5. Језици и опис конструкција језика Јава

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

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

Детаљан опис програмског језика Јава би захтевао засебну књигу сличног обима као што је ова[8]. Стога ће у овом поглављу бити дат релативно сажет и непотпун преглед неких значајнијих конструкција језика.

5.1. Граматика, синтакса и семантика

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

- **Граматика** је наука о језику и његовим законима. Она дефинише скуп правила којима се описују све исправне конструкције, прихватљиве у језику.

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

- **Синтакса** је наука о језику у којој се изучава формирање граматички исправних конструкција језика. Синтаксу језика чини скуп правила за формирање конструкција језика. Уколико је програм написан без поштовања правила о запису конструкција програмског језика, јављају се синтаксне грешке. По правилу, ова врста грешака не изазива велике тешкоће јер може да их открије и програм-преводилац.

- **Семантика** је наука о језику у којој се изучава значење конструкција језика. Семантику језика чини скуп правила за утврђивање значења конструкција језика. За разлику од синтаксних грешака, већину семантичких грешака не може да открије преводилац, већ то мора да уради човек. Овакве грешке се, у принципу, теже откривају.

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

- **Објект-језик** је језик који се изучава.

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

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

- Бекусова нотација и

- синтаксни дијаграми.

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

Семантика програмских језика се, обично, формално описује помоћу:

- Бечког дефиниционог језика или

- Ван Вајнгаренових граматика.

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

5.1.1. Бекусова нотација

Синтакса програмског језика се описује помоћу коначног скупа металингвистичких формула (МЛФ). МЛФ се састоји из леве и десне стране раздвојене универзалним метасимболом **'::='**.

Дакле, МЛФ је облика:

```
α ::= γ
```

где је:

- α металингвистичка променљива;
- γ металингвистички израз.

Металингвистичка променљива је фраза природног језика ограђена стреличастим заградама '<, >'.

Металингвистичка константа је знак или кључна реч објект-језика.

Универзални метасимбол **'::='** чита се „по дефиницији је“.

Универзални метасимбол **'|'** чита се „или“.

Металингвистички израз може бити:

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

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

Цео декадни број се у програмским језицима записује на сличан начин као и у математици, уз изузетак што је дозвољена употреба подвлаке **'_'** у оквиру записа броја. Примери исправно записаних целих декадних бројева су:

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

Подвлака записана у оквиру броја не утиче на његову вредност, на пример, број -1__00__00 је исто што и -10000.

Употреба подвлаке на крајевима записа није дозвољена. Записи који следе нису коректно записани цели декадни бројеви:

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

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

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

Модификација изворне Бекусове нотације постиже се увођењем нових метасимбола:

1. лева и десна велика заграда **'{**, **}'** – означавају понављање обухваћене конструкције нула, једном или више пута;

2. лева и десна средња заграда **'[**, **]'** – означавају опционо понављање обухваћене конструкције, дакле нула или једном, и

3. лева и десна мала заграда **'(**, **)'** – означавају груписање конструкција.

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

```
<не нула декадна цифра> ::= 1|2|3|4|5|6|7|8|9
<декадна цифра> ::= 0|<не нула декадна цифра>
<декадна цифра или подвлака> ::= <декадна цифра>|_
```

```
<декадни цео број без знака> ::= <не нула декадна цифра>
    |<не нула декадна цифра>{<декадна цифра или подвлака>}<декадна цифра>
<знак броја> ::= +|-
<декадни цео број> ::= [<знак броја>]0
    |[<знак броја>]<декадни цео број без знака>
```

5.2. Елементарне конструкције језика Јава

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

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

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

Ове елементарне конструкције су детаљно објашњене у даљем тексту.

5.2.1. Идентификатори

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

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

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

Следи списак неколико примера идентификатора у Јави:

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

Следеће речи не представљају идентификаторе у Јави:

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

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

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

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

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

5.2.2. Кључне речи

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

Кључне речи се могу дефинисати следећом конструкцијом:

```
<кључна реч> ::= abstract|assert|boolean|break|byte|case|catch|char|class
    |const|continue|default|do|double|else|enum|extends|final|finally
    |float|for|goto|if|implements|import|instanceof|int|interface|long
    |native|new|package|private|protected|public|return|short|static
```

```
    |strictfp|super|synchronized|switch|this|throw|throws|transient|try
    |void|volatile|while
```

Напомене:

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

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

5.2.3. Литерали

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

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

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

Целобројни литерали

Цели бројеви у Јави могу бити записани као декадни, октални или хексадекадни (а од верзије 7 Јаве и као бинарни). Почев од верзије 7, Јава допушта да целобројни литерал садржи и подвлаку. Дефиниције означених целобројних литерала у овим различитим основама су аналогне дефиницији означеног декадног целобројног литерала уз разлику да бинарни број започиње префиксом `0b` или `0B`, октални префиском `0`, хексадекадни префиксом `0x` или `0X` и притом се користи одговарајући редуковани/проширени скуп цифара:

```
<бинарна цифра> ::= 0|1
<октална цифра> ::= 0|1|2|3|4|5|6|7
<хексадекадна цифра> ::= <декадна цифра>|a|b|c|d|e|f|A|B|C|D|E|F
```

