где је: Операција = { *Update, Insert, Delete* }.

У *Ограниччењу* и *Акцији* дозвољен је приступ подацима у свим релацијама базе података, али је дозвољена промена само *Атрибута* који припада *Релацији* специфицираној у $\text{ПИ}_{\text{базе}}$.

Интегритет прелаза из стања у стање одређује услове прелаза из једног (старог, почетног) у друго (ново, одредишно) стање. За то су потребне две помоћне привремене релације: једна која садржи измене н-торке са старим вредностима (*OLD* или *DELETED*) и друга која садржи исте те измене н-торке, али са новим вредностима (*NEW* или *INSERTED*). Поређењем вредности атрибута у старим н-торкама са вредностима атрибута у новим н-торкама (тј. старог стања са новим стањем), утврђује се да ли је ограничење за дозвољени прелаз из једног стања у друго стање нарушено. Ограниччење могу да наруше операције ажурирања (*Insert, Update* и *Delete*). Ако је ограничење нарушено, приступа се акцији која базу преводи у неко ново или враћа у старо конзистентно стање.

$\text{ПИ}_{\text{прелаза}} = <\text{Релација}, \text{Атрибут}, \text{Операција}, \text{Ограниччење}, \text{Акција}, \text{Тренутак}>$

У *Ограниччењу* и *Акцији* дозвољен је приступ и промена података у свим релацијама базе података, а не само у *Релацији* специфицираној у $\text{ПИ}_{\text{прелаза}}$.

3.5. Кодова релационна правила

Кодова правила⁵⁴ чини скуп од тринест правила која је предложио Едгар Ф. Код, са циљем да дефинише шта је потребно да поседује један систем за управљање базом података да би га могли сматрати релационим (Relational DBMS, RDBMS). Ова правила су срочена са намером да се спречи комерцијално “разводњавање” дефиниције релационог модела података и укаже на неодобравајуће имплементације система за управљање базама података⁵⁵. Детаљно бављење овом проблематиком довело је Е. Ф. Кода до књиге *The Relational Model for Database Management: Version 2*, у којој је

⁵⁰ Лошији приступ.

⁵¹ Болији приступ.

⁵² Лошији приступ.

⁵³ Болији приступ.

⁵⁴ Видети у литератури [6] и [7]. Као и линк [4].

⁵⁵ Узети у обзир напомену Дејва Вурхиса (Dave Voorhis): “Note that [...] these rules were appropriate to their time (mid 1980s), their intent (quash marketing hype at a time when any remotely DBMS-like product would probably claim to be “relational”), and their audience (readers of ComputerWorld, a popular semi-technical industry magazine) [...].”

изложио 333 релационих правила. Треба рећи да већина комерцијалних система никада није у потпуности задовољила ни ових овде наведених 1+12 правила.

0. Основно правило:

Да би систем назвали релационим системом за управљање базама података (РСУБП), он у свом функционисању мора користити искључиво сопствене релационе могућности за управљање подацима.

Следи дванаест правила⁵⁶ који су разложено и потанко објашњено *Основно правило*:

1. Правило представљања података:

Све подаци у релационој бази података су на логичком нивоу представљени само на један начин – вредностима у релацијама (табелама).⁵⁷

2. Правило обезбеђеног начина приступа подацима:

Сваки појединачни податак (атомска вредност) у релационој бази доступан је навођењем назива релације (табеле), вредности примарног кључа и назива колоне.⁵⁸

3. Систематично третирање NULL вредности:

У потпуно релационом СУБП-у за представљање податка који недостаје или који се не може применити (недостајућа вредности или непримениљиво својство) користи се NULL вредност⁵⁹. Систем подржава све операције са NULL вредностима, без обзира на тип података.

4. Динамички, увек доступан каталог (речник) података заснован на релационом моделу:

Опис базе података (мета-подаци) на логичком нивоу представљен је на исти начин као и обични подаци⁶⁰, тако да овлашћени корисник може да користи исти релациони упитни језик за приступ и мета-подацима и обичним подацима.

5. Правило свеобухватног подјезика за податке:

Релациони систем може да подржава више језика и више различитих начина крајње употребе података. Међутим, мора да постоји барем један језик чије се наредбе могу изразити као низ карактера са добро дефинисаном синтаксом и који подржава:

- Дефиницију података
- Дефиницију погледа
- Манипулацију подацима (интерактивно и кроз апликативне програме)
- Дефиницију правила интегритета
- Ауторизацију (сигурност података, права коришћења података)
- Дефинисања граница трансакција (begin, commit, rollback).

6. Правило ажурирања погледа:

Сви погледи који су теоријски ажурабилни, ажурабилни су и од стране система.⁶¹

7. Уношење, мењање и брисање (уклањање) података на нивоу скупова:

И над базним релацијама и над изведеним релацијама (погледима) могу се извршавати не само операције извештавања (приказа података), већ и операције ажурирања података (insertion, update and deletion of data).

8. Физичка независност података:

Апликативни програми и интерактивна комуникација остају логички непромењени кад год се промени физичка организација базе или метод приступа.⁶²

9. Логичка независност података:

⁵⁶ За која Е. Ф. Код каже: "All 12 rules are motivated by Rule Zero defined above, but a DBMS can be more readily checked for compliance with these 12 than with Rule Zero".

⁵⁷ Структура базе података се на концептуалном (овде: логичком) нивоу представља искључиво релацијама (табелама).

⁵⁸ Без унапред задатих приступних путева и без рекурзије или итерације.

⁵⁹ Специфична ознака, различита од свих других вредности у бази; различита од размака, празног низа знакова, нуле или било ког броја.

⁶⁰ То јест, у облику релација (табела).

⁶¹ Сви погледи које је теоријски могуће ажурирати, могу се ажурирати и кроз систем.

⁶² Промене на физичком нивоу (начин на који се чувају подаци, начин на који се приступа посацима) не смеју изазивати промене у апликацијама или интерактивним комуникацијама.

Апликативни програми и интерактивна комуникација остају логички непромењени кад год се обаве теоријски могуће промене над базним релацијама (табелама), којима се не губе подаци.⁶³

10. Независност правила интегритета:

Правила интегритета базе података морају бити дефинисана независно од апликативних програма и сачувана у каталогу. Мора бити предвиђена могућност њихове измене у било ком тренутку без непотребног утицаја на постојеће апликације.

11. Независност од дистрибуције:

Све претходно наведене карактеристике су независне од дистрибуирањости базе података. Дистрибуција делова базе на различите локације мора бити невидљива за кориснике базе. Постојеће апликације морају наставити са радом:

- (а) када се дистрибуција података уводи први пут и
- (б) при било којој следећој редистрибуцији података у систему.

12. Забрана субверзије:

Ако релациони систем поседује или може да ради са неким језиком нижег нивоа (у ком се обрађује једна слог у једном тренутку), кроз тај језик се не могу подривати или заобићи правила интегритета задата преко релационог језика⁶⁴.

⁶³ Промене на логичком нивоу (табела и колона) не смеју захтевати промене апликација (и интерактивних комуникација) које их користе.

⁶⁴ Који је језик вишег нивоа, јер обрађује више рекорда у једном тренутку.

4. Релациони упитни језик: Structured Query Language (SQL)

Релациони упитни језик *Structured Query Language* (SQL) је доменски-специфичан⁶⁵ језик осмишљен за:

- манипулацију подацима (креирање, ажурирање, заштита и одржавање података),
- очување интегритета података (специфицирање⁶⁶ или програмирање⁶⁷ правила интегритета),
- издавање података (специфицирање или програмирање приступа подацима ради њиховог приказа)

у релационим системима за управљање базама података.

SQL је скуповно базиран, декларативни програмски језик⁶⁸. Међутим, током времена су SQL-у додавана разна процедурална проширења, као што су наредбе за управљање током извршења (control-of-flow constructs). Неке од најпознатијих комерцијално расположивих проширенih верзија стандардног SQL-а су:

| Назив стандарда | Скраћени назив | Пун назив |
|----------------------|----------------|---|
| ANSI/ISO Standard | SQL/PSM | SQL/Persistent Stored Modules |
| Назив СУБП | | |
| Interbase / Firebird | PSQL | Procedural SQL |
| IBM DB2 | SQL PL | SQL Procedural Language (implements SQL/PSM) |
| IBM Informix | SPL | Stored Procedural Language |
| Microsoft / Sybase | T-SQL | Transact-SQL |
| MySQL | SQL/PSM | SQL/Persistent Stored Module (implements SQL/PSM) |
| Oracle | PL/SQL | Procedural Language/SQL (based on Ada) |
| PostgreSQL | PL/pgSQL | Procedural Language/PostgreSQL Structured Query Language (implements SQL/PSM) |
| Sybase | Watcom-SQL | SQL Anywhere Watcom-SQL Dialect |
| Teradata | SPL | Stored Procedural Language |
| SAP | SAP HANA | SQL Script |

Поред стандардом прописане екstenзије (SQL/PSM extensions) и бројних појединачних додатака, водећи произвођачи комерцијалних СУБП развили су могућности структурног и објектно-оријентисаног програмирања преко интеграције СУБП са другим језицима. Тако, SQL стандард дефинише SQL/JRT екstenзију (SQL Routines and Types for the Java Programming Language) за подршку Јава (Java) кôду унутар SQL базе. Microsoft од верзије *SQL Server 2005* користи *SQLCLR* (SQL Server Common Language Runtime) за хостовање управљаних .NET склопова (host managed .NET assemblies) у SQL бази, док су претходне верзије SQL Server-а биле ограничene на спољне проширене усклаиштене процедуре примарно писане у C/C++. PostgreSQL омогућава програмерима да пишу функције у разним језицима (Perl, Python, Tcl и C).

4.1. Врсте SQL наредби

У стандарду ISO/IEC 9075-1:2011 у одељку 4.11.2, SQL наредбе су разврстане по функционалности у следеће врсте⁶⁹:

⁶⁵ Језик намењен само једном апликационом домену, тј. језик посебне намене. Идеја DCJ је блиска идеји софтвера као модела разматраног система. Основна намера доменског инжењерства јесте повећати ниво апстракције, семантичко богаство и изражайност DC језика у односу на опшненаменске језике (C, C++, Јава, C#). Цена за то је мања свеобухватност језика, сужена област примене на само један домен, време утрошено за израду DCJ и (можда) слабије перформансе. Са pragматског гледишта, циљ је брза изградња оперативних модела који могу бити лако валидирани од стране стручњака (доменских експерата) и лако изменљиви (било због потребе да се коригује модел због лошег моделовања, било због потребе да се врши адаптација модела због промене разматраног система).

⁶⁶ Декларативан приступ правилима интегритета.

⁶⁷ Процедурални приступ правилима интегритета.

⁶⁸ За разлику од Паскала (Pascal) или Ца (C) који су императивни програмски језици.

⁶⁹ Наведено у изворном облику. Навод је из радне верзије стандарда (draft version) која је аутору била доступна са линка наведеног у попису референци.

1. **SQL-schema** statements that can be used to create, alter, and drop schemas and schema objects.
2. **SQL-data-manipulation** statements that perform insert, update, and delete operations on tables. Execution of an SQL-data statement is capable of affecting more than one row, of more than one table.
3. **SQL-data-retrieve** statements that perform queries on tables. Execution of an SQL-data statement is capable of affecting more than one row, of more than one table.
4. **SQL-transaction** statements that set parameters for, and start or terminate transactions.
5. **SQL-control** statements that may be used to control the execution of a sequence of SQL statements.
6. **SQL-connection** statements that initiate and terminate connections, and allow an SQL-client to switch from an SQL-session with one SQL-server to an SQL-session with another.
7. **SQL-session** statements that set some default values and other parameters of an SQL-session.
8. **SQL-diagnostics** statements that get diagnostics (from the diagnostics area) and signal exceptions in SQL routines.
9. **SQL-dynamic** statements that support the preparation and execution of dynamically-generated SQL-statements, and obtaining information about them.

У средишту наше пажње биће прве три групе наредби.

4.2. SQL-schema statements

Наредбе за прављење објекта базе података.

Некада: Data Definition Language (DDL) statements

CREATE (креирај), DROP (уништи), ALTER (замени)

4.2.1. SQL типови података

Свака колона у SQL табели дефинисана је над неким типом података (data type). ISO/ANSI SQL прописује следеће основне типове.

Ниске знакова (character strings):

- CHARACTER(*n*) или CHAR(*n*): стринг фиксне дужине од *n* знакова
- CHARACTER VARYING(*n*) или VARCHAR(*n*): стринг променљиве дужине, максимално *n* знакова
- NATIONAL CHARACTER(*n*) или NCHAR(*n*): стринг фиксне дужине од *n* знакова који подржава ICS (international character set)
- NATIONAL CHARACTER VARYING(*n*) или NVARCHAR(*n*): стринг променљиве дужине, максимално *n* знакова који подржава ICS (international character set)

Ниске битова (bit strings)

- BIT(*n*): низ од *n* битова
- BIT VARYING(*n*): низ који има до *n* битова
- BYTE VARYING(*n*): низ од *n* бајтова

Бројеви (numbers)

- INTEGER, SMALLINT и BIGINT
- FLOAT, REAL и DOUBLE PRECISION
- NUMERIC(*precision, scale*) или DECIMAL(*precision, scale*)

Временски (temporal или date/time)

- DATE: за датум (нпр. 2011.11.23)
- TIME: за време (нпр. 15:51:36)
- TIME WITH TIME ZONE или TIMETZ: исто као TIME, али укључује и податке о временској зони
- TIMESTAMP: DATE и TIME заједно
- TIMESTAMP WITH TIME ZONE или TIMESTAMPTZ: DATE и TIME заједно, али укључује и податке о временској зони.

4.2.2. Домени

Домен је у SQL-у прост, кориснички дефинисан именован објекат који се може користити као алтернатива за предефинисан тип податка над којим се дефинише колона. Може имати default вредност и једно или више ограничења. Домен се креира наредбом:

```
CREATE DOMAIN <naziv domena> [AS] <predefinisani tip>
[DEFAULT <vrednost>]
[[CONSTRAINT <naziv ograničenja>] CHECK (<ograničenje>)];
```

Дефиниција домена се мења наредбом ALTER:

```
ALTER DOMAIN <naziv domena>
SET DEFAULT <vrednost> |
DROP DEFAULT |
ADD [CONSTRAINT <naziv ograničenja>] CHECK (<ograničenje>) |
DROP CONSTRAINT <naziv ograničenja>;
```

Домен се уништава наредбом:

```
DROP DOMAIN <naziv domena>;
```

4.2.3. Табеле и колоне

Креирање табеле:

```
CREATE TABLE <naziv tabele>
(<naziv kolone1> <tip podatka> NOT NULL,
<naziv kolone> <tip podatka> [NOT NULL], ...)
```

Измена дефиниције табеле:

Добавање нове колоне
ALTER TABLE <naziv tabele>
[ADD COLUMN] <definicija kolone>;

Izmena postojeće kolone:

```
ALTER TABLE <naziv tabele>
[ALTER COLUMN] <naziv kolone>
SET DEFAULT <vrednost> |
DROP DEFAULT;
```

Izbacivanje колоне из табеле:

```
ALTER TABLE <naziv tabele>
DROP [COLUMN] <naziv kolone>;
```

Добављање или избацивање ограничења на вредности:

```
ALTER TABLE <naziv tabele>
ADD [CONSTRAINT <naziv ograničenja> ] <ограничење табеле> |
DROP CONSTRAINT <naziv ograničenja>;
```

Уништавање табеле:

```
DROP TABLE <naziv tabele>;
```

Креирање виртуалне табеле (погледа, приказа):

```
CREATE VIEW <naziv pogleda> [(<naziv kolone1>, {<naziv kolone1>})]
AS
<SQL upit>
[WITH [LOCAL | CASCDED] CHECK OPTION];
```

Измена дефиниције виртуалне табеле:

```
ALTER VIEW <naziv pogleda> [(<naziv kolone1>, {<naziv kolone1>})]
AS
<SQL upit>
[WITH [LOCAL | CASCDED] CHECK OPTION];
```

Уништавање виртуалне табеле:

```
DROP VIEW <naziv pogleda>;
```

4.2.4. Индекси

Индекси су структуре података које олакшавају и чине ефикаснијим приступ подацима базе. Вредности индексираних колона могу бити јединствене уколико се при креирању изабере опција UNIQUE.

```
CREATE [UNIQUE] INDEX <naziv indeksa>
ON <naziv tabele> ( <naziv kolone1> [, <naziv kolone2>, ...] ) ;
```

Уништавање се врши наредбом:

```
DROP INDEX <naziv indeksa>;
```

4.2.5. Схеме

Схема (schema) је именова група логички повезаних објеката у бази; она обједињује све објекте који деле исти именски простор (простор именовања). Схема може садржати једну или више табела, а свака табела може припадати логички тачно једној схеми. Схема се креира наредбом:

```
CREATE SCHEMA <naziv sheme>;
```

Уништавање схеме може бити са опцијом CASCADE (при чему се уништава схема и сви објекти из ње) или опцијом RESTRICT (уништавање шеме која је празна) и остварује се наредбом:

```
DROP SCHEMA <naziv sheme> CASCADE | RESTRICT;
```

Пун назив објекта базе података састоји се из више делова; кад се жели навести потпуно квалификовано име објекта, наводи се троделно (у T-SQL је четвроредично) име:

```
<naziv_baze>.<naziv_sheme>.<naziv_objekta>
```

4.3. SQL-data-manipulation statements

Наредбе за ажурирање података. Некада: Data Manipulation Language (DML) statements

INSERT (убаци), UPDATE (промени), DELETE (избаци)

Додавање нових редова у табелу:

```
[ WITH <common_table_expression> [ ,...n ] ]
INSERT
{
    [ TOP ( expression ) [ PERCENT ] ]
    [ INTO ]
    { <object> | rowset_function_limited
        [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
    }
    {
        [ ( column_list ) ]
        [ <OUTPUT Clause> ]
        { VALUES ( { DEFAULT | NULL | expression } [ ,...n ] ) [ ,...n      ]
        | derived_table
        | execute_statement
        | <dml_table_source>
        | DEFAULT VALUES
        }
    }
} [ ; ]
```

Краће:

```
INSERT INTO <naziv_tabele> VALUES (<vrednost1>, <vrednost2> ...);
```

Промена вредности у табели:

```
[ WITH <common_table_expression> [...n] ]
UPDATE
[ TOP ( expression ) [ PERCENT ] ]
{ { table_alias | <object> | rowset_function_limited
    [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
}
| @table_variable
```

```

    }
    SET
      { column_name = { expression | DEFAULT | NULL }
        | { udt_column_name.{ { property_name = expression | field_name = expression }
          | method_name ( argument [ ,...n ] ) }
        }
        | column_name { .WRITE ( expression , @Offset , @Length ) }
        | @variable = expression
        | @variable = column = expression
        | column_name { += | -= | *= | /= | %= | &= | ^= | |= } expression
        | @variable { += | -= | *= | /= | %= | &= | ^= | |= } expression
        | @variable = column { += | -= | *= | /= | %= | &= | ^= | |= } expression
      } [ ,...n ]

      [ <OUTPUT Clause> ]
      [ FROM{ <table_source> } [ ,...n ] ]
      [ WHERE { <search_condition>
        | { [ CURRENT OF
            { { [ GLOBAL ] cursor_name }
            | cursor_variable_name
          }
        ]
      }
    ]
    [ OPTION ( <query_hint> [ ,...n ] ) ]
  [ ; ]
}

<object> ::= {
  [ server_name . database_name . schema_name .
  | database_name .[ schema_name ] .
  | schema_name .
  ]
  table_or_view_name}

```

Краће:

```

UPDATE TABLE
SET <naziv kolone1> = <izraz1> [, <naziv kolone2> = <izraz2>, ...]
[WHERE <uslov>];

```

Брисање редова у табели:

```

[ WITH <common_table_expression> [ ,...n ] ]
DELETE
  [ TOP ( expression ) [ PERCENT ] ]
  [ FROM ]
  { { table_alias
    | <object>
    | rowset_function_limited
    [ WITH ( table_hint_limited [ ...n ] ) ] }
    | @table_variable
  }
  [ <OUTPUT Clause> ]
  [ FROM table_source [ ,...n ] ]
  [ WHERE { <search_condition>
    | { [ CURRENT OF
        { { [ GLOBAL ] cursor_name }
        | cursor_variable_name } ] } }
  ]
  [ OPTION ( <Query Hint> [ ,...n ] ) ]
[;]

<object> ::=
{
  [ server_name.database_name.schema_name.

```

```
| database_name. [ schema_name ] .  
| schema_name.  
]  
table_or_view_name  
}
```

Краће:

```
DELETE TABLE <naziv tabele>;
```

4.4. SQL-data-retrieve statements

Наредбе за издавање података. Алтернативно: наредбе за приказ података.

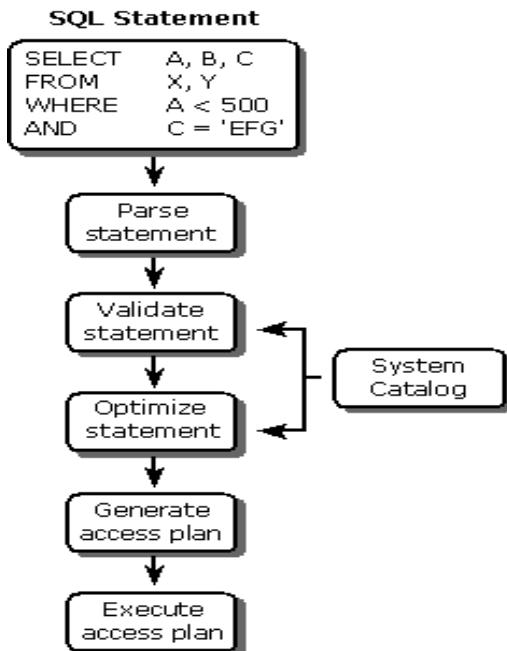
Некада: Data Retrieve Language (DRL) statements

SELECT (прикажи)

4.4.1. Поступак извршавања SQL упита

Приликом извршавања SQL упита (SQL statement, SQL query), систем за управљање базама података (СУБП) извршава следећих пет корака:

1. СУБП прво парсира SQL упит: дели упит у поједине речи (које называемо токени), проверава њихове међусобне синтаксне односе, проверава да ли се упит састоји из обавезних делова, да ли садржи кључне речи и да ли су оне тачно записане итд. У овом кораку се могу детектовати синтаксне грешке.
2. Потом СУБП врши валидацију SQL упита. При томе се користи каталог система (системски каталог, речник података). Да ли све табеле наведене у SQL упиту постоје у бази података? Да ли све наведене колоне постоје и да ли су имена колона недвосмислена? Да ли корисник има потребне привилегије да изврши овај упит? Одређене семантичке грешке могу се детектовати у овом кораку.
3. Ако је у претходном кораку све уреду, СУБП генерише план извршења SQL упита (*execution plan*, *(exp)*; алтернативни назив: приступни план, *access plan*). План извршења SQL упита је бинарна репрезентација акција које су потребне за обављање упита; то је СУБП еквивалент извршног кода програмских језика (*executable code*), тј. $(\text{exp})_{\text{DBMS}} = (\text{exe})_{\text{PL}}$.
4. СУБП оптимизује план извршења. Испитују се различити начине његовог извршења: Може ли индекс да се користи за убрзавање претраге? Да ли да се прво изврши рестрикција над Табелом А (уклоне редови који не одговарају услову претраге), па да се онда изврши спајање са табелом Б, или би било боље обрнуто (прво спајање, па онда рестрикција)? Може ли се секвенцијално претраживање целе табеле табеле избеги? Ако не може, како га ограничити на неки подскуп података из табеле?
5. Након процењивања различитих планова извршења, СУБП бира најефикаснији.



Слика 1. Кораци у извршењу SQL наредбе $(\text{exp})_{\text{DBMS}}$ је

Претходни кораци у извршавању SQL упита се разликују по приступу различитим објектима базе (каталог, индекси, кластери, табеле, погледи итд) и њиховој учесталости, па сходно томе и по количини времена које троше. Парсирање SQL упита не захтева приступ бази података и може да се уради веома брзо. Оптимизација је, с друге стране, процесорски веома захтевна активност и неопходан јој је приступ каталогу система. За сложене упите над више табела, у поступку оптимизацију, испитује се више хиљада различитих начина извођења истог упита. Међутим, трошкови неефикасног извршавања упита су обично толико високи, да се време проведено у оптимизација више него исплати, јер значајно повећава брзину извршења упита. Резултати оптимизације су још значајнији ако се изабрани (а то значи и оптимизовани) план извршења вишеструко извршава⁷¹.

⁷⁰ Резултат = подаци и/или порука

⁷¹ То се дешава приликом: (1) вишеструког извршавања истих упита, (2) извршавања параметризованих упита и (3) извршавања упита чија се структура не мења, већ се мењају само вредности у WHERE клузули.

4.4.2. Синтакса наредбе SELECT

Наредба SELECT убраја се у најсложеније SQL наредбе. Овде се наводи њен општи облик, без детаљних објашњења:

```
SELECT [ ALL | DISTINCT ]
[ TOP ( expression ) [ PERCENT ] [ WITH TIES ] ]
<select_list>
<select_list> ::= 
{
*
| { table_name | view_name | table_alias }.*
| {
    [ { table_name | view_name | table_alias }. ]
    { column_name | $IDENTITY | $ROWGUID }
    | udt_column_name [ { . | :: } { { property_name | field_name }
    | method_name ( argument [ ,...n] ) } ]
    | expression
    [ [ AS ] column_alias ]
}
| column_alias = expression
} [ ,...n ]
```

Краће, основни облик упита у SQL-у :

```
SELECT <lista kolona>
FROM <llista tabela>
WHERE <uslov>;
```

Листом колона задаје се операција *пројекције* жељених колона; листом табела одређује се извор података (који се добија *спајањем*, најчешће); условом се задаје услов *рестрикције* (*селекције*) жељених редова. Клаузуле SELECT и FROM су обавезне, а клаузула WHERE није.

5. Референце

ЛИТЕРАТУРА

- [1] Lazarević Branislav i dr., **Baze podataka**, FON, Beograd, 2003.
- [2] Codd Edgar F., **Data Models in Database Management**, објављено у: *Proceedings of Workshop on Data Abstraction, Databases, and Conceptual Modelling*, Michael L. Brodie & Stephen N. Yilles, eds., Pingree Park, Colo., June 1980.
- [3] Могин Павле и Луковић Иван, **Принципи база података**, ФТН и МП "Stylos", Нови Сад, 1996.
- [4] Codd Edgar F., **A Relational Model of Data for Large Shared Data Banks**, *Communications of the ACM*, Association for Computing Machinery, 13 (6): 377–87, June 1970.
- [5] Date Christopher J., **An Introduction to Database Systems**, 6th edition, Addison-Wesley, 1995.
- [6] Codd Edgar F., **Is Your DBMS Really Relational?**, *ComputerWorld*, 14. October 1985.
- [7] Codd Edgar F., **Does Your DBMS Run By the Rules?**, *ComputerWorld*, 21. October 1985.
- [8] Darwen Hugh and Date C. J., **Databases, Types, and The Relational Model: The Third Manifesto**, 3rd edition, Addison-Wesley, 2006.
- [9] Codd Edgar F., **The Relational Model for Database Management: Version 2**, Addison-Wesley, 1990.

ИНТЕРНЕТ

- [1] ISO/IEC FDIS 9075-1 Information technology - Database languages - SQL, Part 1: Framework
http://www.jtc1sc32.org/doc/N2151-2200/32N2153T-text_for_ballot-FDIS_9075-1.pdf
- [2] Data Types (Transact-SQL)
<https://msdn.microsoft.com/en-us/library/ms187752.aspx>
- [3] Proposed foundation for future (relational) database systems
<http://thethirdmanifesto.com>
- [4] Codd's 12 Rules (for a relational database product) <http://computing.derby.ac.uk/c/codds-twelve-rules/>

6. Прилог

У прилогу је изложен студијски примера: (скраћени) поступак развоја базе података која се користи за постављање упита. Имплементација је урађена у *Transact-SQL*-у (DBMS: *SQL Server 2012*, RADT: *SQL Server Management Studio*).

6.1. Концептуализација

6.1.1. Вербални модел

Неопходно је направити софтвер који омогућава рачунарски подржану евиденцију о основним ентитетима факултетског информационог система. Факултет има два одсека (студијска програма): Менаџмент и Информатика. На сваком од одсека постоје по три смера (студијске групе). На Информатику то су: Софтверско инжењерство, Информациони системи и Информационе технологије. На Менаџменту то су: Управљање производњом, Управљање пословањем и Управљање кадровима. Прва година студија је заједничка за све студенте (тј. студенти не бирају смер на првој години), а од друге године сваки студент се одлучује за један смер; није дозвољено студирање на више смерова истовремено. За сваког студента се води евиденција о следећим подацима:

- Број индекса (обавезно; прве две цифре значе годину уписа, а преостале три редни број),
- Презиме и име (обавезно),
- Средње име (није обавезно),
- Датум рођења (обавезно),
- Година студија (обавезно; од 1 до 4),
- Да ли студент поседује возачку дозволу и које је она категорије (није обавезно),
- Слика студента (није обавезна),
- Износ студентског кредита (обавезно, мада неки студенти не примају кредит),
- Износ стипендије (није обавезно).

Предмети на факултету имају свој назив, врсту (обавезни или изборни) и ЕСПБ. Сваки предмет у надлежности је једног од два наведена одсека („припада“ одсеку). Студент полаже предмете на испитима. На испиту се бележи датум полагања и оцена; петице се не уписују (не води се евиденција о неположеним испитима); у случају поништења испита и поновног полагања, уписује се само последња оцена (не води се историја полагања испита). Основни подаци о студентима (име, презиме, средње име), предметима (назив), смеровима (назив) и одсечима (назив), због законских обавеза, записани су ћириличним писмом српскога језика.

Врло често су потребни:

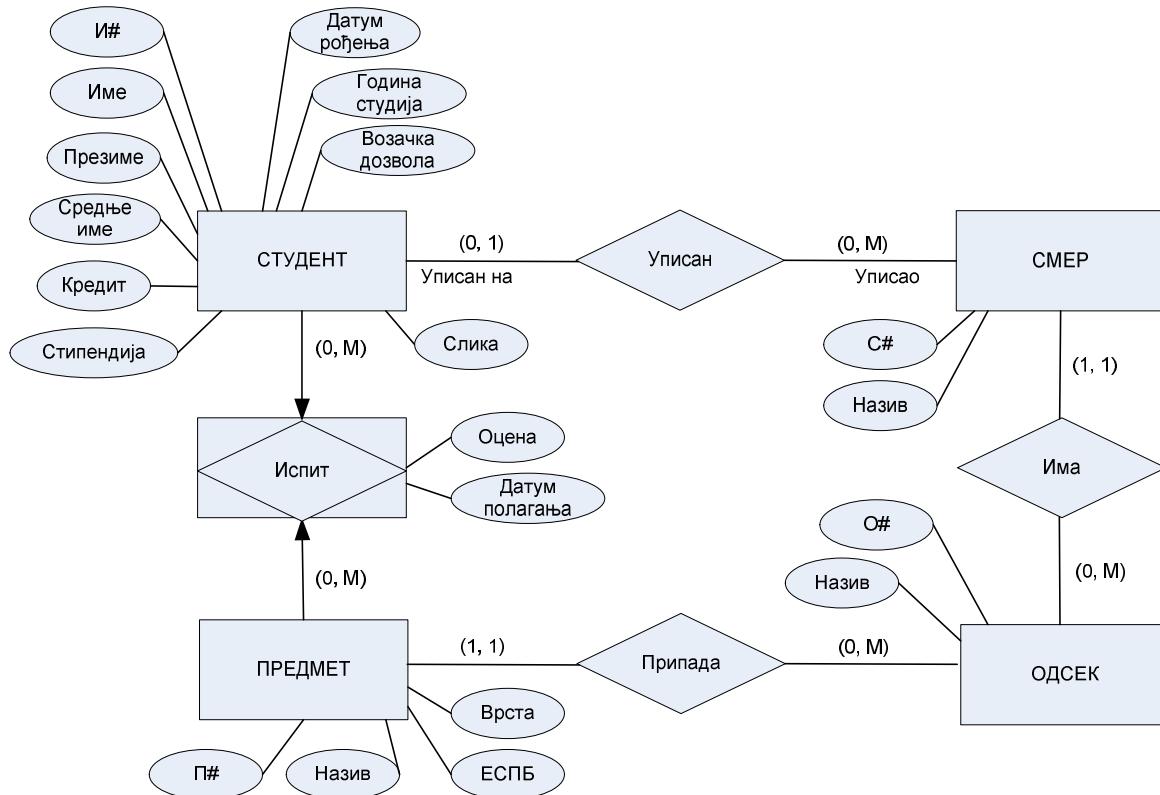
- Подаци о студентима са одсека Информатика (број индекса, име, презиме, година студија, одсек, а може и смер),
- Подаци о студентима са одсека Менаџмент (број индекса, име, презиме, година студија, одсек, а може и смер),
- Подаци о положеним испитима (који студент је положио, који предмет, када и са којом оценом),
- Подаци о студентским „примањима“ (стипендије, студентски кредити, укупно).