Коректно записани целобројни литерали су:

```
0   ►(0)       125   ►(125)       3567       ►(3567)    +0B_11_00 ►(12)
0564►(372)     -0XABC►(-2748)     0x23_a4    ►(9124)    01011     ►(521)
56  ►(56)      04343 ►(2275)      0XE6_53_a  ►(943418)  0b1011    ►(11)
```

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

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

Реални литерали

Реални литерали су константе које се записују у облику покретне тачке (покретног зареза). У Јави се разликују два типа реалних литерала: `float` и `double`. Разлика између ових типова појављује се само у прецизности записа литерала. Реални литерали могу да се изразе у позиционом запису или експоненцијалном запису. Уколико је литерал типа `float` слово `f` или `F`, мора се навести на крају литерала. Тип `double` је подразумевани тип за реални литерал те се слово `d` или `D` не мора навести.

```
<реални литерал> ::= <мантиса>[<експонент><индикатор>]
<мантиса> ::= <декадни цео број>
    |<декадни цео број>.[<декадни цео број без знака>]
    |[<знак броја>].<декадни цео број без знака>
<експонент> ::= (e|E)<декадни цео број>
<индикатор> ::= f|F|d|D
```

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

Следеће речи су синтаксно исправни реални литерали:

```
23.57           8.879f              .345d
.569            0.569F              4455.D
3.14            2e-5f               0.003e+4d
123E-5          1.456575e+12F       3.5E7
```

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

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

Логички литерали

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

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

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

Знаковни литерали

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

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

Графички симбол је један знак између апострофа.

Примери записа графичких симбола су:

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

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

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

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

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

Литерали-ниске

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

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

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

Примери ниски:

```
""                                    ► празна ниска
"Programiranje i matematika"          ► ниска са различитим писмима
"Ovo je navodnik \", a ovo ne \u3232" ► ниска са ескејп-секвенцама
```

5.2.4. Сепаратори

У Јави постоји неколико знакова који служе за раздвајање једне врсте елементарних конструкција од других. На пример, у сепараторе спада симбол ; који служи за раздвање наредби у Јави.

Сепаратори су дефинисани на следећи начин:

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

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

5.2.5. Оператори и изрази

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

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

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

Примери неких добро формираних израза су:

```
args.length
funk(x,y)
pera.mika.zika(a,b)
stampaj(a,b,c+funk(b,a))
```

```
Math.random()
System.out.print(niz)
```

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

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

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

Аритметички оператори и изрази

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

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

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

Аритметички израз је дефинисан на следећи начин:

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

Уочава се рекурентна природа горње дефиниције – аритметички израз је дефинисан преко аритметичког израза. Дефиниција за

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

У следећем примеру указује се на редослед извршења операција при евалуацији аритметичког израза `7*3-7/2+4`. Редослед извршења операција приликом евалуације израза прецизно је дефинисан табелом оператора на крају секције 5.2.5. Пример извршавања је:

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

Битовни оператори и изрази

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

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

Прва четири међу овим операторима (`&`, `|`, `~` и `^`) су логички битовни, и њима се над одговарајућим битовима операнада реализују логичке операције конјункције, дисјункције, ексклузивне дисјункције и негације.

Последња три побројана оператора (тј. `<<`, `>>` и `>>>`) су оператори померања и њима се реализују операције померања за једно место над бинарним садржајем операнда: логичко померање улево, логичко померање удесно и аритметичко померање удесно.

Битовни изрази увек, као резултат извршавања, производи број. С обзиром на ову чињеницу јасно је да постоји велика сличност између

неких битовних и аритметичких израза – разлика је једино у симболима оператора који се користе и у њиховом приоритету и асоцијативности.

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

Пример примене неких битовних оператора је дат испод.

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

Релациони оператори и изрази

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

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

У Јави се, исто као и у програмском језику C, за испитивање да ли су два операнда једнака користи симбол `==` (двоструки знак једнакости). За испитивање да ли су два операнда различита користи се оператор `!=`. Резултат примене релационих оператора је увек логичког типа (`false` или `true`).

Синтакса релационог израза може бити дефинисана на следећи начин:

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

У горњим формулама фигурише и елемент `<инстанцна променљива>`, описана у секцији 8.1.1, која се односи на поређење објеката описано у секцији 8.1.4.

Може се приметити да се у дефиницији користи и `<логички израз>`, који је дефинисан тек у следећој секцији. Разлог је узајамна рекурзија у дефиницији ова два концепта. Наиме, иако је у горњем делу релациони израз дефинисан преко логичког, и логички израз ће бити дефинисан преко релационог израза.