Сем претходно наведених, нема додатних посебних (пословних) ограничења.

6.1.2. Концептуални модел

6.1.2.1. Структура

А) Дијаграм објекта и веза



Концептуални модел: ДОВ Студент положио предмет

Б) Речник података

| ЕНТИТЕТ | АТРИБУТ | НАПОМЕНА |
|---------|---|---|
| ОДСЕК | О#: integer, not null Назив: nvarchar(12), not null | Шифра одсека |
| СМЕР | С#: integer, not null Назив: nvarchar(24), not null | Шифра смера |
| СТУДЕНТ | БИ: integer, not null Име: nvarchar(7), not null Презиме: nvarchar(7), not null СредњеИме: nvarchar(7), null ДатумРођења: date, not null Слика: varbinary(max), null ГодинаСтудија: integer, not null, [1..4] ВозачкаКатегорија: null, ('А','Б','Ц','Д','Е') Кредит: integer, not null, Кредит >= 0, default=0 Стипендија: integer, null Стипендија >= 0 | Број индекса |
| ПРЕДМЕТ | П#: integer, not null Назив: nvarchar(22), not null Врста: nchar(3), ('ИЗБ', 'ОБВ') ЕСПБ: integer, null, [2..8] | Шифра предмета Изборни, Обавезни Евро.сис.пренос.бодова |
| ИСПИТ | ДатумПолагања: date, not null, Оцена: integer, not null, [6..10] | |

6.1.2.2. Ограничева

Нема.

6.2. Спецификација

6.2.1. Схема релационе базе података: базне релације

ОДСЕК (О#, Назив)
 СМЕР (С#, Назив, Одсек#)
 СТУДЕНТ (БИ, Име, Презиме, СредњеИме, ДатумРођења, Слика, ГодинаСтудија,
 ВозачкаКатегорија, Кредит, Стипендија, Смер#)
 ПРЕДМЕТ (П#, Назив, Врста, ЕСПБ, Одсек#)
 ИСПИТ (БИ, П#, ДатумПолагања, Оцена)

6.2.2. Схема релационе базе података: изведене релације

ИНФ_СТУДЕНТ (БИ, Име, Презиме, ГодинаСтудија, Одсек), Одсек = `Информатика`
 МЕН_СТУДЕНТ (БИ, Име, Презиме, ГодинаСтудија, Одсек), Одсек = `Менаџмент`
 СТУД_ПОЛОЖИО (БИ, Име, Презиме, Предмет, Врста, ЕСПБ, ДатумПолагања, Оцена)
 СТУД_ФИНАНС (БИ, Име, Презиме, Кредит, Стипендија, Укупно)

6.2.3. Правила интегритета

Нема.

6.3. Имплементација

6.3.1. DDL

DDL = Data Definition Language

```
-- *****
-- Креирање базе података
-- *****

CREATE DATABASE fakultet
    COLLATE Serbian_Cyrillic_100_CS_AI;
USE fakultet;

-- *****
-- Креирање базних табела
-- *****

CREATE TABLE ODSEK(
    O#      tinyint not null primary key,
    Naziv   nvarchar(12) not null
);

CREATE TABLE SMER(
    S#      tinyint not null primary key,
    Naziv   nvarchar(24) not null,
    Odsek#  tinyint not null foreign key references ODSEK(O#)
);

CREATE TABLE STUDENT(
    I#      smallint not null primary key,          -- број индекса
    Ime     nvarchar(7) not null,
    Prezime nvarchar(7) not null
        check (Prezime LIKE '[А-Ш]%' ),
    Slika   varbinary(max) null,
    DatRod date not null,                          -- датум рођења
    GodStud tinyint not null                      -- година студија
        check ((GodStud >= 1) and (GodStud <= 4)),
    
```

```

VozKat  nchar(1) null                                -- возачка категорија
        check (VozKat in ('А', 'Б', 'Ц', 'Д', 'Е')),
Kredit   tinyint not null                            -- студентски кредит
        default 0
        check (Kredit >= 0),
Stip    tinyint null                               -- стипендија
        check ((Stip > 0) or (Stip is NULL)),
Smer#   tinyint null
        foreign key references SMER(S#)
);

ALTER TABLE STUDENT
DROP COLUMN Slika;

ALTER TABLE STUDENT
ADD SredIme nvarchar(7) null;                      -- средње име

CREATE TABLE PREDMET(
P#      tinyint not null primary key,
Naziv  nvarchar(22) not null unique,
Vrsta   nchar(3) not null check (Vrsta in ('ИЗБ', 'ОБВ')),
                    -- (ИЗБ)орни или (ОБ)а(В)езни предмет
ESPB    tinyint null,    -- (Е)вропски (С)истем (П)реносних (Б)одова
Odsek#  tinyint not null
        foreign key references ODSEK(O#)
);

CREATE TYPE tocena
FROM tinyint not null;

CREATE TABLE ISPIT(
Indeks#  smallint not null,           -- број индекса
Predmet#  tinyint not null,           -- шифра предмета
DatPol    date not null,              -- датум полагања испита
Ocena     tocena
        check ((Ocena >= 6) and (Ocena <= 10)),
constraint PK_Ispit primary key (Indeks#, Predmet#),
constraint FK_Ispit_Student foreign key (Indeks#) references STUDENT,
constraint FK_Ispit_Predmet foreign key (Predmet#) references PREDMET
);

-- *****
-- Креирање виртуелних табела (погледа)
-- *****

CREATE VIEW INF_STUDENT AS
SELECT
    I# as [Број индекса],
    Ime as [Име],
    Prezime as [Презиме],
    GodStud as [Година студија],
    SM.Naziv as [Смер],
    O.Naziv as [Одсек]
FROM STUDENT ST
JOIN SMER SM ON ST.Smer# = SM.S#
JOIN ODSEK O ON SM.Odsek# = O.O#
WHERE O.Naziv LIKE N'Информатика';

```

```

CREATE VIEW MEN_STUDENT AS
SELECT
    I# as [Број индекса],
    Ime as [Име],
    Prezime as [Презиме],
    GodStud as [Година студија],
    SM.Naziv as [Смер],
    O.Naziv as [Одсек]
FROM STUDENT ST
JOIN SMER SM ON ST.Smer# = SM.S#
JOIN ODSEK O ON SM.Odsek# = O.O#
WHERE O.Naziv LIKE N'Менаџмент';

CREATE VIEW STUDENT_POLOZIO AS
SELECT
    ST.I# as [Број индекса],
    Ime as [Име],
    Prezime as [Презиме],
    Naziv as [Предмет],
    Vrsta as [Врста],
    ESPB as [ЕСПБ],
    DatPol as [Датум испита],
    Ocena as [Оцена]
FROM STUDENT ST
JOIN ISPIT I ON ST.I# = I.Indeks#
JOIN PREDMET P ON I.Predmet# = P.P#;

CREATE VIEW STUDENT_FINANS AS
SELECT
    I# as [Број индекса],
    Ime as [Име],
    Prezime as [Презиме],
    Kredit as [Кредит],
    ISNULL(Stip, 0) as [Стипендија],
    [Укупно] = (Kredit + ISNULL(Stip, 0))
FROM STUDENT;

```

6.3.2. DML

DML = Data Manipulation Language

```

USE fakultet;

INSERT INTO ODSEK VALUES
    (100, N'Информатика'),
    (200, N'Менаџмент');

INSERT INTO SMER VALUES
    (10, 'Софтверско инжењерство', 100),
    (20, 'Информациони системи', 100),
    (30, 'Информационе технологије', 100),
    (50, 'Управљање производњом', 200),
    (60, 'Управљање пословањем', 200),
    (70, 'Управљање кадровима', 200);

INSERT INTO STUDENT
(I#, Ime, Prezime, SredIme, DatRod, GodStud, VozKat, Kredit, Stip, Smer#)
VALUES
    (17002, 'Ана', 'Костић', NULL, '1998.07.27', 1, NULL, 100, NULL, NULL),
    (17014, 'Ана', 'Марић', 'Јова', '1998.05.16', 1, 'Б', 100, NULL, NULL),

```

```
(17008, 'Анка', 'Анић', 'Саша', '1998.05.23', 1, NULL, 100, 50, NULL),
(16002, 'Аница', 'Барић', NULL, '1997.09.23', 2, 'Б', 150, 20, 10),
(16014, 'Мара', 'Илић', 'Мита', '1998.09.11', 2, 'А', 150, NULL, 20),
(16008, 'Мила', 'Јовић', 'Саша', '1998.07.27', 2, 'Ц', default, 50, 60),
(15002, 'Аца', 'Костић', 'Јова', '1996.06.17', 3, NULL, 200, NULL, 10),
(15014, 'Мома', 'Којић', NULL, '1996.06.17', 3, 'Б', 200, 20, 20),
(15008, 'Јова', 'Кун', 'Саша', '1996.12.12', 3, NULL, 200, NULL, 50),
(14002, 'Лаза', 'Марић', 'Раша', '1995.02.21', 4, NULL, 220, 20, 10),
(14014, 'Јова', 'Киш', NULL, '1995.03.23', 4, 'Ц', 220, 20, 50);
```

>>> . . .

6.3.3. RDB

RDB = Relational Database (релациона база података)

ОДСЕК

0# Назив

```
-----  
100 Информатика  
200 Менаџмент
```

СМЕР

| C# | Назив | Одсек# |
|----|--------------------------|--------|
| 10 | Софтверско инжењерство | 100 |
| 20 | Информациони системи | 100 |
| 30 | Информационе технологије | 100 |
| 50 | Управљање производњом | 200 |
| 60 | Управљање пословањем | 200 |
| 70 | Управљање кадровима | 200 |

СТУДЕНТ

| I# | Име | Презиме | СредИме | ДатРод | ГодСтуд | ВозКат | Кредит | Стип | Смер# |
|-------|-------|---------|---------|------------|---------|--------|--------|------|-------|
| 14002 | Лаза | Марић | Раша | 1995-02-21 | 4 | NULL | 220 | 20 | 10 |
| 14014 | Јова | Киш | NULL | 1995-03-23 | 4 | Ц | 220 | 20 | 50 |
| 15002 | Аца | Костић | Јова | 1996-06-17 | 3 | Б | 200 | NULL | 10 |
| 15008 | Јова | Кун | Саша | 1996-12-12 | 3 | NULL | 200 | NULL | 50 |
| 15014 | Мома | Којић | NULL | 1996-06-17 | 3 | Б | 200 | 20 | 20 |
| 16002 | Аница | Барић | NULL | 1997-09-23 | 2 | Б | 150 | 20 | 10 |
| 16008 | Мила | Јовић | Саша | 1998-07-27 | 2 | Ц | 0 | 50 | 60 |
| 16014 | Мара | Илић | Мита | 1998-09-11 | 2 | А | 150 | NULL | 20 |
| 17002 | Ана | Костић | NULL | 1998-07-27 | 1 | NULL | 100 | NULL | NULL |
| 17008 | Анка | Анић | Саша | 1998-05-23 | 1 | Б | 100 | 50 | NULL |
| 17014 | Ана | Марић | Јова | 1998-05-16 | 1 | Б | 100 | NULL | NULL |

ПРЕДМЕТ

| P# | Назив | Врста | ЕСПБ | Одсек# |
|----|----------------|-------|------|--------|
| 1 | Математика 1 | ОБВ | 6 | 100 |
| 2 | Програмирање 1 | ОБВ | 6 | 100 |
| 3 | Економија | ОБВ | 8 | 200 |
| 4 | Увод у ИС | ОБВ | 6 | 100 |

| | | | |
|---------------------------|-----|---|-----|
| 5 Основи организације | ОБВ | 6 | 200 |
| 6 Менаџмент | ОБВ | 6 | 100 |
| 7 Базе података | ОБВ | 6 | 100 |
| 8 Социологија | ИЗБ | 4 | 200 |
| 9 Психологија | ИЗБ | 4 | 200 |
| 10 Логика | ИЗБ | 4 | 200 |
| 11 Философија | ИЗБ | 4 | 200 |
| 12 Оперативни системи | ОБВ | 5 | 100 |
| 13 Основи квалитета | ОБВ | 6 | 200 |
| 14 Архитектура рачунара | ОБВ | 5 | 100 |
| 15 Производно инжењерство | ОБВ | 6 | 200 |
| 16 Математика 2 | ОБВ | 5 | 100 |
| 17 Програмирање 2 | ОБВ | 4 | 100 |
| 18 Маркетинг | ОБВ | 6 | 200 |
| 19 Финансије | ОБВ | 6 | 200 |
| 20 Алгоритми | ИЗБ | 4 | 100 |
| 21 Структуре података | ОБВ | 6 | 100 |
| 22 Управљање залихама | ОБВ | 6 | 200 |

ИСПИТ

| И# | П# | Дат | Пол | Оцена |
|-------|----|------------|-----|-------|
| 14002 | 1 | 2014-06-26 | | 9 |
| 14002 | 2 | 2016-01-28 | | 6 |
| 14002 | 3 | 2014-01-28 | | 6 |
| 14002 | 5 | 2014-01-28 | | 6 |
| 14002 | 16 | 2014-09-21 | | 8 |
| 14002 | 17 | 2015-01-28 | | 7 |
| 14014 | 1 | 2014-06-21 | | 6 |
| 14014 | 3 | 2014-06-27 | | 7 |
| ... | | ... | | |
| 17008 | 1 | 2017-09-29 | | 8 |
| 17008 | 5 | 2017-01-28 | | 10 |
| 17014 | 1 | 2017-06-29 | | 8 |
| 17014 | 2 | 2016-06-29 | | 8 |
| 17014 | 5 | 2017-06-28 | | 8 |
| 17014 | 16 | 2017-09-29 | | 8 |

6.3.4. ADS

ADS = Application Data Submodels

INF_STUDENT

| Број индекса | Име | Презиме | Година студија | Смер | Одсек |
|--------------|-------|---------|----------------|--------------------------|-------------|
| 14002 | Лаза | Марић | | 4 Софтверско инжењерство | Информатика |
| 15002 | Аца | Костић | | 3 Софтверско инжењерство | Информатика |
| 15014 | Мома | Којић | | 3 Информациони системи | Информатика |
| 16002 | Аница | Барић | | 2 Софтверско инжењерство | Информатика |
| 16014 | Мара | Илић | | 2 Информациони системи | Информатика |

MEN_STUDENT

| Број индекса | Име | Презиме | Година студија | Смер | Одсек |
|--------------|------|---------|----------------|-------------------------|-----------|
| 14014 | Јова | Киш | | 4 Управљање производњом | Менаџмент |

| | | | |
|------------|-------|-------------------------|-----------|
| 15008 Јова | Кун | З Управљање производњом | Менаџмент |
| 16008 Мила | Јовић | 2 Управљање пословањем | Менаџмент |

STUDENT_POLOZIO

| Број индекса | Име | Презиме | Предмет | Врста ЕСПБ | Датум испита | Оцена |
|--------------|-------|---------|---------------------|------------|--------------|-------|
| 14002 | Лаза | Марић | Математика 1 | ОБВ | 6 2014-06-26 | 9 |
| 14002 | Лаза | Марић | Програмирање 1 | ОБВ | 6 2016-01-28 | 6 |
| 14002 | Лаза | Марић | Економија | ОБВ | 8 2014-01-28 | 6 |
| 14002 | Лаза | Марић | Основи организације | ОБВ | 6 2014-01-28 | 6 |
| 14002 | Лаза | Марић | Математика 2 | ОБВ | 5 2014-09-21 | 8 |
| 14002 | Лаза | Марић | Програмирање 2 | ОБВ | 5 2015-01-28 | 7 |
| 14014 | Јова | Киш | Математика 1 | ОБВ | 6 2014-06-21 | 6 |
| 14014 | Јова | Киш | Економија | ОБВ | 8 2014-06-27 | 7 |
| 14014 | Јова | Киш | Основи организације | ОБВ | 6 2014-06-28 | 7 |
| 14014 | Јова | Киш | Математика 2 | ОБВ | 5 2014-09-21 | 6 |
| 15002 | Аца | Костић | Математика 1 | ОБВ | 6 2015-09-09 | 10 |
| 15002 | Аца | Костић | Програмирање 1 | ОБВ | 6 2016-09-26 | 7 |
| 15002 | Аца | Костић | Економија | ОБВ | 8 2015-09-26 | 7 |
| 15002 | Аца | Костић | Основи организације | ОБВ | 6 2015-09-28 | 7 |
| 15002 | Аца | Костић | Математика 2 | ОБВ | 5 2016-09-09 | 10 |
| 15002 | Аца | Костић | Програмирање 2 | ОБВ | 5 2016-09-26 | 7 |
| 15008 | Јова | Кун | Математика 1 | ОБВ | 6 2015-09-09 | 9 |
| 15008 | Јова | Кун | Економија | ОБВ | 8 2015-01-28 | 8 |
| 15008 | Јова | Кун | Основи организације | ОБВ | 6 2015-01-28 | 6 |
| 15008 | Јова | Кун | Математика 2 | ОБВ | 5 2015-09-29 | 9 |
| 15008 | Јова | Кун | Управљање залихама | ОБВ | 6 2016-01-28 | 6 |
| 15014 | Мома | Којић | Математика 1 | ОБВ | 6 2015-09-09 | 8 |
| 15014 | Мома | Којић | Програмирање 1 | ОБВ | 6 2016-06-27 | 9 |
| 15014 | Мома | Којић | Економија | ОБВ | 8 2015-06-27 | 7 |
| 15014 | Мома | Којић | Основи организације | ОБВ | 6 2015-06-28 | 8 |
| 15014 | Мома | Којић | Математика 2 | ОБВ | 5 2016-09-09 | 6 |
| 15014 | Мома | Којић | Програмирање 2 | ОБВ | 5 2016-06-27 | 8 |
| 16002 | Аница | Барич | Математика 1 | ОБВ | 6 2016-06-29 | 9 |
| 16002 | Аница | Барич | Програмирање 1 | ОБВ | 6 2016-09-26 | 10 |
| 16002 | Аница | Барич | Економија | ОБВ | 8 2016-09-26 | 8 |
| 16002 | Аница | Барич | Основи организације | ОБВ | 6 2016-09-28 | 9 |
| 16002 | Аница | Барич | Програмирање 2 | ОБВ | 5 2017-09-26 | 10 |
| 16008 | Мила | Јовић | Математика 1 | ОБВ | 6 2016-06-27 | 6 |
| 16008 | Мила | Јовић | Економија | ОБВ | 8 2016-01-24 | 9 |
| 16008 | Мила | Јовић | Основи организације | ОБВ | 6 2016-01-28 | 9 |
| 16008 | Мила | Јовић | Математика 2 | ОБВ | 5 2016-09-27 | 6 |
| 16014 | Мара | Илић | Математика 1 | ОБВ | 6 2016-01-28 | 7 |
| 16014 | Мара | Илић | Програмирање 1 | ОБВ | 6 2016-06-23 | 8 |
| 16014 | Мара | Илић | Основи организације | ОБВ | 6 2016-06-28 | 9 |
| 16014 | Мара | Илић | Математика 2 | ОБВ | 5 2016-06-28 | 7 |
| 16014 | Мара | Илић | Програмирање 2 | ОБВ | 5 2017-01-24 | 8 |
| 17002 | Ана | Костић | Математика 1 | ОБВ | 6 2017-06-24 | 6 |
| 17002 | Ана | Костић | Програмирање 1 | ОБВ | 6 2016-09-22 | 9 |
| 17002 | Ана | Костић | Економија | ОБВ | 8 2017-09-22 | 10 |
| 17002 | Ана | Костић | Основи организације | ОБВ | 6 2017-09-28 | 8 |
| 17002 | Ана | Костић | Програмирање 2 | ОБВ | 5 2016-09-22 | 9 |
| 17008 | Анка | Анић | Математика 1 | ОБВ | 6 2017-09-29 | 8 |
| 17008 | Анка | Анић | Основи организације | ОБВ | 6 2017-01-28 | 10 |
| 17014 | Ана | Марић | Математика 1 | ОБВ | 6 2017-06-29 | 8 |
| 17014 | Ана | Марић | Програмирање 1 | ОБВ | 6 2016-06-29 | 8 |
| 17014 | Ана | Марић | Основи организације | ОБВ | 6 2017-06-28 | 8 |
| 17014 | Ана | Марић | Математика 2 | ОБВ | 5 2017-09-29 | 8 |

STUDENT_FINANS

| Број индекса | Име | Презиме | Кредит | Стипендија | Укупно |
|--------------|-------|---------|--------|------------|--------|
| 14002 | Лаза | Марић | 220 | 20 | 240 |
| 14014 | Јова | Киш | 220 | 20 | 240 |
| 15002 | Аца | Костић | 200 | 0 | 200 |
| 15008 | Јова | Кун | 200 | 0 | 200 |
| 15014 | Мома | Којић | 200 | 20 | 220 |
| 16002 | Аница | Барић | 150 | 20 | 170 |
| 16008 | Мила | Јовић | 0 | 50 | 50 |
| 16014 | Мара | Илић | 150 | 0 | 150 |
| 17002 | Ана | Костић | 100 | 0 | 100 |
| 17008 | Анка | Анић | 100 | 50 | 150 |
| 17014 | Ана | Марић | 100 | 0 | 100 |

6.3.5. DRL

DRL = Data Retrieve Language

```
=====
- 1. УПИТИ НАД ЈЕДНОМ ТАБЕЛОМ КОЈИМА СЕ ПРИКАЗУЈЕ ЈЕН НЕИЗМЕЊЕН САДРЖАЈ
-- 1.0. ЈЕДНОСТАВАН ПРИКАЗ НЕИЗМЕЊЕНОГ САДРЖАЈА
-- 1.1. УПОТРЕБА WHERE КЛАУЗУЛЕ
-- 1.2. КОРИШЋЕЊЕ ОЗНАКЕ NULL
- 2. УПИТИ НАД ЈЕДНОМ ТАБЕЛОМ КОЈИМА СЕ ПРИКАЗУЈЕ ЈЕН ИЗМЕЊЕН САДРЖАЈ
-- 2.1. ФУНКЦИЈЕ
--- 2.1.1. ФУНКЦИЈЕ: Сумарне нумеричке
--- 2.1.2. ФУНКЦИЈЕ: За рад са нискама (стринговима)
--- 2.1.3. ФУНКЦИЈЕ: За пребројавање (утврђивање кардиналности)
-- 2.2. УПОТРЕБА GROUP BY КЛАУЗУЛЕ
-- 2.3. УПОТРЕБА HAVING КЛАУЗУЛЕ
-- 2.4. УПИТИ ИЗ КЛАУЗУЛЕ FROM
-- 2.5. УПИТИ са КЛАУЗУЛОМ INTO (SELECT...INTO)
-- 2.6. ПРИВРЕМЕНЕ ТАБЕЛЕ
-- 2.7. CASE ИЗРАЗИ
- 3. УПИТИ НАД ДВЕ ИЛИ ВИШЕ ТАБЕЛА
--3.1. ПОДУПИТИ
--- 3.1.1. ЈЕДНОСТАВНИ ПОДУПИТИ
---- 3.1.1.1. ЈЕДНОСТАВНИ ПОДУПИТИ ЗАСНОВАНИ НА УПОТРЕБИ ОПЕРАТОРА ПОРЕЂЕЊА
---- 3.1.1.2. ЈЕДНОСТАВНИ ПОДУПИТИ ЗАСНОВАНИ НА УПОТРЕБИ ОПЕРАТОРА IN
---- 3.1.1.3. ЈЕДНОСТАВНИ ПОДУПИТИ ЗАСНОВАНИ НА УПОТРЕБИ ОПЕРАТОРА ANY и ALL
--- 3.1.2. КОРЕЛИСАНИ ПОДУПИТИ
---- 3.1.2.1. КОРЕЛИСАНИ ПОДУПИТИ ЗАСНОВАНИ НА УПОТРЕБИ ФУНКЦИЈЕ EXISTS
-- 3.2. ОПЕРАТОРИ СПАЈАЊА
-- 3.3. РЕКУРЗИВНИ УПИТИ
-- 3.4. СКУПОВНИ ОПЕРАТОРИ
=====
```

```
=====
-- 1. УПИТИ НАД ЈЕДНОМ ТАБЕЛОМ КОЈИМА СЕ ПРИКАЗУЈЕ ЈЕН НЕИЗМЕЊЕН САДРЖАЈ
=====
```

```
USE fakultet;
GO
```

```
=====
-- 1.0. ЈЕДНОСТАВАН ПРИКАЗ НЕИЗМЕЊЕНОГ САДРЖАЈА
=====
```

```
-- ПРИМЕР. /Приказ свих колона једне табеле/
-- Приказати све податке о свим особама.
```

```
SELECT * FROM PREDMET;
GO
```

```
-- ПРИМЕР. /ПРОЈЕКЦИЈА: Приказ одређених колона једне табеле/
-- Приказати индексе, имена и презимена свих студената.
```

```
SELECT I#, Ime, Prezime
FROM STUDENT;
GO
```

```
-- ПРИМЕР. /Уређени (сортиран) приказ података/
-- Приказати сва презимена студената, уређена у растућем редоследу.
```

-- Прво, приказ свих презимена:

```
SELECT ALL Prezime
```

```
FROM STUDENT;
```

-- НАПОМЕНА: Опција ALL је подразумевана вредност, па се не мора писати.

-- Друго, уређени приказ свих презимена:

```
SELECT ALL Prezime
```

```
    FROM STUDENT
```

```
    ORDER BY Prezime;
```

```
GO
```

-- ПРИМЕР. /Уређени приказ података по више критеријума/

-- Приказати сва презимена, имена и године студија студената,

-- уређена у растућем редоследу, прво по презимену, а затим по имену.

```
SELECT Prezime, Ime, GodStud
```

```
    FROM STUDENT
```

```
    ORDER BY Prezime, Ime;
```

```
GO
```

-- ПРИМЕР. /Уређени приказ података у опадајућем редоследу/

-- Приказати сва презимена студената, уређена у опадајућем редоследу.

-- растући редослед: ASC(ending) = default order

```
SELECT Prezime
```

```
    FROM STUDENT
```

```
    ORDER BY Prezime ASC;
```

-- опадајући редослед: DESC(ending)

```
SELECT Prezime
```

```
    FROM STUDENT
```

```
    ORDER BY Prezime DESC;
```

-- опадајући редослед за више колона

```
SELECT Prezime, Ime
```

```
    FROM STUDENT
```

```
    ORDER BY Prezime DESC, Ime DESC;
```

-- опадајући редослед по презимену, растући по имену

```
SELECT Prezime, Ime
```

```
    FROM STUDENT
```

```
    ORDER BY Prezime DESC, Ime ASC;
```

```
GO
```

-- ПРИМЕР. /Приказ само различитих презимена/

-- Приказати сва презимена студената, без понављања истоветних.

-- Прво, приказ свих:

```
SELECT ALL Prezime FROM STUDENT ORDER BY Prezime;
```

-- А потом, приказ само различитих:

```
SELECT DISTINCT Prezime
```

```
    FROM STUDENT;
```

```
GO
```

-- ЗАДАТАК. Приказати сва имена студената, без понављања истих, у растућем редоследу.

```
=====
-- 1.1. УПОТРЕБА WHERE КЛАУЗУЛЕ
=====

-- ПРИМЕР. /СЕЛЕКЦИЈА: Приказ одређених редова једне табеле/
-- Приказати све податке о студентима који су са смера чија је Smer# = 10.
SELECT *
    FROM STUDENT
    WHERE Smer# = 10;
GO

-- ПРИМЕР. /Коришћење релационих оператора/
-- Приказати све податке о студентима...

-- ... који нису са смера чија је Smer# = 10.
SELECT * FROM STUDENT WHERE Smer# != 10;
-- Или:
SELECT * FROM STUDENT WHERE Smer# <> 10;
GO

-- ... чији је кредит већи од 200 н.ј.
SELECT *
    FROM STUDENT
    WHERE Kredit > 200;
GO

-- ... чији је кредит једнак 200 н.ј. или је виши.
SELECT *
    FROM STUDENT
    WHERE Kredit >= 200;
GO

-- ЗАДАТАК. Приказати све податке о студентима...

-- ... који су са смера чија је Smer# = 50.
-- ... стипендија није једнака 20 н.ј.
-- ... који нису више бруцоши (нису прва година студија).
-- ... који се презивају 'КОСТИЋ'. (да ли је свеједно да ли пише 'КОСТИЋ' или 'Костић'?  
Зашто?)
-- ... који се не презивају 'КОСТИЋ'.
-- ... који имају стипендију мању од 50 н.ј.

-- ПРИМЕР. /Коришћење логичког оператора AND/
-- Приказати име, презиме, годину студија и висину кредита свих студената са друге године
-- који имају кредит већи од 100 н.ј.
SELECT Ime, Prezime, GodStud, Kredit
    FROM STUDENT
    WHERE GodStud = 2
        AND Kredit > 100;
GO

-- ЗАДАТАК. Приказати име, презиме, возачку категорију и годину студија студената
-- који су старији од бруцоша и имају возачку дозволу која није категорије A.
SELECT Ime, Prezime, VozKat, GodStud
    FROM STUDENT
    WHERE GodStud
        AND VozKat ;
GO
```

```
-- ПРИМЕР. /Коришћење логичког оператора OR/
-- Приказати број индекса, име и презиме свих студената
-- који се презивају 'Костић' или 'Марић'.
SELECT I#, Ime, Prezime
    FROM STUDENT
    WHERE Prezime = 'Костић'
        OR Prezime = 'Марић';
GO

-- ЗАДАТАК. Приказати име, презиме, средње име и годину студија свих студената којима
-- се отац зове 'Саша', 'Раша' или 'Јова'.
-- Напомена: Приказује се 6 редова.
SELECT Ime, Prezime,
    FROM STUDENT
    WHERE   = ''
        OR   = ''
;
GO

-- ПРИМЕР. /Коришћење логичког оператора != или NOT/
-- Приказати име, презиме, годину студија и возачку дозволу свих
-- студената који нису прва година и немају возачку дозволу 'Б' категорије.

-- B.1.
SELECT Ime, Prezime, GodStud, VozKat
    FROM STUDENT
    WHERE GodStud != 1
        AND VozKat != 'Б';

-- B.2.
SELECT Ime, Prezime, GodStud, VozKat
    FROM STUDENT
    WHERE NOT (GodStud = 1)
        AND NOT (VozKat = 'Б');
GO

-- ЗАДАТАК. Приказати име, презиме и смер свих студената које нису на смеру који има шифру
10.
-- Напомена: задатак решити на два начина.
-- B.1.
SELECT Ime, Prezime,
    FROM STUDENT
    WHERE   != ;
-- B.2.
    SELECT Ime, Prezime, Пол
    FROM STUDENT
    WHERE NOT(   = );
GO

-- ЗАДАТАК. Приказати све податке о студентима које се не зову 'Јова'
-- и на трећој су години студија.

-- B.1.
SELECT *
    FROM STUDENT
    WHERE Ime != 'Јова'
        AND GodStud = 3;
```

-- B.2.

```
SELECT *
    FROM STUDENT
   WHERE NOT(Ime = 'Јова')
     AND NOT((GodStud = 1) OR (GodStud = 2) OR (GodStud = 4));
```

-- B.3.

```
SELECT *
    FROM STUDENT
   WHERE NOT(Ime = 'Јова')
     AND NOT(GodStud IN (1, 2, 4));
GO
```

-- ПРИМЕР. /Коришћење заграда да би се дефинисао редослед примене логичких оператора/

-- Приказати име, презиме, годину студија и возвачку категорију за све студенте са

-- 2. или 3. године, а који имају Б категорију.

```
SELECT Ime, Prezime, GodStud, VozKat
    FROM STUDENT
   WHERE (GodStud = 2 OR GodStud = 3)
     AND VozKat = 'Б';
```

GO

-- Питање: Зашто следећи упит не враћа исте редове као и претходни?

```
SELECT Ime, Prezime, GodStud, VozKat
    FROM STUDENT
   WHERE GodStud = 2
     OR GodStud = 3
     AND VozKat = 'Б';
```

GO

-- Одговор: Зато што се он извршава као следећи упит:

```
SELECT Ime, Prezime, GodStud, VozKat
    FROM STUDENT
   WHERE GodStud = 2
     OR (GodStud = 3 AND VozKat = 'Б');
```

-- Разлог: оператор AND има већи приоритет од оператора OR.