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

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

Логички оператори и изрази

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

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

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

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

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

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

Условни оператор и израз

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

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

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

Инстанцни оператор и израз

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

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

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

Оператор `instanceof` враћа вредност `true` ако је објекат, чији је назив дат са леве стране оператора `instanceof`, примерак наведене класе (или интерфејса), чије је име дато са десне стране оператора `instanceof`. У супротном враћа вредност `false`. О овом оператору и његовом коришћењу биће посвећена већа пажња у секцији 8.5.2.

Оператор и израз за прављење објекта

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

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

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

Оператори доделе и изрази доделе

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

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

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

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

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

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

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

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

Саставни оператор доделе служи за компактнији запис наредби увећања, умањења и других сличних операција. Наредба `<идентификатор>=<идентификатор><оператор><израз>` се може краће записати у облику `<идентификатор><оператор>=<израз>`.

Следи неколико примера коришћења саставног оператора доделе:

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

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

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

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

Оператори: арност, асоцијативност, приоритет

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

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

Табела 2 приказује приоритете и асоцијативност Јава оператора. Када је у питању приоритет, мањи број указује на виши приоритет, на пример, оператор `++` је приоритетнији од оператора `||`.

Табела 2. Оператори и њихови приоритети и асоцијативност

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

5.2.6. Белине

Белина је знак који нема графички приказ на излазном уређају. Белине служе за међусобно раздвајање елементарних конструкција и за обликовање програма. Белине могу бити: размак, табулатор, знак за крај реда, знак за нову страну и знак за крај датотеке.

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

Белина нема никакав утицај на даљи рад програма. Другим речима, око сваке металингвистичке променљиве у формули може бити произвољан број белина, које неће променити семанитку (значење) формуле. У металингвистичким формулама које следе то неће бити посебно наглашено, већ ће се постављати један размак између конструкција које морају бити раздвојене белинама.

(Напомена: овде се не мисли на белине које се појављују у оквиру ниски – оне, наравно, утичу на семантику програма, тј. нису неутралне конструкције програмског језика Јава.)

5.2.7. Коментари

Коментари служе да се објасне поједина места у програму. Коментари су, пре свега, намењени човеку, али се у Јави могу искористити и за аутоматско генерисање документације. Конструкције Јаве су често довољно јасне па коментари понекад могу бити и сувишни. Пожељно је на почетку програма објаснити чему програм служи, ко га је писао, када је написан итд.

У Јави постоје 3 врсте коментара: вишелинијски (коментар у стилу језика C), једнолинијски (коментар у стилу језика C++) и документациони.

Једнолинијски коментари се могу писати од почетка реда или у реду где се завршава нека наредба. На пример, има смисла писати:

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

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

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

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

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

Коментари се могу описати Бекусовом нотацијом на следећи начин:

```
<коментар> ::= <једнолинијски коментар>|<вишелинијски коментар>
    |<документациони коментар>
<једнолинијски коментар> ::= //{<не крај реда>}<знак за крај реда>
<не крај реда> ::= <Unicode знак различит од знака за крај реда>
<вишелинијски коментар> ::= /*{{< не звезда>}*{<не коса црта>}}*/
<не звезда> ::= <Unicode знак различит од знака ‘*’>
<не коса црта> ::= <Unicode знак различит од знака ‘/’>
<документациони коментар> ::= /**{{<не звезда>}*{<не коса црта>}}*/
```

5.3. Типови података у Јави

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

1. Тип података одређује скуп вредности које могу бити додељене променљивама или изразима.

2. Над њима се могу извршавати одређене операције, односно функције.

3. Тип променљиве или израза може се одредити на основу изгледа или описа, а да није неопходно извршити неко израчунавање (на пример, не мора се стварно извршити операција дељења за конкретне бројеве да би се знало да је резултат реалан број у рачунарском смислу).

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

Увођењем типова података омогућава се да преводилац лакше открије неисправне конструкције у језику, што даље представља један вид

семантичке анализе. Типови података доприносе прегледности програма, лакшој контроли операција од стране преводиоца и већој ефикасности преведеног програма.

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

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

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

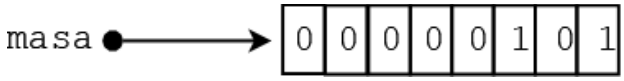
```
<тип> ::= <примитивни тип>|<објектни тип>
```

5.3.1. Примитивни типови података

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

```
byte masa = 5;
```

У меморији рачунара ће постојати локација којој је додељено име `masa` и која ће садржати вредност `5` у бинарном облику, као што приказује Слика 31.



Слика 31. Представљање променљиве целобројног типа

На слици није приказано, али се подразумева да ће се са леве стране налазити доста већи број нула, јер целобројни тип у Јави заузима најмање 4 бајта.

У зависности од конкретног примитивног типа, величина меморијске локације може бити различита, али она ће увек садржати вредност примитивног типа.

Сваки примитивни тип карактерише нека резервисана реч. Примитивни тип може бити аритметички или логички. Логички тип се описује резервисаном речју `boolean`:

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

Аритметички тип може бити целобројни или реални.

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

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

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

Реални тип може бити једноструке и двоструке тачности, тј:

```
<реални тип> ::= float|double
```

Целобројни тип података

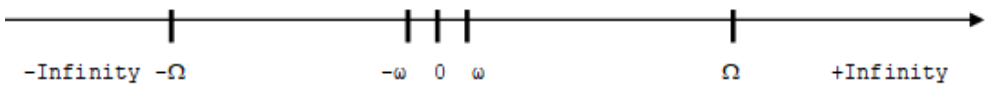
У оквиру целобројних типова података се разликују следећи типови: `byte`, `short`, `int`, `long` и `char`. За сваки од тих типова постоји одређени интервал (одређен величином меморијске речи) из којег се могу узимати

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

Реални тип података

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

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



Слика 32. Бројна оса за реални тип података

Овде су са ω и Ω означени, редом, минимални и максимални реалан број по апсолутној вредности у оквиру одговарајућег реалног типа.

Реални бројеви из области (-∞, - Ω) не могу се регистровати и ако је резултат неке операције из тог интервала, наступило је прекорачење (енг. overflow) – тај резултат третира се као -∞ (-Infinity). Слично, бројеви из области (Ω, +∞) не могу се регистровати и третирају се као +∞ (+Infinity).

Реални бројеви из области (-ω, 0) ∪ (0, ω) такође не могу бити регистровани. Ако је резултат неке операције из ове области, појављује се поткорачење (енг. underflow), али тај резултат се третира као нула (зато што је реч о веома малим бројевима – блиским нули). Међутим, при операсању са оваквим бројевима треба бити опрезан јер се могу добити некоректни резултати.

Реални бројеви из области [-Ω, -ω] ∪ {0} ∪ [ω, Ω] могу се регистровати у Јави. У ствари, тачно се могу регистровати само тзв. централни бројеви, а сви остали само приближно. Ако је x централни број, тада се сви реални бројеви (у математичком смислу), из довољно мале околине за x, замењују бројем x.

На реални тип података могу да се примењују:

- релациони и
- аритметички оператори.

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

Знаковни тип података

Знаковни тип је одређен скупом знаковних литерала из Unicode 2.0[10] и операцијама над њима. Сваки знак у Unicode 2.0 је једнозначно одређен

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

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

Логички тип података

Логички (`boolean`) тип је окарактерисан:

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

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

5.3.2. Објектни тип

Објектни тип у Јави може бити:

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

Објектни тип се може дефинисати на следећи начин:

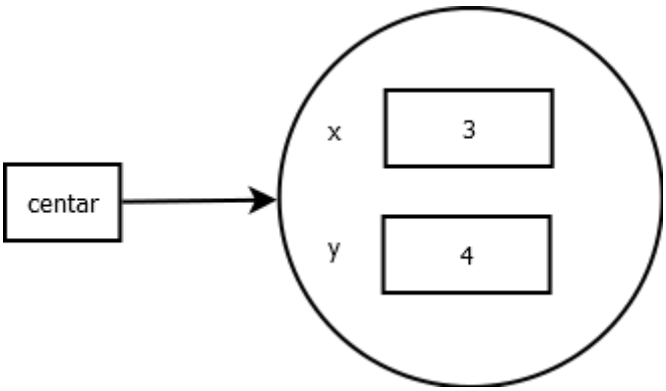
```
<објектни тип> ::= <кориснички објектни тип>|<низовни тип>
    |<набројиви тип>
<кориснички објектни тип> ::= <тип класе>|<тип интерфејса>
```

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

Објекти и класе

Јава објекти постоје само током извршавања програма. На нивоу меморије променљива која представља објекат (референца) у Јави, у ствари, садржи информацију којом се реферише на део меморијског простора који заузима дати објекат.

На пример, инстанцна променљива `centar` представља тачку у дводимензионалном простору са координатама `(3,4)`, што се визуелно може представити на следећи начин.



Слика 33. Представљање променљиве објектног типа

Дакле, вредност која одговара променљивој `centar` је адреса простора у ком су смештене координате тачке.

Дефиницијом класе практично се дефинише нови тип у Јави и сваки инстанца те класе ће имати струкутуру (атрибуте и методе) који су одређени дефиницијом класе.

Дефиниција класе би, коришћењем Бекусове нотације, могла бити дата на следећи начин:

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

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

5.3.3. Експлицитна конверзија типа

Конверзија типова у програмском језику Јава је, што се тиче примитивних типова, иста као што је то у програмском језику C – ако се приликом евалуације израза у Јави појави потреба да се ужи тип интерпретира као шири (на пример, `int` у `long`, `int` у `double` и слично), то ће бити аутоматски урађено. Дакле, у том случају програмер нема обавезу да означава да ће наступити конверзија.

Међутим, ако програмер уочи потребу да податак ширег типа буде тумачен као податак ужег типа (и на тај начин се можда изгуби прецизност или опсег), тада мора применити експлицитну конверзију типа (кастовање) и употребити тзв. израз кастовања.