GO

-- ЗАДАТAK. Приказати име, презиме, средње име и годину студија свих студената који

-- се презивају Марић, Костић или Киш, а нису студенти 4. године .

```
SELECT Ime, Prezime, SredIme, GodStud
    FROM STUDENT
   WHERE (Prezime = 'Марић' OR Prezime = 'Костић' OR Prezime = 'Киш')
     AND GodStud != 4;
```

GO

-- ПРИМЕР. /Коришћење клаузиле BETWEEN која проверава да ли су тражене вредности

-- у наведеном распону/

-- Приказати име, средње име, презиме, кредит и годину студија за све студенте који

-- имају кредит између 100 н.ј. и 200 н.ј.

```
SELECT Ime, SredIme, Prezime, Kredit, GodStud
    FROM STUDENT
   WHERE Kredit BETWEEN 100 AND 200;
```

GO

-- ЗАДАТAK. Приказати шифре и називе свих смерова чије се

-- шифре налазе у опсегу од 20 до 60.

```
-- ПРИМЕР. /Коришћење оператора IN (који мења вишеструку примену оператора OR)/
-- Претходни задатак:
-- Приказати име, презиме, средње име и годину студија свих студената који
-- се презивају Марић, Костић или Киш, а нису студенти 4. године.

-- Претходно решење:
SELECT Ime, Prezime, SredIme, GodStud
  FROM STUDENT
 WHERE (Prezime = 'Марић' OR Prezime = 'Костић' OR Prezime = 'Киш')
   AND GodStud != 4;

-- Ново решење:
SELECT Ime, Prezime, SredIme, GodStud
  FROM STUDENT
 WHERE Prezime IN ('Марић', 'Костић', 'Киш')
   AND GodStud != 4;
GO

-- ЗАДАТАК. Приказати шифру, назив и број бодова (ЕСПБ) свих предмета
-- који немају 6, 8 или 4 ЕСПБ. (Употреба: NOT IN)

-- ПРИМЕР. /Коришћење клаузуле LIKE/
-- Претраживање на основу узора (string pattern). Специјални знаци:
-- % ::= ниска (стринг) од 0 или више знакова
-- _ ::= позиција (место) једног знака
-- [] ::= списак или опсег знакова

-- Приказати све податке о студентима које се презивају Костић.
SELECT *
  FROM STUDENT
 WHERE Prezime LIKE 'Костић';
GO
-- Приказати све податке о предметима чији назив почиње знаком 'П'.
SELECT *
  FROM PREDMET
 WHERE Naziv LIKE 'П%';
GO
-- Приказати имена и презимена свих студената којима се презиме завршава
-- знаковима 'ић' и презима има тачно пет знакова.
SELECT *
  FROM STUDENT
 WHERE Prezime LIKE '__ић'; -- у узору су три доње цртице
GO

-- ЗАДАТАК. Приказати имена и презимена свих студената којима...
-- ... име почиње ниском 'Ан'.
SELECT Ime, Prezime
  FROM STUDENT
 WHERE Ime LIKE 'Ан%';
GO
-- ... се име завршава ниском 'ва'.
SELECT Ime, Prezime
  FROM STUDENT
 WHERE Ime LIKE '%ва';
GO
-- ... се средње име завршава ниском 'ша' и има тачно четири знака.
SELECT Ime, Prezime, SredIme
```

```
        FROM STUDENT
        WHERE SredIme LIKE '__ша';
GO
-- ... се презиме не завршава на 'ић'.
SELECT Ime, Prezime
      FROM STUDENT
     WHERE Prezime NOT LIKE '%ић';
GO
-- ... је други знак презимена 'а'.
SELECT Ime, Prezime
      FROM STUDENT
     WHERE Prezime LIKE '_а%';
GO
-- ... презиме има тачно пет знакова.
SELECT Ime, Prezime
      FROM STUDENT
     WHERE Prezime LIKE '_____';
GO

-- ЗАДАТАК. Приказати имена и презимена свих студената којима...

-- ... је у презимену знак 'о' пре знака 'и'.
SELECT Ime, Prezime
      FROM STUDENT
     WHERE Prezime LIKE '%о%и%';
GO
-- А шта ако презиме почиње знаком 'О' (великим знаком 'о')?
INSERT INTO STUDENT
    (I#, Ime, Prezime, SredIme, DatRod, GodStud, VozKat, Kredit, Stip, Smer#)
VALUES
    (12002, 'Ранко', 'Орлић', NULL, '1998.07.27', 1, NULL, 100, NULL, NULL);
GO
-- Онда упит изгледа овако:
SELECT Ime, Prezime
      FROM STUDENT
     WHERE Prezime LIKE 'О%и%' OR Prezime LIKE '%о%и%';
GO

-- ... није у презимену знак 'о'/'О' пре знака 'и'.
SELECT I#, Ime, Prezime
      FROM STUDENT
     WHERE Prezime NOT LIKE 'О%и%' OR Prezime NOT LIKE '%о%и%' /* Овако не може */
     WHERE NOT (Prezime LIKE 'О%и%' OR Prezime LIKE '%о%и%'); /* Овако може */
     WHERE Prezime NOT LIKE 'О%и%' AND Prezime NOT LIKE '%о%и%' /* Овако може */
GO

-- Враћамо се у претходно стање:
DELETE FROM STUDENT WHERE I# = 12002;
GO

-- ПРИМЕР. /Употреба опсега карактера/
-- Приказати све предмете којима назив почиње неким
-- од знакова из (азбучног) опсега Б-О.

SELECT *
      FROM PREDMET
     ORDER BY Naziv;

SELECT *
```

```

    FROM  PREDMET
    WHERE Naziv LIKE '[Б-О]%'
        ORDER BY Naziv;
GO

-- ЗАДАТАК. Приказати имена, презимена и годину студија свих студената којима презиме
-- почиње неким од карактера из (азбучног) опсега Ј-Л.
SELECT
    Prezime +' '+ Ime AS [Презиме и име], GodStud AS [Година студија]
    FROM  STUDENT
    WHERE Prezime LIKE '[Ј-Л]%' ;
GO

-- ЗАДАТАК. Приказати имена, презимена и године студија свих студената којима презиме
-- почиње неким од карактера из (азбучног) опсега А-И и М-Ш.
SELECT
    Prezime +' '+ Ime AS [Презиме и име], GodStud AS [Година студија]
    FROM  STUDENT
    WHERE Prezime LIKE '[А-И, М-Ш]%' ;
ORDER BY 1;
GO

-- ПРИМЕР. /Употреба карактера изван опсега /
-- Приказати све предмете којима назив не почиње неким
-- од знакова из (азбучног) опсега Б-О.

SELECT *
    FROM  PREDMET
        ORDER BY Naziv;

SELECT *
    FROM  PREDMET
    WHERE Naziv LIKE '[^Б-О]%' ;
ORDER BY Naziv;
GO

-- ЗАДАТАК. Приказати имена, презимена и године студија свих студената...
-- ... којима презиме не почиње неким од карактера 'Ј', 'К', 'Л' и 'М',
-- а уз то им име не почиње карактером 'А' или 'Б'.

-- К.1.
SELECT Prezime, Ime, GodStud
    FROM  STUDENT
        ORDER BY Prezime;
-- К.2.
SELECT
    Prezime +' '+ Ime AS [Презиме и име], GodStud AS [Година студија]
    FROM  STUDENT
    WHERE Prezime LIKE '[^Ј-М]%' ;
ORDER BY Prezime;
-- К.3.
SELECT
    Prezime +' '+ Ime AS [Презиме и име], GodStud AS [Година студија]
    FROM  STUDENT
    WHERE Prezime LIKE '[^Ј-М]%' 
        AND Ime LIKE '[^АБ]%' ;
ORDER BY Prezime;
GO

```

```
-- ...који су рођени 1996. године.  
-- Напомена: Тип DATE је низ знакова одређеног формата.  
SELECT Prezime, Ime, GodStud, DatRod  
      FROM STUDENT  
     WHERE DatRod LIKE '1996%';  
GO  
  
-- ...који нису рођени у месецу мају, без обзира које године.  
SELECT Prezime, Ime, GodStud, DatRod  
      FROM STUDENT  
     WHERE DatRod LIKE '%-[01][^5]-%';  
GO  
  
-- ...који имају кредит између 100 и 150 н.ј.  
-- B.1.  
SELECT Prezime, Ime, GodStud, Kredit  
      FROM STUDENT  
     WHERE Kredit LIKE '1[012345][0-9]';  
-- B.2.  
SELECT Prezime, Ime, GodStud, Kredit  
      FROM STUDENT  
     WHERE Kredit BETWEEN 100 AND 150;  
GO  
  
-- ...који имају кредит између 200 и 230 н.ј., али не и оне који имају кредит 220 н.ј.  
-- B.1.  
SELECT Prezime, Ime, GodStud, Kredit  
      FROM STUDENT  
     WHERE Kredit BETWEEN 200 AND 230  
           AND Kredit LIKE '2[^2]0';  
-- B.2.  
SELECT Prezime, Ime, GodStud, Kredit  
      FROM STUDENT  
     WHERE Kredit LIKE '2[0-3][0-9]' -- 200 201 202 203 ... 209 210 211 212 ... 219  
(220) 221 222 ... 229 230  
           AND Kredit LIKE '2[^2]0';  
GO  
  
=====  
-- 1.2. КОРИШЋЕЊЕ ОЗНАКЕ NULL  
=====  
/*  
NULL означава недефинисану вредност =>  
    - NULL је ознака,  
    - Ознака NULL није вредност,  
    - Ознака NULL није једнака нули.  
  
Ознака NULL има две врсте значења:  
    - још увек непозната вредност и  
    - непримениљиво својство.  
Без обзира на врсту значења, вредности одређене колоне могу се проверавати да ли садрже  
ознаку NULL, само помоћу две специјалне клаузуле:  
    - IS NULL и  
    - IS NOT NULL.  
Оператори поређења се не могу користити за 'тестирање на' NULL, јер NULL није вредност,  
већ ознака да недостаје вредност.  
*/
```

```
-- ПРИМЕР. /Употреба клаузуле IS NULL/
-- Приказати име, средње име и презиме свих особе којима је непознато средње име.
SELECT Ime, SredIme, Prezime
    FROM STUDENT
    WHERE SredIme IS NULL;
GO

-- ЗАДАТAK. Приказати име, презиме, средње име, годину студија и возачку категорију
-- свих студената за које је непознато да ли имају/немају возачку категорију.
SELECT Ime, SredIme, Prezime, GodStud, VozKat
    FROM STUDENT
    WHERE VozKat IS NULL;
GO

-- Следећи упит јесте синтаксно исправан, али семантички НИЈЕ исправан!
-- Зато се не приказује ни један ред.
SELECT Ime, SredIme, Prezime, GodStud, VozKat
    FROM STUDENT
    WHERE VozKat = NULL;
GO

-- ПРИМЕР. /Употреба клаузуле IS NOT NULL/
-- Приказати име, средње име и презиме свих студената којима је познато средње име.
SELECT Ime, SredIme, Prezime
    FROM STUDENT
    WHERE SredIme IS NOT NULL;
GO

-- Следећи упит није исправан! Зашто?
SELECT Ime, SredIme, Prezime, GodStud, VozKat
    FROM STUDENT
    WHERE VozKat != NULL;
GO

-- ПРИМЕР. /Употреба ф-је ISNULL(): замена NULL-а задатом вредношћу./
-- Приказати имена, средња имена и презимена свих студената.
-- Уколико средње име није познато приказати карактер '/'.
-- Синтакса ф-је: ISNULL(check_expression, replacement_value)
SELECT
    Ime,
    SredIme,
    Prezime
    FROM STUDENT;

SELECT
    Ime,
    ISNULL(SredIme, '/') as [Средње име],
    Prezime
    FROM STUDENT;

-- или:
SELECT
    Ime,
    [Средње име] = ISNULL(SredIme, '/'),
    Prezime
    FROM STUDENT;
GO
```

-- ПРИМЕР. Приказати име, презиме, годину студија и шифру смера за све студенте.
-- Ознаку NULL заменити ниском 'Непознат смер'.

```
SELECT  
    Ime,  
    Prezime,  
    GodStud,  
    ISNULL(CAST(Smer# as nvarchar(14)), 'Непознат смер') as [Šifra smera]  
FROM STUDENT;  
GO
```

-- ЗАДАТAK. Приказати име, презиме и стипендију за све студенте којима је
-- непознат износ стипендије. Ознаку NULL заменити ниском 'Непознато'.

```
SELECT  
    Ime,  
    Prezime,  
    Stip,  
    ISNULL(CAST(Stip as nvarchar(9)), 'Непознато') as Stipendija  
FROM STUDENT  
WHERE Stip IS NULL;  
GO
```

/*

Функција COALESCE()

=====

Аргумент φ-је COALESCE() је низ израза; φ-ја враћа вредност
првог не-NUL израза. Уколико су сви изрази NULL, φ-ја враћа NULL.

Синтакса:

```
-----  
COALESCE ( izraz [ ,...n ] )
```

Напомена: Функција COALESCE(izraz-1, izraz-2, ..., izraz-N) је еквивалентна
следећем CASE изразу:

```
.....  
CASE  
    WHEN (izraz-1 IS NOT NULL) THEN izraz-1  
    WHEN (izraz-2 IS NOT NULL) THEN izraz-2  
    ...  
    ELSE izraz-N  
END  
.....  
*/
```

-- ПРИМЕР. /Употреба φ-је COALESCE()/

-- Табела ISPLATA сатоји се од три колоне које садрже податке о годишњим исплатама
запослених.

-- Могућа су три начина обрачуна исплате: исплата по броју радних сати (колона: сатница),
исплата

-- утврђене годишње плате (колона: плата) и по оствареној продаји (колоне: провизија,
број_продаја).

-- Запослени прима исплату само по једном од наведених модела. Зато се у табели налазе
подаци

-- само за један од наведених модела за сваког од запослених. Израчунати колико ће примити
новца

-- свако од запослених.