Имплицитна и експлицитна конверзија типа постоје не само за примитивни тип, већ и за типове класа и интерфејса, при чему се информација о томе који је тип шири (општији) а који ужи, ослања на односе наслеђивања класа (секција 8.5) и проширења и имплементације интерфејса (секције 9.2.3 и 9.2.2).

Израз кастовања има следећу синтаксу:

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

5.4. Променљиве

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

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

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

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

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

A. атрибути инстанце (зову се и атрибути примерка или атрибути објекта) и

B. атрибути класе (или статички атрибути).

Атрибути инстанце су везани за животни век конкретног објекта неке класе.

Атрибути класе, са друге стране, немају везе са конкретним објектима, већ са самом класом – објекат уопште не мора бити направљен да би постојао атрибут класе.

Локалне променљиве и аргументи метода се карактеришу сличним понашањем, и њихов животни век одговара животном веку конкретног позива метода у којем се они налазе. Локалне променљиве могу бити и примитивног и референтног типа.

Још једна карактеристика локалних променљивих је да оне морају имати фиксирану величину, унапред познату још у фази компајлирања програма. Разлог овоме је чињеница да се позиви метода ослањају на стек. Наиме, да би стек могао да функционише, тј. да би стек оквири за

узастопне позиве метода могли да се слажу један на други, величина сваког оквира мора бити унапред позната (не динамичка). Овај услов је очигледно испуњен будући да примитивни типови имају фиксну величину, а када су у питању објекти направљени унутар неког метода, мора се нагласити да се на стеку налазе само референце ка њима, док је њихов садржају на хипу.

У поглављу 8 биће детаљније објашњен начин рада са атрибутима, опсег њиховог важења, њихови меморијски аспекти итд.

5.4.1. Декларација и иницијализација вредности променљиве

Свака променљива мора бити декларисана, при чему се одређује и тип променљиве. Опционо, може се доделити и почетна вредност (иницијализација).

Променљиве примитивних типова се декларишу и по потреби иницијализују слично као што је рађено у програмском језику C, коришћењем резервисаних речи `byte`, `short`, `int`, `long`, `char`, `boolean`, `float`, `double` за тип променљиве.

На сличан начин се декларишу и по потреби иницијализују објектне променљиве, тј. променљиве типа класе. Наиме, сваки објекат мора бити инстанца (примерак) неке дефинисане класе. Име класе које се користи за декларисање променљиве доводи до тога да декларисана променљива добија тип те класе.

На пример, ако се дефинише класа `Osoba` на следећи начин:

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

тада има смисла декларисати променљиве:

```
Osoba pera, mika;
```

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

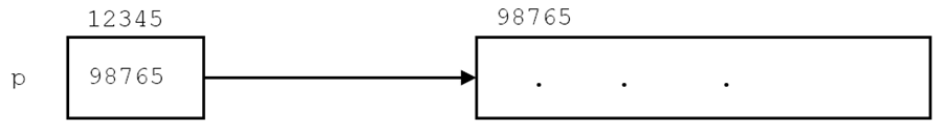
Начин записа података објектног типа (тј. објеката) у меморији се разликује од начина записа података примитивног типа. У оба случаја, подацима у меморији се приступа преко променљивих. Међутим, док код примитивних типова променљиве садрже податке са којима се оперише, код објектног типа променљиве представљају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

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

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

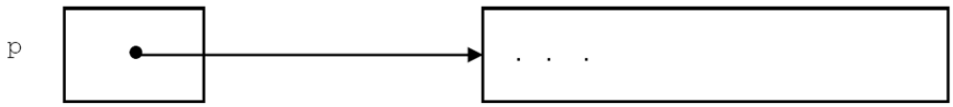
Овим је постигнуто да `p` садржи референцу на адресу у меморији од које почиње запис направљеног објекта.

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



Слика 34. Референца на објекат

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



Слика 35. Референца на објекат

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

Следеће металингвистичке формуле описују декларацију и иницијализацију променљивих:

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

Дакле, у декларацији (и евентуалној иницијализацији) најпре се наводи тип променљиве. Потом следи један или више идентификатора раздвојених зарезима, који представљају називе променљивих. Специјално, након назива променљиве може уследити и иницијализација, која је заснована на претходно описаним изразима.

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

Следе примери декларација и иницијализација локалних променљивих.

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

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

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

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

5.5. Наредбе

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

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

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

5.5.1. Наредба израза

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

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

```
<наредба израза> ::= <израз>;
```

5.5.2. Блок

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

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

Уочава се да је претходна дефиниција блока рекурзивна:

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

Тела класа, метода итд. су блокови.

Блок може садржати друге блокове и било које друге наредбе које се извршавају једна за другом док се не наиђе на наредбу за промену тока управљања.

У блоку могу бити декларисане локалне променљиве. Оне су видљиве (могу се користити) само од места декларисања до краја блока. Локална променљива не може бити коришћена уколико јој није додељена почетна вредност.