USE tempdb;

```
IF OBJECT_ID('ISPLATA') IS NOT NULL  
    DROP TABLE ISPLATA;
```

GO

```

CREATE TABLE ISPLATA(
    zaposlen_id tinyint identity,
    satnica decimal NULL,
    plata decimal NULL,
    provizija decimal NULL,
    broj_prodaja tinyint NULL);
GO
INSERT ISPLATA (satnica, plata, provizija, broj_prodaja) VALUES
    (10.00, NULL, NULL, NULL),
    (20.00, NULL, NULL, NULL),
    (30.00, NULL, NULL, NULL),
    (40.00, NULL, NULL, NULL),
    (NULL, 10000.00, NULL, NULL),
    (NULL, 20000.00, NULL, NULL),
    (NULL, 30000.00, NULL, NULL),
    (NULL, 40000.00, NULL, NULL),
    (NULL, NULL, 15000, 3),
    (NULL, NULL, 25000, 2),
    (NULL, NULL, 20000, 6),
    (NULL, NULL, NULL, NULL),
    (NULL, NULL, 14000, 4);
GO
SELECT * FROM ISPLATA;
SELECT
    zaposlen_id,
    CAST(
        COALESCE(
            satnica * 40 * 52, -- 40 = broj radnih sati u sedmici; 52 = broj sedmica u
godini
            plata,
            provizija * broj_prodaja
        ) as money
    ) as 'Isplata'
FROM ISPLATA
ORDER BY 'Isplata';
GO
=====

-- 2. УПИТИ НАД ЈЕДНОМ ТАБЕЛОМ КОЈИМА СЕ ПРИКАЗУЈЕ ЊЕН ИЗМЕЊЕН САДРЖАЈ
=====

=====

-- 2.1. ФУНКЦИЈЕ
=====

/*
MS SQL SERVER FUNCTIONS
=====
A. BUILT-IN FUNCTIONS
    1. Rowset Functions
        OPENDATASOURCE, OPENQUERY, OPENROWSET, OPENXML
    2. Aggregate Functions
        AVG, CHECKSUM_AGG, COUNT, COUNT_BIG, GROUPING, GROUPING_ID,
        MAX, MIN, SUM, STDEV, STDEVP, VAR, VARP
    3. Ranking Functions
        RANK, DENSE_RANK, NTILE, ROW_NUMBER
    4. Scalar Functions
        4.1. Configuration Functions
        4.2. Conversion Functions

```

```
4.3. Cursor Functions
4.4. Date&Time Data Functions
4.5. Logical Functions
4.6. Mathematical Functions
4.7. Metadata Functions
4.8. Security Functions
4.9. String Functions
4.10. System Functions
4.11. System Statistical Functions
4.12. Text&Image Functions
B. USER-DEFINED FUNCTIONS
1. Transact-SQL Functions
    1.1. T-SQL Scalar Functions
    1.2. T-SQL Inline Table-valued Functions
    1.3. T-SQL Multistatement Table-valued Functions
2. CLR functions
    2.1. CLR Scalar Functions
    2.2. CLR Table-valued Functions
*/
=====
-- 2.1.1. ФУНКЦИЈЕ: Сумарне нумеричке
=====
/*
Функције за добијање сумарних података су:
    AVG(колона) - средња вредност
    SUM(колона) - укупна вредност
    MIN(колона) - минимална вредност
    MAX(колона) - максимална вредност
*/
USE fakultet;
GO

-- ПРИМЕР. Приказати минимални, максимални и средњи износ кредита који примају студенти.
SELECT I#, Ime, Prezime, Висина
FROM STUDENT;

SELECT MIN(Kredit), MAX(Kredit), AVG(Kredit)
FROM STUDENT;
-- Има резултата, али нема назива колона. Зато се резултатима агрегатних функција дају имена.

-- Има и резултата и назива колона:
SELECT
    MIN(Kredit) as [Минимални износ кредита],
    MAX(Kredit) as [Максимални износ кредит],
    AVG(Kredit) as [Просечан износ кредита]
FROM STUDENT;

-- Или (различита употреба алијаса):
SELECT
    [Минимални износ кредита] = MIN(Kredit),
    [Максимални износ кредит] = MAX(Kredit),
    [Просечан износ кредита] = AVG(Kredit)
FROM STUDENT;

-- ЗАДАТAK. Приказати просечну оцену коју су добили студенти на испиту за предмет
-- чија је шифра Predmet# = 1 (Математика 1).
GO
```

```
=====
-- 2.1.2. ФУНКЦИЈЕ: За рад са нискама (стринговима)
=====
/*
Следеће скаларне функције врше операције над улазним вредностима типа стринг (или integer),
а враћају нови стринг/character или нумеричку (целобројну) вредност.

    ASCII(char_or_string) - враћа вредност ASCII кода крајње левог карактера у
    'char_or_string';
        наведени 'char_or_string' може бити и израз који враћа вредност типа char или
        типа стринг.
    CHAR(int_val) - враћа знак који одговара вредности 'int_val'
    CHARINDEX() -
    CONCAT() -
    DIFFERENCE() -
    FORMAT() -
    LEFT() -
    LEN(char_or_string) - број карактера у 'char_or_string'
    LOWER() -
    LTRIM() -
    NCHAR() -
    PATINDEX() -
    QUOTENAME() -
    REPLACE() -
    REPLICATE() -
    REVERSE() -
    RIGHT() -
    RTRIM() -
    SOUNDEX() -
    SPACE() -
    STR() -
    STUFF() -
    SUBSTRING('string', od, dužina) - подстринг у стрингу 'string' од позиције 'od' до
    позиције 'od'+dužina'
        UNICODE() -
        UPPER() - претвара све знакове датог стринга у велике знакове
*/
-- ПРИМЕР. Приказати све особе које су рођене у месецу септембру.
-- Напомена: колону DatRod прво конвертујемо у низ знакова.
SELECT I#, Prezime, Ime
  FROM STUDENT
 WHERE SUBSTRING(CAST(DatRod AS char(10)), 6, 2) LIKE '09';
GO

-- ЗАДАТАК. Приказати све студенте који су рођени 1998. године.

-- ПРИМЕР. Приказати све особе које се презивају 'МАРИЋ'.
SELECT Ime, UPPER(Ime), Prezime, UPPER(Prezime) FROM STUDENT;

-- Нуспешно:
SELECT I#, Prezime, Ime
  FROM STUDENT
 WHERE UPPER(Prezime) LIKE 'Марић';
```

```
-- Успешно:
SELECT I#, Prezime, Ime
      FROM STUDENT
     WHERE UPPER(Prezime) LIKE 'МАРИЋ';
GO

-- ПРИМЕР. Приказати датум у различитим форматима.
DECLARE @d DATETIME = '10/31/2011';
SELECT FORMAT ( @d, 'd', 'en-US' ) as 'US English Result'
      ,FORMAT ( @d, 'd', 'en-GB' ) as 'Great Britain English Result'
      ,FORMAT ( @d, 'd', 'de-DE' ) as 'German Result'
      ,FORMAT ( @d, 'd', 'zh-CN' ) as 'Simplified Chinese (PRC) Result'
      ,FORMAT ( @d, 'd', 'sr-SR' ) as 'Serbian Result';

SELECT FORMAT ( @d, 'D', 'en-US' ) as 'US English Result'
      ,FORMAT ( @d, 'D', 'en-gb' ) as 'Great Britain English Result'
      ,FORMAT ( @d, 'D', 'de-de' ) as 'German Result'
      ,FORMAT ( @d, 'D', 'zh-cn' ) as 'Chinese (Simplified PRC) Result'
      ,FORMAT ( @d, 'D', 'sr-SR' ) as 'Serbian Result';
GO
/*
РЕЗУЛТАТ:

US English Result           Great Britain English Result   German Result
                  Chinese (Simplified PRC) Result       Serbian Result
Monday, October 31, 2011    31 October 2011
Oktober 2011  2011年10月31日
2011.                                     Montag, 31.
                                         ponedeljak, 31. oktobar
*/
-- ПРИМЕР. Приказати новчани износ у различитим форматима.
DECLARE @iznos MONEY = 1234.5678;
SELECT 'GERMAN' as CULTURA
      ,FORMAT(@iznos, 'N', 'de-de') as 'Numeric Format'
      ,FORMAT(@iznos, 'G', 'de-de') as 'General Format'
      ,FORMAT(@iznos, 'C', 'de-de') as 'Currency Format';
SELECT 'USA - AMERICAN' as CULTURA
      ,FORMAT(@iznos, 'N', 'en-US') as 'Numeric Format'
      ,FORMAT(@iznos, 'G', 'en-US') as 'General Format'
      ,FORMAT(@iznos, 'C', 'en-US') as 'Currency Format';
GO

-- ЗАДАТАК. Приказати новчани износ 1234.5678 у форматума N, G и C уобичајен у Србији.
-- ОПРЕЗ. Није подржано 'ср-СР' у овом случају.

-- Како у овом случају није подржано 'sr-SR', биће искоришћен немачки формат:
DECLARE @iznos MONEY = 1234.5678;
SELECT 'SRPSKA' as CULTURA
      ,FORMAT(@iznos, 'N', 'de-DE')+ ' RSD' as 'Currency Format'-- 'Numeric Format' as
      'Currency Format'
GO
=====
-- 2.1.3. ФУНКЦИЈЕ: За пребројавање (утврђивање кардиналности)
=====
/*
Функције за пребројавање су дефинисане над колонама било ког типа:
  - COUNT(*) - израчунава број вредности (редова) у резултату (result set-y);
    ово је једина агрегатна функција која узима у обзир и ознаку NULL
```

```

- COUNT(kolona) - израчунава број вредности у колони, занемарујући ознаке NOT NULL
- COUNT(DISTINCT kolona) - израчунава број различитих вредности у колони,
    занемарујући ознаке NOT NULL
*/
-- ПРИМЕР. Приказати број студената прве године.
SELECT Ime, GodStud
  FROM STUDENT
 WHERE GodStud = 1;

SELECT COUNT(*) as [Број бруцоша]
  FROM STUDENT
 WHERE GodStud = 1;

-- ПРИМЕР. Приказати број студената који имају возачку дозволу.
SELECT Ime, VozKat
  FROM STUDENT;

SELECT COUNT(VozKat) as [Број студената који имају возачку дозволу]
  FROM STUDENT;
GO

-- ЗАДАТАК. Колико има различитих презимена?
SELECT Prezime FROM STUDENT;
SELECT DISTINCT Prezime FROM STUDENT;

SELECT
    COUNT(DISTINCT Prezime) as [Број различитих презимена]
  FROM STUDENT;
GO

=====
-- 2.2. УПОТРЕБА GROUP BY КЛАУЗУЛЕ
=====
/*
    Клаузула GROUP BY групише све редове табеле које имају исту вредност.
    Партиционише скуп (табелу) у подскупове (групе) по задатој вредности.
*/
-- ПРИМЕР. Приказати минимални, средњи и максимални износ кредита студената 1., 2. и 3.
године,
-- као и број студената у наведеним годинама.
-- Решење је низ врло сличних упита, за сваку од тражених година...

SELECT MIN(Kredit), MAX(Kredit), AVG(Kredit) -- без именовања резултујућих колона
  FROM STUDENT
 WHERE GodStud = 1;

SELECT MIN(Kredit) [Min], MAX(Kredit) [Max], AVG(Kredit) [Avg], COUNT(*) [Broj studenata]
  FROM STUDENT
 WHERE GodStud = 1;

SELECT MIN(Kredit) [Min], MAX(Kredit) [Max], AVG(Kredit) [Avg], COUNT(*) [Broj studenata]
  FROM STUDENT
 WHERE GodStud = 2;

SELECT MIN(Kredit) [Min], MAX(Kredit) [Max], AVG(Kredit) [Avg], COUNT(*) [Broj studenata]
  FROM STUDENT
 WHERE GodStud = 3;

```

```
-- ... или један упит који користи GROUP BY клаузулу.  
SELECT MIN(Kredit) [Min], MAX(Kredit) [Max], AVG(Kredit) [Avg], COUNT(*) [Broj studenata]  
    FROM STUDENT  
    WHERE GodStud IN (1, 2, 3)  
    GROUP BY GodStud;  
GO  
  
-- ЗАДАТAK. Шта ради следећи упит?  
SELECT ESPB, Vrsta, COUNT(*) [Број предмета]  
    FROM PREDMET  
    GROUP BY ESPB, Vrsta;  
-- ОДГОВОР:  
SELECT Одговор  
      = 'Групише ПРЕДМЕТЕ по ЕСПБ, а затим по врсти, па их потом преbroјава.';  
GO  
  
-- ЗАДАТAK. Утврдити колико студената има исто средње име и које.  
SELECT  
    SredIme, COUNT(*) [Број студената]  
    FROM STUDENT  
    GROUP BY SredIme;  
  
    -- Зашто ово није добар упит?  
    SELECT  
        SredIme, COUNT(SredIme) [Број студената]  
        FROM STUDENT  
        GROUP BY SredIme;  
  
    SELECT Одговор  
      = 'Зато што ф-ја COUNT(SredIme) израчунава број вредности у  
колони, занемарујући ознаке NOT NULL';  
  
    -- Зашто ово није добар упит?  
    SELECT  
        SredIme, COUNT(DISTINCT SredIme) [Број студената]  
        FROM STUDENT  
        GROUP BY SredIme;  
  
    SELECT Одговор  
      = 'Зато што ф-ја COUNT(DISTINCT SredIme) израчунава број различитих вредности  
у колони, занемарујући ознаке NOT NULL.';  
GO  
  
-- ЗАДАТAK. Колико има студената и колики им је просечан износ кредита,  
-- разврстаних по смеровима.  
SELECT Smer#, COUNT(*) [Број студената], AVG(Kredit) [Просечан кредит]  
    FROM STUDENT  
    GROUP BY Smer#;  
GO
```

```
=====  
-- 2.3. УПОТРЕБА HAVING КЛАУЗУЛЕ  
=====  
/*  
Клаузула HAVING одређује критеријум селекције група пошто су групе већ  
формиране клаузулом GROUP BY. Клаузуле WHERE и GROUP BY могу се користити  
заједно, при чему WHERE клаузула увек иде пре GROUP BY клаузуле. Поступак  
извршења упита је следећи:
```

- 1) клаузулом `SELECT` се најпре изврши пројекција по колонама (тј. атрибутима релације),
 - 2) потом, клаузулом `WHERE` се изврши селекција редова у табели (тј. н-торки у релацији),
 - 3) затим се селектовани редови групишу `GROUP BY` клаузулом,
 - 4) на крају се изврши селекција формираних група `HAVING` клаузулом.
- */

```
SELECT Ime, Prezime
  FROM STUDENT
 ORDER BY 2;
```

-- ПРИМЕР. Приказати презимена студената која се јављају два или више пута.

```
SELECT Prezime, COUNT(*)
  FROM STUDENT
 GROUP BY Prezime
 HAVING COUNT(*) >= 2;
```

GO

-- ПРИМЕР. Приказати средњу висину студената груписаних по полу,
-- уколико је та висина већа од 161 цм. При рачунању изоставити
-- особе које се презивају 'Арсић'.

-- ПИТАЊЕ: Како се извршава овај упит?

-- ОДГОВОР:

--1) клаузулом `SELECT` се најпре изврши пројекција по колонама

```
SELECT Пол, Висина, Prezime
  FROM STUDENT;
```

--2) потом, клаузулом `WHERE` се изврши селекција редова у табели

```
SELECT Пол, Висина, Prezime
  FROM STUDENT
 WHERE Prezime NOT LIKE 'Арсић';
```

--3) затим се селектовани редови групишу `GROUP BY` клаузулом

```
SELECT Пол, AVG(Висина) [Посечна висина]
  FROM STUDENT
 WHERE Prezime NOT LIKE 'Арсић'
 GROUP BY Пол;
```

--4) на крају се изврши селекција формираних група `HAVING` клаузулом.

```
SELECT Пол, AVG(Висина) [Посечна висина]
  FROM STUDENT
 WHERE NOT (Prezime LIKE 'Арсић')
 GROUP BY Пол
 HAVING AVG(Висина) > 161;
```

GO

-- ЗАДАТAK. Приказати све смерове које похађа три или више студената.

```
SELECT Smer#
  FROM STUDENT
 ORDER BY Smer#;
```

```
SELECT Smer#, COUNT(*) [Број студената на смеру]
  FROM STUDENT
 GROUP BY Smer#
 HAVING COUNT(*) >= 3;
```