Пример 1. Наредни пример приказује шта су блокови у Јави и каква је веза видљивости локалних променљивих и блокова.□

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

Тело класе је блок, као и тело сваког метода. У оквиру метода `main()` се налази блок који, чини се, нема никакву намену. Међутим, он има утицај на видљивост локалне променљиве `x`, која неће бити видљива (дефинисана) ван њега. То омогућава да се у оквиру наредног блока (бројачки блок за наредбу `for`) поново користи променљива са истим називом. Ова два блока су независна (ни један се не садржи у другом) што омогућава вишеструко дефинисање променљиве са истим називом – `x`.

Покушај поновне декларације променљиве `x` у оквиру бројачког блока није могућ па је кôд те декларације коментарисан – у супротном се програм не би компајлирао.

Променљива `y`, која је дефинисана у оквиру блока наредбе `if`, није видљива ван овог блока. Из тог разлога је коментарисан кôд који ван тела наредбе `if` оперише са променљивом `y` – у супротном се програм не би компајлирао.

Програм штампа следећи текст.

```
Здраво свете
5
0
1
```

```
2
3
4
5
6
7
8
9
```

5.5.3. Наредбе гранања

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

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

Наредба if

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

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

Непотпуна наредба if

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

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

У следећем делу програма:

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

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

Потпуна наредба if

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

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

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

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

Уз напомену да испред наредбе `else` мора да стоји `;` (тачка-зарез).

У овом примеру се, после израза у `if` наредби и у `else` грани, појављују блокови.

```
...
if (figura.equals("krug")){
  obim = 2*pi*r;
```

```
  povrsina = pi*r*r;
}else{
  obim = 4*a;
  povrsina = a*a;
}
...
```

Блок испред речи `else`, у овом случају, не сме да се завршава знаком `;` (тачком-зарезом).

Наредба која веома подсећа на потпуну наредбу `if` је наредба условног израза. Условни оператор и условни израз су уведени раније у оквиру секције 5.2.5. Наредба условног израза омогућава сажет запис потпуне наредбе `if` у програму. Тако уместо:

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

може се писати:

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

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

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

да ли ће се `Nar2` извршити ако је:

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

Из претходних описа није јасно како ће се извршити. У језику Јава је, према дефиницији, наредба `else` придружена другој наредби `if`. То значи да ће се претходна конструкција реализовати према опису под 1.

Ако је потребна релизација према опису под 2., онда другу наредбу `if` треба сместити у блок, тј. треба написати:

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

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

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

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

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

Пример 2. Написати програм за израчунавање вредности функције:

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

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

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

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

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

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

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

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

Извршавањем било којег од наведених програма добија се резултат попут следећег:

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

или

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

Пример 3. Написати програм за израчунавање минимума два цела броја. Бројеви се учитавају преко стандардног улаза.□

Задатак се једноставно решава коришћењем једне потпуне наредбе `if`.

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

У претходном решењу није испитивано да ли су бројеви једнаки. Ако су једнаки, минимум је било који од њих. (У овом случају то ће бити вредност променљиве `b`).

За потребе учитавања корисничког уноса искоришћен је објекат класе `Scanner` (пун назив класе је `java.util.Scanner`) и њен метод `nextInt()`, који омогућава учитавање целог броја са стандардног улаза. Детаљнији примери употребе класе `Scanner` ће бити дати у секцији 6.4.

Пример извршавања је дат испод.

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

Пример 4. Написати програм за уређење три реална броја из интервала [0,1] у неопадајући поредак.□

Претпоставимо да се реални бројеви из интервала [0,1] генеришу на случајан начин и памте преко променљивих: `a`, `b` и `c`. Извршиће се измена садржаја ових променљивих тако да се у `a` налази број мањи или

једнак осталима. То се постиже упоређивањем садржаја променљиве `a` са садржајем променљиве `b`. Ако је у променљивој `b` мања вредност, врши се размена ових двеју променљивих. Затим се понавља поступак за `a` и `c`. Након тога у `a` се налази најмања вредност. Након упоређивања `b` и `c`, у `b` се смешта мања вредност и тиме се обезбеђује да низ вредности `a`, `b` и `c` буде неопадајући.

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

Следи пример извршавања.

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

Наредба switch и наредба break

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

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

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

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

Литерали који се појављују иза резервисане речи `case` морају бити истог типа као и селектор. Додатно, сви литерали морају бити међусобно различити.

Семантика наредбе `switch` детаљније је објашњена следећим примером.

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

Ток извршавања је следећи:

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

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

3. Извршава се наредба `Nar3`.

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

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

```
public class BrojDanaUMesecu {
    public static void main(String[] args) {
        int g, m;
        java.util.Scanner sc = new java.util.Scanner(System.in);
        System.out.println("Унеси годину");
        g = sc.nextInt();
        if (g < 1)
```

```
                System.out.println("Неисправан унос " + g);
        System.out.println("Унеси редни број месеца [1-12]");
        m = sc.nextInt();
        switch (m) {
        case 2:
                if (g % 400 == 0 || (g % 4 == 0 && g % 100 != 0))
                        System.out.println("Број дана је 28.");
                else
                        System.out.println("Број дана је 29.");
                break;
        case 4:
        case 6:
        case 9:
        case 11:
                System.out.println("Број дана је 30.");
                break;
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
                System.out.println("Број дана је 31.");
                break;
        default:
                System.out.println("Неисправан унос " + m);
        }
    }
}
```

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

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

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

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

5.5.4. Наредбе понављања

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

У Јави постоје четири врсте наредби за циклусе.

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

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

Наредба while

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

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

Ефекат наредбе `while` је следећи:

1. Израчунава се вредност логичког израза.

2. Ако је вредност израза истинита, извршава се наредба и враћа се на корак 1.

3. Ако је вредност израза неистинита, завршава се извршавање наредбе `while` и прелази се на прву следећу наредбу иза наредбе `while`.

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

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

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

Следи испис добијен извршењем програма.

```
1 2 3 4 5 6 7 8 9 10
```

Пример 7. Написати програм за израчунавање $S = \sum_{i=1}^{n} \frac{1}{i^3}$.□

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

```
                }
                System.out.println("Тражени збир је: " + s);
        }
    }
}
```

Следи испис добијен извршењем програма.

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

Наредба do-while

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

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

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

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

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

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

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

Следи испис добијен извршењем програма.

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

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

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

постиже се исти ефекат као и наредбама:

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

Све што се описује помоћу наредбе `while`, може се описати помоћу наредбе `do-while` и наредбе `if`.

Наредбе у телу циклуса код наредбе `do-while` морају да се изврше бар једанпут, за разлику од наредбе `while`, где не морају.

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

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

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

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

Следи испис добијен извршењем програма.

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

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

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

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

```
        System.out.println("Унесите називе школских предмета");
        do {
                s = sc.next();
                n++;
                if (s.equals("програмирање"))
                        bp++;
                if (s.equals("математика"))
                        bm++;
                if (s.equals("физика"))
                        bf++;
        } while (!s.equals("КРАЈ"));
        System.out.println("Број учитаних речи " + n);
        System.out.println("Број појава речи 'програмирање'  је: "
                                                                + bp);
        System.out.println("Број појава речи 'математика'  је: " + bm);
        System.out.println("Број појава речи 'физика'  је: " + bf);
    }
}
```

Следи пример извршавања програма.

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

Наредба for (бројачки циклус)

Наредба `for` је моћна наредба за опис циклуса. Најчешће се користи за опис циклуса код којих је број понављања наредби унапред познат. У том смислу она представља типичну „бројачку“ наредбу.

Синтакса наредбе `for` је следећа:

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

Наредба `for` се извршава на следећи начин:

1. Најпре се извршава иницијализациони део петље `for`. Ако нема садржаја у овом делу, не врши се иницијализација. Начелно, инцијализација је израз у којем се подешава вредност управљачке променљиве петље `for`. Иницијализација се извршава само једанпут.

2. Након иницијализације, израчунава се вредност логичког израза који представља услов останка у петљи (ако постоји). Ако израз није дат, његова подразумевана вредност је `true`.

3. Ако израз има вредност `true`, извршава се наредба петље (тело петље), а затим се извршава итерација петље и поново се иде на корак 2.

4. Уколико израз има вредност `false`, окончава се извршавање петље.

У наредби `for` често се појављује променљива помоћу које се врши некакво пребројавање и која се назива бројач. Пребројавање може да се врши уз увећање вредности бројача (бројање унапред), почев од неке почетне вредности, све док је логички израз тачан или уз умањење вредности бројача (бројање уназад).

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

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

Програм производи тражени испис.

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

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

Следи резултат једног могућег извршавања.

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

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

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

```
}
```

Овај задатак једноставно можемо решити примењујући наредбу `for` за бројање унапред. То важи генерално: свуда где се примењује наредба `for` за бројање унапред, може се применити и наредба `for` за бројање уназад. Важи и обрнуто. Међутим, у неким ситуацијама програм је прегледнији ако се користи бројање унапред, а другим ако се користи бројање уназад.

Напоменимо да постављени задатак можемо једноставно решити (без употребе циклуса) користећи формулу за израчунавање суме аритметичког низа. Међутим, овде је циљ да се прикажу могућности наредбе `for`, а не да се нађе што једноставније/ефикасније решење.

Следи резултат једног могућег извршавања.

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

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

Пример 14. Реализовати бесконачни `for` циклус.☐

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

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

ограничени број битова, улазити у вишеструка прекорачења током трајања извршавања.

Пример 15. Написати програм којим се израчунава сваки наредни степен променљиве (чија почетна вредност 0.9) све док не постане мањи или једнак 0.1. Овде наредба `for` практично замењује наредбу `while` будући да се изрази за иницијализацију и за итерацију изостављају.☐

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

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

```
0.9^22 = 0.0984770902183612
```

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

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

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

```
}
```

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

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

У оквиру наредбе `for` се може користити више променљивих, што је илустровано на наредном примером.