GO

```
--=====
-- 2.4. УПИТИ ИЗ КЛАУЗУЛЕ FROM
=====

/*
Где год стоји назив табеле, може да стоји и SELECT који враћа табелу.
Зато је могућ у клаузули FROM уместо табеле написати нови упит, који
ће да врати (привремену) табелу.
Упит из клаузуле FROM мора да има алијас.

-- ПРИМЕР. Приказати имена, презимена, годину студија и возачку дозволу свих
-- студената друге и треће године који имају возачку дозволу.
SELECT Ime, Prezime, GodStud, VozKat
    FROM (
        SELECT *
        FROM STUDENT
        WHERE VozKat IS NOT NULL) as [Возачи]
    WHERE GodStud IN (2, 3);

-- НАПОМЕНА: Како упит из клаузуле FROM мора да има алијас, у претходном упиту
-- тај алијас је [Возачи].
-- НАПОМЕНА: У претходном упиту направљена је привремена табела [Возачи] из које су
приказане
-- колоне (специфициране у SELECT листи) и редови (специфицирани у клаузули WHERE).
GO

-- ЗАДАТAK. Приказати називе, врсте и бодове предмета који имају мање од шест бодова
(ЕСПБ)
-- сортиран у опадајућем редоследу назива.

SELECT Naziv, Vrsta, ESPB
    FROM (
        SELECT *
        FROM PREDMET
        WHERE ESPB < 6
    ) [Predmeti]
    ORDER BY 1 DESC;
GO

--=====
-- 2.5. УПИТИ са КЛАУЗУЛОМ INTO (SELECT...INTO)
=====

/*
SELECT...INTO креира нову табелу у 'default filegroup'; сви редови добијени
извршењем упита смештају се у ту нову табелу.

-- ПРИМЕР. /SELECT...INTO + мало подупита/
-- Креирати табелу СМЕДЕРЕВЦИ у коју ће бити смештени подаци о
-- свим студентима које су рођене у Смедереву.

--      DROP TABLE IZBORNI;
SELECT *
    INTO IZBORNI
    FROM PREDMET
    WHERE Vrsta LIKE 'ИЗБ';

SELECT * FROM IZBORNI;
GO
```

```
=====
-- 2.6. ПРИВРЕМЕНЕ ТАБЕЛЕ
=====
/*
    Привремене (temporary) табеле су посебна врста табела, која се од основних (base)
    табела разликује због две своје особине:
        - сваку привремену табелу имплицитно укљања (тј. брише) СУБП,
        - свака привремена табела се смешта у системску БП tempdb.
    Привремене табеле могу бити локалне или глобалне. Локалне привремене табеле се
    укљањују на крају сваке текуће сесије; специфицирају се префиксом # (на пример:
    #priv_tab). Глобалне привремене табеле се специфицирају префиксом ## и укљањују
    се на крају сесије у којој је креирана табела.
*/
-- ПРИМЕР. Креирати привремену привремену табелу...
CREATE TABLE #Privremena_1 (
    Ime nvarchar(33) not null,
    Prezime nvarchar(33) not null,
    Visina int null
);
GO
SELECT * FROM #Privremena_1;
GO

-- ПРИМЕР. Креирати привремену привремену табелу...
SELECT Ime, Prezime, DatRod
    INTO #Privremena_2
    FROM STUDENT;
GO
SELECT * FROM #Privremena_2;
GO

=====
-- 2.7. CASE ИЗРАЗИ
=====
/*
Приликом програмирања апликација заснованих на базама података понекад је неопходно
модификовати начин на који се подаци приказују. На пример, уколико је у бази података
пол студената кодиран вредностима М, Ж и Н (за: мушки, женски и непознат) или статус
производа вредностима 1 и 2 (за: у_понуди и ван_понуде), CASE израз може олакшати
начин њиховог приказа. Сам CASE израз је сложен (в. синтаксу ниже) и састоји се из
више једноставнијих израза :
    - улазног израза (input expression),
    - када-израза (when expression),
    - резултујућег израза (result expression),
    - резултујућег иначе-израза (else result expression) и
    - логичког израза (Boolean expression).

Постоје два облика CASE израза:
    1) Једноставан CASE израз (simple CASE expression) упоређује вредност улазног
       израза (input expression) са скупом вредности једноставних када-израза (when
       expression)
        да би одредио вредност резултујућег израза (result expression).
    2) Сложени CASE израз (searched CASE expression) упоређује вредности логичких
       израза
        (Boolean expressions) да би одредио вредност резултујућег израза (result
       expression).

```

Синтакса:

1) Simple CASE expression:

```
CASE input_expression
    WHEN when_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

2) Searched CASE expression:

```
CASE
    WHEN Boolean_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

Оба облика CASE израза подржавају опционо ELSE део.

CASE се може користити у било ком исказу (наредби) или клаузули која дозвољава валидне изразе. На пример, CASE се може користити у наредбама као што су SELECT, UPDATE, DELETE и SET, а и у клузулама као што су SELECT листа, IN, WHERE, ORDER BY и HAVING.

*/

```
-- ПРИМЕР. /Једноставан CASE израз/
-- Приказати имена, презимена, шифре смерова, називе смерова и датуме рођења свих
студената,
-- ако је познато следеће:
```

```
--          Смер#      Назив
-- ======  ======
--      10      Софтверско инжењерство
--      20      Информациони системи
--      30      Информационе технологије
--      ...     Остали смерови
--
```

```
SELECT
    Ime, Prezime, Smer#, Smer =
    CASE Smer#
        WHEN 10 THEN 'Софтверско '+'инжењерство' -- може израз
        WHEN 20 THEN 'Информациони системи'      -- а може и само вредност
        WHEN 30 THEN 'Информационе технологије'
        ELSE 'Један од осталих смерова'
    END,
    DatRod
FROM STUDENT
ORDER BY Smer;
```

-- Уколико желимо да прикажемо бруцаше (за њих важи: Smer# IS NULL):

```
SELECT
    Ime, Prezime, Smer#, Smer =
    CASE ISNULL(Smer#, 0)
        WHEN 10 THEN 'Софтверско инжењерство'
        WHEN 20 THEN 'Информациони системи'
        WHEN 30 THEN 'Информационе технологије'
        WHEN 0 THEN 'Бруцаш'
        ELSE 'Један од осталих смерова'
    END,
    DatRod
FROM STUDENT
```

```

ORDER BY Smer;
GO

-- ЗАДАТАК. Шта ради следећи упит?
SELECT
    P#, --Naziv,
    VrstaIspita =
        CASE Vrsta
            WHEN 'ОБВ' THEN Naziv + ' (обавезан испит)'
            WHEN 'ИЗБ' THEN Naziv + ' (изборан испит)'
            ELSE 'Ово никада не би требало да се изврши.'
        END
FROM PREDMET;
GO

-- ПРИМЕР. /Сложени CASE израз/
-- Приказати имена и презимена студената као и њихова 'примања' (кредите и стипендије).
-- Уз сваког студента приказати и одговарајући опис:
-- - ако је износ кредита и стипендије мањи или једнак 100 н.ј.: мали износ,
-- - ако је износ кредита и стипендије већи од 100 н.ј. и мањи од 200 н.ј.: средњи износ,
-- - ако је износ кредита и стипендије већи од 200 н.ј.: велики износ.

SELECT * FROM STUDENT_FINANS;

SELECT
    I#, Ime, Prezime,
    Kredit,
    Stipendija = ISNULL(Stip, 0),
    Ukupno = Kredit + ISNULL(Stip, 0),
    Opis = CASE
        WHEN (Kredit + ISNULL(Stip, 0)) <= 100 THEN '-- Мали износ'
        WHEN (Kredit + ISNULL(Stip, 0)) > 100 AND (Kredit + ISNULL(Stip, 0)) <= 200)
        THEN 'Средњи износ'
        WHEN (Kredit + ISNULL(Stip, 0)) >= 200 THEN '++ Велики износ'
    END
    FROM STUDENT;

-- Употреба погледа. Који је упит једноставнији: претходни или следећи?
SELECT *,
    Опис = CASE
        WHEN Укупно <= 100 THEN '-- Мали износ'
        WHEN Укупно > 100 AND Укупно <= 200 THEN 'Средњи износ'
        WHEN Укупно >= 200 THEN '++ Велики износ'
    END
    FROM STUDENT_FINANS;
-- Ако упит неће да се изврши у овом облику пробати: [Укупно]
GO
-----
```

-- 3. УПИТИ НАД ДВЕ ИЛИ ВИШЕ ТАБЕЛА

```
USE fakultet;
GO
```

-- 3.1. ПОДУПИТИ

```
/*
```

Сви предходни упити садрже различита поређења вредности колоне(а) са неким изразом или константом. Поред тога, језик T-SQL располаже и могућностима да упит садржи подупити (subqueries). Подупит је упит који се изврши унутрашњег упита, па се сматрају за први начин повезивања вредности различитих табела. Подупити могу бити унутрашњи и спољашњи упити (inner query).

Подупити се могу користити за постављање упита над две и више табела, па се сматрају за први начин повезивања вредности различитих табела.

Подупити могу бити унутрашњи и у INSERT, UPDATE или DELETE наредбама.

Постоје два типа подупита:

- једноставни и
- корелисани.

У једноставном подупиту унутрашњи упит се извршава тачно једном, динамичком заменом резултата унутрашњег упита у клаузули WHERE спољашњег упита. То значи да се увек прво извршава унутрашњи упит, а затим спољашњи упит.

Извршавање корелисаног подупита се разликује од извршавања једноставног подупита

по томе што се унутрашњи упит извршава сваки пут када се преузме нови ред спољашњег упита. То значи да се увек прво извршава спољашњи упит, а затим се извршава унутрашњи упит онолико пута колико има редова у резултату (додијелом извршењем спољашњег упита). */

-- 3.1.1. ЈЕДНОСТАВНИ ПОДУПИТИ

```
/*
```

Једноставни упити могу да буду засновани на употреби:

- оператора поређења (релационих оператора):
 - # једнакост (=),
 - # неједнакост (!= или <>) и
 - # остали (<, >, <=, >=),
- оператора IN,
- оператора ANY или ALL.

```
*/
```

-- 3.1.1.1. ЈЕДНОСТАВНИ ПОДУПИТИ ЗАСНОВАНИ НА УПОТРЕБИ ОПЕРАТОРА ПОРЕЂЕЊА

-- ПРИМЕР. /Подупити засновани на употреби оператора (не)једнакости/
-- Приказати имена и презимена свих студената који су уписаны на
-- смеру 'Софтверско инжењерство'.

-- 1. корак:

```
SELECT S#, Naziv
    FROM SMER
    WHERE Naziv LIKE 'Софтверско инжењерство';
```

-- 2. корак:

```
SELECT S#, Ime, Prezime, Smer#
    FROM STUDENT
    WHERE Smer# = 10;
```

-- Или, у једном кораку:

```
SELECT S#, Ime, Prezime, Smer#
    FROM STUDENT
    WHERE Smer# =
        (
            SELECT S#
            FROM SMER
            WHERE Naziv LIKE 'Софтверско инжењерство');
```

GO

-- ПРИМЕР. Приказати имена и презимена свих студената који нису уписаны на
-- смер 'Софтверско инжењерство'.

-- Следећи упит изгледа тачно, међутим он пријављује грешку. Зашто?

```
SELECT S#, Ime, Prezime, Smer#
    FROM STUDENT
    WHERE Smer# =
        (
            SELECT S#
            FROM SMER
            WHERE Naziv NOT LIKE 'Софтверско инжењерство');
```

/*

ОДГОВОР:

```
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value.
This is not permitted when the subquery follows =, !=, <, <= , >, >=
or when the subquery is used as an expression.
```

*/

-- Зато мора овако:

```
SELECT S#, Ime, Prezime, Smer#
    FROM STUDENT
    WHERE Smer# != (
        SELECT S#
        FROM SMER
        WHERE Naziv LIKE 'Софтверско инжењерство');
```

GO

-- ПРИМЕР. Приказати све смерове који припадају одсеку 'Менаџмент'.

-- В.1.

```
SELECT S#, Naziv
    FROM SMER
    WHERE Odsek# =
        (
            SELECT O#
            FROM ODSEK
            WHERE Naziv LIKE 'Менаџмент');
```

-- B.2.

```
SELECT S#, Naziv, Odsek = 'М е н а Ѿ м е н т'  
FROM SMER  
WHERE Odsek# = (  
    SELECT O#  
    FROM ODSEK  
    WHERE Naziv LIKE 'Менаџмент');  
GO
```

-- ПРИМЕР. /Подупити засновани на употреби релационих оператора: <, >, <=, >= /

-- Приказати имена, презимена и кредите свих студената које имају кредите
-- мање или једнаке кредиту који има студент са бројем индекса 16014.

```
SELECT S#, Ime, Prezime, Kredit  
    FROM STUDENT  
    WHERE Kredit <= (  
        SELECT Kredit  
        FROM STUDENT  
        WHERE S# = 16014);
```

-- НАПОМЕНА: Подупити овог типа морају да враћају тачно једну вредност, као и
-- подупити засновани на употреби (не)једнакости.

GO

-- 3.1.1.2. ЈЕДНОСТАВНИ ПОДУПИТИ ЗАСНОВАНИ НА УПОТРЕБИ ОПЕРАТОРА IN

-- ПРИМЕР. /Подупити засновани на употреби оператора IN/
-- Приказати студенте који уписани на смеру 'Управљање пословањем' или 'Управљање производњом'.

```
SELECT S#, Ime, Prezime, Smer#  
    FROM STUDENT  
    WHERE Smer# IN (  
        SELECT S#  
        FROM SMER  
        WHERE Naziv IN ('Управљање пословањем', 'Управљање производњом'));  
GO
```

-- ПРИМЕР. Приказати све студенте који су уписани на одсеку 'Информатика'.

```
SELECT Ime, Prezime, Smer#  
    FROM STUDENT  
    WHERE Smer# IN (  
        SELECT S#  
        FROM SMER  
        WHERE Odsek# = (  
            SELECT O#  
            FROM ODSEK  
            WHERE Naziv LIKE 'Информатика'));
```

GO

-- ПРИМЕР. Приказати називе смерова које су уписали студенти који имају највише кредите.

```
SELECT Naziv  
    FROM SMER  
    WHERE S# IN (  
        SELECT Smer#  
        FROM STUDENT
```

```

WHERE Kredit = (
    SELECT MAX(Kredit)
    FROM STUDENT));
GO

=====
-- 3.1.1.3. ЈЕДНОСТАВНИ ПОДУПИТИ ЗАСНОВАНИ НА УПОТРЕБИ ОПЕРАТОРА ANY И ALL
=====
/*
    Оператори ANY и ALL се увек користе заједно са неким од оператора поређења
    (=, !=, <, >, <=, >=).
    Оператор ANY враћа тачну вредност (true) уколико се у резултату извршавања
    унутрашњег упита налази бар једна вредност која задовољава услов поређења. Реч
    SOME је синоним за ANY.
    Оператор ALL враћа тачну вредност (true) уколико се у резултату извршавања
    унутрашњег упита налазе све вредности која задовољавају услов поређења.
*/
-- ПРИМЕР. /Подупити засновани на употреби оператора ANY/
-- Приказати имена, презимена и кредите свих студената, сем оних који имају
-- најмањи кредите,
-- уређен по кредиту.

-- Сви студенти
SELECT Ime, Prezime, Kredit
    FROM STUDENT
        ORDER BY Kredit;

-- Тражени упит:
SELECT Ime, Prezime, Kredit
    FROM STUDENT
    WHERE Kredit > ANY (
        SELECT DISTINCT Kredit
            FROM STUDENT)
        ORDER BY Kredit;
GO

-- ПРИМЕР. Приказати имена, презимена и кредите свих студената, сем оних који
-- имају највеће кредите, уређен по кредиту.
SELECT Ime, Prezime, Kredit
    FROM STUDENT
    WHERE Kredit < ANY (
        SELECT DISTINCT Kredit
            FROM STUDENT)
        ORDER BY Kredit;
GO

-- ПРИМЕР. /Подупити засновани на употреби оператора ALL/
-- Приказати имена, презимена и кредите студената који
-- имају највеће кредите.
SELECT Ime, Prezime, Kredit
    FROM STUDENT
    WHERE Kredit >= ALL (
        SELECT DISTINCT Kredit
            FROM STUDENT)
        ORDER BY Kredit;
GO

```

```
-- ПРИМЕР. Приказати имена, презимена и кредите студената који
-- имају најмање кредите.
SELECT Ime, Prezime, Kredit
    FROM STUDENT
    WHERE Kredit <= ALL (
        SELECT DISTINCT Kredit
            FROM STUDENT)
    ORDER BY Kredit;
GO

-- НАПОМЕНА: Немојте користити операторе ANY или ALL. Сваки упит у којем се
-- користе ови оператори може се боље (читајте: лакше) формулисати помоћу
-- функције EXISTS.
```

```
=====
-- 3.1.2. КОРЕЛИСАНИ ПОДУПИТИ
=====
/*
```