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

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

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

Следи испис произведен извршавањем програма.

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

Наредба break и циклуси

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

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

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

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

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

Следи испис произведен извршавањем.

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

Наредба continue и циклуси

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

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

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

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

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

Уколико је број `x` негативан, применом наредбе `continue` се моментално прелази у наредну итерацију.

Следи испис произведен извршавањем овог програма.

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

5.5.5. Обележена наредба

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

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

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

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

или:

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

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

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

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

Спољни `for` циклус је обележен са `test` па се позиви `break test` и `continue test` односе на спољни циклус. Код `break` наредбе додавање обележја није било неопходно пошто она већ директно припада спољном циклусу, али свакако не шкоди. У унутрашњем циклусу проверава се да ли се текст `podstring`, на одређеној позицији испод текста `tekst`, поклапа са истим. Ако се наиђе на прво непоклапање,

нема смисла даље проверавати поклапање, па се одмах завршава унутрашњи циклус и започиње наредна итерација спољашњег, односно померање („клизање“) `podstring` текста за једно место десно у односу на `tekst`.

Следи испис произведен извршавањем овог програма.

```
Пронађена
```

5.5.6. Празна наредба

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

```
<празна наредба> ::= ;
```

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

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

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

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

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

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

5.6. Резиме

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

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

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

5.7. Питања и задаци

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

2. Шта су граматика, синтакса и семантика језика?

3. Дефинисати реалне бројеве у модификованој Бекусовој нотацији.

4. Да ли су следеће речи идентификатори? Образложити одговор.

    - `new`
    - `a1`
    - `prvi Broj`
    - `8$4`
    - `prvi_Broj`
    - `finally`
    - `$x#`
    - `true`

5. Који од следећих литерала су исправно записани? Образложити одговор.

   - `"abc " def"`
   - `'\x'`
   - `'0'`
   - `23.14-56`
   - `23.14E5`
   - `12`
   - `0X45G`

6. Да ли има разлике у префиксној и постфиксној примени оператора `++` и `--`? Илустровати на примерима.

7. Нека су дати:

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

   Колике су вредности променљивих `a, b, c, d, uslovA` и `uslovB` након извршења наведног дела кода?

8. Нека је дат 8-битни запис целог неозначеног броја 00010101. Извршити померање за једну позицију улево и објаснити како померање утиче на декадну вредност датог броја. Након тога, извршити померање броја за једну позицију удесно и објаснити како померање утиче на декадну вредност датог броја.

9. За шта се користе коментари? Које врсте коментара постоје у програмском језику Јава?

10. Објаснити зашто је Јава строго типизиран језик. Истражити који су још програмски језици строго типизирани, као и шта је разлика између строго и слабо типизираних језика?

11. У чему је разлика између примитивних и референтних (објектних) типова у програмском језику Јава? Упоредити референтни тип из програмског језика Јава и показивачки тип из програмског језика C.

12. Размотрити ситуације када се користе различити целобројни типови `byte`, `short`, `int`, `long` и `char`.

13. Када долази до прекорачења, а када до поткорачења при употреби реалних типова?

14. Које објектне типове разликујемо у програмском језику Јава?

15. Како се у меморији записују подаци објектног типа, а како подаци примитивног типа?

16. Шта су променљиве и које врсте променљивих се разликују у програмском језику Јава?

17. За сваку наредбу у коду примера 1 одредити којој врсти наредби припада.

18. Илустровати на примеру ситуације када је потребно користити потпуну `if` наредбу, као и ситуацију када је довољна употреба непотпуне `if` наредбе.

19. Дат је следећи део кода:

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

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

20. Дат је следећи део кода:

```
int a, b;
System.out.println("Унеси 1 цели број");
a = sc.nextInt();
```

```
switch(a){
      case 100:
      case 1000:
      case 10000:
            b = a/10;
            break;
      case 100000:
            b = a/100;
            break;
      default:
            b = a;
}
```

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

21. Задатак из примера 9, за израчунавање дужине (броја цифара) унетог целог броја типа `long`, решити употребом `for` петље.

22. Задатак из примера 15, за израчунавање сваког наредног степена променљиве (чија почетна вредност је већа од 0, а мања од 1) све док не постане мањи или једнак 0.1, решити употребом `while` петље.

23. Илустровати на примеру ситуацију у којој постоји разлика између употребе `do-while` наредбе понављања и употребе `while` или `for` наредби понављања.

24. Да ли је неопходно постојање наредбе `do-while`?

25. Задатак из примера 18, који који сумира све унете бројеве док се не унесе први негативан број, решити без употребе наредбе `break`.

26. Задатак из примера 19, који сумира све позитивне бројеве док се не унесе број 0, након чега се извршавање програма прекида, написати без употребе наредби `break` и `continue`.

27. Објаснити шта су обележене наредбе и илустровати на примеру ситуацију када је корисно да се користе.

28. Шта је празна наредба и у којим ситуацијама се користи?