Други начин образовања подупита су корелисани подупити. За неки подупит се каже да је корелисан уколико унутрашњи упит зависи од спољног упита за сваку од његових вредности.

```
*/
```

```
-- ПРИМЕР. /Разлика између упита и корелисаног подупита/
-- Приказати све студенте које су уписале смер који има Smer# = 10.
```

```
-- Упит:
SELECT Ime, Prezime, Smer#
    FROM STUDENT
    WHERE Smer# = 10;
```

```
-- Корелисани подупит:
SELECT Ime, Prezime, Smer#
    FROM STUDENT AS ST
    WHERE 10 = (
        SELECT S#
            FROM SMER AS SM
            WHERE SM.S# = ST.Smer#
    );
GO
```

```
-- ЗАДАТAK. Приказати све студенте који су уписали смер 'Управљање производњом'.
```

```
-- Подупит:
SELECT Ime, Prezime, Smer#
    FROM STUDENT
    WHERE Smer# = (
        SELECT S#
            FROM SMER
            WHERE Naziv LIKE 'Управљање производњом');
GO
```

```
-- Корелисани подупит:
SELECT Ime, Prezime, Smer#
    FROM STUDENT AS ST
    WHERE 'Управљање производњом' = (
        SELECT Naziv
```

```

        FROM SMER AS SM
        WHERE SM.S# = ST.Smer#
    );
GO

=====
-- 3.1.2.1. КОРЕЛИСАНИ ПОДУПИТИ ЗАСНОВАНИ НА УПОТРЕБИ функције EXISTS
=====
/*
За функцију EXISTS подупит представља аргумент, при чему се тачна вредност (true) добија уколико подупит врати један или више редова, а нетачна вредност (false) се добија уколико подупит не врати ниједан ред (тј. ако врати нула редова). Подупит ф-је EXISTS скоро у свим случајевима зависи од неке променљиве спољног упита. Због тога је подупит ф-је EXISTS најчеће корелисани подупит.
*/
-- ПРИМЕР. /Подупити засновани на употреби функције EXISTS/
-- Приказати имена и презимена свих студената који су уписали смер 'Софтверско инжењерство'.
SELECTS#, Ime, Prezime, Smer#
    FROM STUDENT ST
    WHERE EXISTS (
        SELECT *
        FROM SMER SM
        WHERE ST.Smer# = SM.S#
        AND Naziv LIKE 'Софтверско инжењерство');
/*
Упоредите овај корелисани подупит са једноставним подупитом у 3.1.1.1:
SELECT S#, Ime, Prezime, Smer#
    FROM STUDENT
    WHERE Smer# = (
        SELECT S#
        FROM SMER
        WHERE Naziv LIKE 'Софтверско инжењерство');
*/
GO
/*
ПИТАЊЕ: Како се извршава претходни упит са корелисаним подупитом?
ОДГОВОР: Прво, спољни упит обрађује први ред из табеле STUDENT. Затим се извршава подупит ф-је EXISTS, како би се одредило да ли у табели SMER постоји неки ред у којем је назив смера 'Софтверско инжењерство' и да је S# тог смера једнака Smer# студента из спољног упита. Ако се подупит евалуира у TRUE (тј. врати се бар један ред), онда се та особа приказује, иначе се прелази на следећи ред из табеле STUDENT.
*/
-- ЗАДАТАК. Приказати имена, презимена, S# места рођења и S# места живљења свих -- особа које живе у месту у којем су и рођене.

```

```

SELECT Ime, Prezime, IdMestaRod, Smer#
    FROM STUDENT O1
    WHERE EXISTS (

```

```
SELECT *
FROM STUDENT_02
WHERE 01.ИдМестаРод = 02.Smer#
      AND 01.S# = 02.S#);      -- ВРЛО ВАЖНО: да би се радило о истој
особи
GO
```

-- ЗАДАТAK. Приказати називе смерова које није уписао ни један студент.

```
SELECT S#, Naziv
FROM SMER SM
WHERE NOT EXISTS (
      SELECT *
      FROM STUDENT ST
      WHERE ST.Smer# = SM.S#);
GO
```

```
=====
-- 3.2. ОПЕРАТОРИ СПАЈАЊА
=====
```

/*
Спајање (енгл. join) је начин обједињавања две (или више) табеле(а) у једну (резултујућу).
Спајањем две табеле, првој табели се приодају колоне друге табеле (повећава се степен табеле).

Две (или више) табела се могу спојити на више различитих начина:

- безусловно спајање (Декартов производ, Картизијански производ; CROSS JOIN)
- унутрашње спајање (INNER JOIN или само JOIN), које може бити:
 - # спајање на основу једнакости (спајање оператором једнакости, екви-џоин; EQUI-JOIN),
 - # спајање на основу једнакости без понављања (природно спајање; NATURAL JOIN),
- спајање на основу неједнакости (спајање оператором поређења; THETA JOIN),
- спољно спајање (спајање мимо услова; OUTERJOIN), које може бити:
 - # лево (LEFT OUTER JOIN),
 - # десно (RIGHT OUTER JOIN) и
 - # централно (FULL OUTER JOIN),
- самоспајање (SELF JOIN).

*/

```
-- Безусловно спајање (Декартов производ, Картизијански производ; CROSS JOIN)
SELECT COUNT(*) as BrojSmerova FROM SMER ;
SELECT COUNT(*) as BrojStudenata FROM STUDENT;
SELECT Ukupno = (SELECT COUNT(*) FROM STUDENT) * (SELECT COUNT(*) FROM SMER);
GO
```

-- ПРИМЕР. /Декартов производ/

```
SELECT Ime, Prezime, Naziv, S#
      FROM STUDENT CROSS JOIN SMER;
```

-- Стара синтакса:

```
SELECT Ime, Prezime, Naziv, S#
      FROM STUDENT, SMER;
GO
```

```
=====
-- Унутрашње спајање (INNER JOIN или само JOIN), које може бити:
```

--# спајање на основу једнакости (спајање оператором једнакости, екви-џоин; EQUI-JOIN)

```
SELECT * FROM STUDENT;
SELECT * FROM SMER;

SELECT STUDENT.* , SMER.*
  FROM STUDENT
 JOIN SMER ON STUDENT.Smer# = SMER.S#;

-- Стара синтакса:
SELECT STUDENT.* , SMER.*
  FROM STUDENT, SMER
 WHERE STUDENT.Smer# = SMER.S#;

-- Или:
SELECT T.* , M.*
  FROM STUDENT T, SMER M
 WHERE T.Smer# = M.S#;
GO
```

--# спајање на основу једнакости без понављања (природно спајање; NATURAL JOIN)

```
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
 INNER JOIN SMER M ON S.Smer# = M.S#;

SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
 JOIN SMER M ON S.Smer# = M.S#;

-- исто је следеће и претходно:
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM SMER M
 JOIN STUDENT S ON S.Smer# = M.S#;
```

-- спајање на основу неједнакости (спајање оператором поређења; THETA JOIN)

```
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM SMER M
 JOIN STUDENT S ON S.Smer# < M.S#;
```

-- спољно спајање (спајање мимо услова; спајање без губитка информација; OUTER JOIN),
-- које може бити:

--# лево (LEFT OUTER JOIN),
-- Приказ свих студената који су уписали смер. Број редова = 8.

```
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
```

```
JOIN SMER M ON S.Smer# = M.S#
ORDER BY M.S#;

-- А где су студенти који још увек нису уписали смер? Где су бруцоши?
-- Неповезане редове из леве табеле (овде: STUDENT) добијамо левим спољним
спајањем:
```

```
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
    LEFT OUTER JOIN SMER M ON S.Smer# = M.S#
ORDER BY M.S#;
```

-- Напомена: Број редова = 11, јер су додата три студента.

-- Или (без OUTER):

```
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
    LEFT JOIN SMER M ON S.Smer# = M.S#
ORDER BY M.S#;
GO
```

--# десно (RIGHT OUTER JOIN)

```
-- Приказ свих студената који су уписали смер. Број редова = 8.
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
    JOIN SMER M ON S.Smer# = M.S#
ORDER BY M.S#;
```

-- А где су смерови које још увек нису уписали студенти?

-- Неповезане редове из десне табеле (овде: SMER) добијамо десним спољним
спајањем:

```
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
    RIGHT OUTER JOIN SMER M ON S.Smer# = M.S#
ORDER BY S.I#;
```

-- Напомена: Број редова = 10, јер су додата два смера.

-- Или (без OUTER):

```
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
    RIGHT JOIN SMER M ON S.Smer# = M.S#
ORDER BY S.I#;
GO
```

--# централно (FULL OUTER JOIN),

```
-- Приказ свих студената који су уписали смер. Број редова = 8.
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
    JOIN SMER M ON S.Smer# = M.S#;
```

-- А где су студенти који још увек нису уписали смер?
-- А где су смерови које још увек нису уписали студенти?
-- Када желимо да прикажемо и једно и друго, онда користимо
-- централно спољно спајање:

```
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
    FULL OUTER JOIN SMER M ON S.Smer# = M.S#;
```

-- Напомена: Број редова = 8 + 3 + 2 = 13, јер су на осам студената (који су уписани
-- на неки од смерова) додата три неуписана студента и два смера без уписаних студената.

-- или (без OUTER):

```
SELECT S.Ime, S.Prezime, S.DatRod, M.S#, M.Naziv
  FROM STUDENT S
    FULL JOIN SMER M ON S.Smer# = M.S#;
```

-- Самоспајање (SELF JOIN)

-- Има смисла за атореферентне табеле. Пример: организациона_целина има за надређену (deo je)
-- неке друге организационе_целине, а она је део неке друге итд.

```
IF OBJECT_ID('ORG_CEL', 'U') IS NOT NULL
  DROP TABLE ORG_CEL;
GO
CREATE TABLE ORG_CEL (      -- организациона целина
  OC#          int not null primary key,
  Naziv        nvarchar(55) not null,
  Adresa       nvarchar(99) null,
  NadOrgCel   int null foreign key references ORG_CEL(OC#)
);
GO
INSERT INTO ORG_CEL VALUES
(5, N'Preduzeće XYZ', N'Bulevar rada 123', NULL), -- почетак хијерархије
(52, N'Proizvodnja', N'Industrijska zona b.b.', 5),
(521, N'Fabrika AA', N'Industrijska zona b.b.', 52),
(5211, N'Pogon AA-1', NULL, 521),
(5212, N'Pogon AA-2', NULL, 521),
(52121, N'Odeljenje AA-2-X', NULL, 5212),
(52122, N'Odeljenje AA-2-Y', NULL, 5212),
(52123, N'Odeljenje AA-2-Z', NULL, 5212),
(5213, N'Pogon AA-3', NULL, 521),
(522, N'Fabrika BB', N'Industrijska zona b.b.', 52),
(53, N'Razvoj', N'Tehnološko-razvojni park 77', 5),
(54, N'Komercijala', N'Zeleni venac 8', 5),
(541, N'Prodaja', N'Zeleni venac 8/3', 54),
(5411, N'Domaća prodaja', NULL, 541),
(5412, N'Inostrana prodaja', NULL, 541),
(542, N'Nabavka', N'Zeleni venac 8/4', 54),
(51, N'Uprava', N'Menadžerski trg 1/99', 5);
GO
-- Све организационе целине
SELECT * FROM ORG_CEL;
```

-- ПРИМЕР. Све организационе целине са шифрама и називима надређених орг. целина
SELECT POD.*, NAD.OC# as [Šifra nadređene OC], NAD.Naziv as [Naziv nadređene OC]
FROM ORG_CEL NAD
JOIN ORG_CEL POD ON POD.NadOrgCel = NAD.OC#;

-- ПРИМЕР. А где је врх хијерархије? Како да га прикажемо?
SELECT POD.OC#, POD.Naziv, POD.NadOrgCel, NAD.OC# as [Šifra nadređene OC],
NAD.Naziv as [Naziv nadređene OC]
FROM ORG_CEL NAD
RIGHT JOIN ORG_CEL POD ON POD.NadOrgCel = NAD.OC#;

-- ПРИМЕР. Која организациона целина је надређена ОЦ 'Производња'?
SELECT POD.OC#, POD.Naziv, NAD.OC# as [Šifra nadređene OC], NAD.Naziv as [Naziv
nadređene OC]
FROM ORG_CEL NAD
JOIN ORG_CEL POD ON POD.NadOrgCel = NAD.OC#
WHERE POD.Naziv LIKE 'Proizvodnja';

-- ПРИМЕР. Које организационе целине су подређене ОЦ 'Производња'?
SELECT NAD.OC#, NAD.Naziv, POD.OC# as [Šifra podređene OC], POD.Naziv as [Naziv
Podređene OC]
FROM ORG_CEL POD
JOIN ORG_CEL NAD ON POD.NadOrgCel = NAD.OC#
WHERE NAD.Naziv LIKE 'Proizvodnja';

-- ПИТАЊЕ: Како приказати целокупну организациону структуру?
-- ОДГОВОР: Обични упити, путем спајања, могу да прикажу само први следећи ниво
изнад и
-- први следећи ниво испод, али не могу да прикажу целокупну хијерархију. За то се
-- морају користити рекурзивни упити.

=====
-- 3.3. РЕКУРЗИВНИ УПИТИ
=====

-- За реализацију рекурзивних упита користи се:
-- CTE (common table expression) - привремени именовани резултат (резултујући
скуп).

-- ПРИМЕР. Приказати целокупну организациону структуру за: 'Preduzeće XYZ'
-- то јест, од почетка хијерархије, па до њеног kraja.

```
WITH OrgStruktura(SifraOC, NivoOCuOrgStrukturi, NazivOC, SifraNadredjeneOC/*  
ManagerID, EmployeeID, Title, EmployeeLevel*/) AS  
(  
    SELECT OC#, NivoOCuOrgStrukturi = 0, Naziv, NadOrgCel --ManagerID, EmployeeID,  
    Title, 0 AS EmployeeLevel  
    FROM ORG_CEL  
    WHERE NadOrgCel IS NULL  
          --<<< почетак хијерархије; корен стабла  
    UNION ALL  
    SELECT oc.OC#, NivoOCuOrgStrukturi + 1, oc.Naziv, oc.NadOrgCel  
    FROM ORG_CEL AS oc  
    INNER JOIN OrgStruktura AS s ON oc.NadOrgCel = s.SifraOC  
)  
SELECT SifraOC, NivoOCuOrgStrukturi, NazivOC, SifraNadredjeneOC  
FROM OrgStruktura  
ORDER BY SifraNadredjeneOC;  
GO
```

```

-- ПРИМЕР. Приказати организациону структуру за: 'Fabrika AA'
-- то јест, део хијерархије, који почиње са 'Fabrika AA', па до њеног краја.

WITH OrgStruktura(SifraOC, NivoOCuOrgStrukturi, NazivOC, SifraNadredjeneOC/*  

ManagerID, EmployeeID, Title, EmployeeLevel*/) AS  

(  

    SELECT OC#, NivoOCuOrgStrukturi = 0, Naziv, NadOrgCel --ManagerID, EmployeeID,  

Title, 0 AS EmployeeLevel  

    FROM ORG_CEL  

    WHERE Naziv LIKE 'Fabrika AA' --<<< почетак поднивоа хијерархије; грана  

стабла  

    UNION ALL  

    SELECT oc.OC#, NivoOCuOrgStrukturi + 1, oc.Naziv, oc.NadOrgCel  

    FROM ORG_CEL AS oc  

        INNER JOIN OrgStruktura AS s ON oc.NadOrgCel = s.SifraOC  

)
SELECT SifraOC, NivoOCuOrgStrukturi, NazivOC, SifraNadredjeneOC  

    FROM OrgStruktura  

    ORDER BY SifraNadredjeneOC;
GO

=====
-- 3.4. СКУПОВНИ ОПЕРАТОРИ
=====

/*
    UNION – унија два скупа
    EXCEPT – разлика два скупа
    INTERSECT – пресек два скупа
*/
SET NOCOUNT ON;

CREATE TABLE #AAA (
    broj tinyint, txt nchar(5));
GO
CREATE TABLE #BBB (
    broj tinyint, txt nchar(5));
GO
INSERT INTO #AAA VALUES
    (11, 'aaaaa'),
    (22, 'bbbbbb'),
    (33, 'cccccc'),
    (44, 'ddddd');
GO
INSERT INTO #BBB VALUES
    (33, 'ccccc'),
    (44, 'ddddd'),
    (55, 'eeeeee'),
    (66, 'fffff');
GO

SELECT * FROM #AAA;
SELECT * FROM #BBB;
GO

PRINT 'A UNION B';

SELECT * FROM #AAA
UNION
SELECT * FROM #BBB;

```

```
GO
PRINT 'A EXCEPT B';

SELECT * FROM #AAA
EXCEPT
SELECT * FROM #BBB;
GO
PRINT 'A INTERSECT B';

SELECT * FROM #AAA
INTERSECT
SELECT * FROM #BBB;
GO
```