Лекция 8

1. Променливи в езика Java.

А. Декларация на променливи

Променливите са величини, които приемат стойности от различен тип данни. Декларирането на променливи се извършва посредством следната синтактична конструкция:

тип име на променлива ;

където ***тип*** е типът данни които може да съхранява променливата, а име на променливата е нейният идентификатор (наименование). Може да се декларира променлива от всеки валиден тип, включително и от простите типове, които бяха представени по-горе. Когато се обявява дадена променлива, всъщност се създава екземпляр (инстанция) от дадения тип данни. Типът на променливата не може да се променя по време на изпълнение на програмата. Ето някои примери за деклариране на променливи:

```
int im, in, k, l ;
long lk, lf ;
double pi, exp_1 ;
Boolean bool1, bool2 ;
```

Преди използването на дадена променлива, трябва да й се зададе стойност. Един от начините за задаване на стойности на променливите е чрез използване на синтактичната конструкция за **инициализация** на променливи. За целта, когато се декларира дадена променлива, се задава и начална стойност посредством конструкцията:

тип променлива = стойност ;

Тук 'стойност' е литерална константа, която представлява стойността, която променливата получава при инициализацията. Примери:

```
int number = 10 ;
char xch = 'X' ;
float pi = 3.14159 ;
```

Освен чрез константи, променливите могат да бъдат инициализирани **динамично** по време на създаване на променливата, посредством произволен валиден израз. Ето един пример на динамично инициализиране на променлива:

```
class Volume {
      public static void main () {
       double radius = 4, height = 5 ;
       double volume = 3.14159*radius*radius*height ;
      System.out.println ("Volume is " + volume) ;
      }
}
```

В тази елементарна програма са декларирани три локални променливи **`radius, height`** и **`volume`**. Първите две променливи се инициализират с литерални стойности, а третата се инициализира динамично, като й се присвоява стойността на израз, чрез който се изчислява обема на цилиндър.

Б. Област на видимост и съществуване на променливите.

В разгледаните до тук примери, всички променливи бяха декларирани в началото на метода main(). Java дава възможност да се декларират променливи в рамките на произволен блок. Под блок в език Java се разбира съвкупност от оператори и декларации, оградени с двойка фигурни скоби. Всеки блок дефинира собствена област на видимост. Тази област определя кои обекти са достъпни (видими) за други части от програмата, а така също определя и времето за съществуване на тези обекти.

Повечето от програмните езици дефинират две категории на област на видимост на обектите: глобална и локална. Въпреки че и двете се поддържат от Java, те не са най-добрият начин за

категоризиране на областите на видимост в Java. Най-важните области на видимост в Java са тези дефинирани от един клас, и тези дефинирани от даден метод. Тук ще бъде разгледана областта на видимост дефинирана от един метод (функция).
Като общо правило променливите, декларирани в дадена област, не са достъпни от код извън тази област на видимост. По този начин, когато се декларира една променлива в даден блок (област на видимост), тя се локализира в тази област и се защитава от неоторизиран достъп или модификация. В общи линии, правилата за областите на видимост осигуряват основите на капсулирането.
Областите на видимост могат да се влагат. Всеки път, когато се създава нов блок, всъщност се дефинира нова вложена област на видимост, защото всеки блок се създава в рамките на даден клас или метод. Обектите, които се декларират във вътрешната област на видимост стават невидими за външната област на видимост. Ето един пример за демонстрация на областта на видимост на променливи:

```
// Демонстрира област на видимост за блок.
class Scope {
     public static void main (String args[]) {
     int x ; // Видима в цялата област на метода main()
     x = 10 ;
          {   // Начало на нов блок (област на видимост)
          int y = 25 ;  // Видима само в този блок
          System.out.println("x and y:"+x + "  " + y) ;
          x = x + y ;
          }
           // y = 10 ; Грешка, y не е достъпна
     System.out.println (" x = "+ x ) ;
     }
}
```

2. Оператори в Java

Както в повечето програмни езици, така и в Java, основните структурни единици в програмите са операторите. Синтактическите правила за съставяне на операторите включват служебни думи от езика, изрази в които се използват различни типове операции и някои специфични синтактически конструкции.
Като наследник на C++, Java предоставя богат набор от операции. Те се разделят на четири основни групи: аритметични, побитови, релационни (операции за сравнение) и логически. Към тях се отнасят и някои допълнителни операции, които се използват при специални случаи. Списъкът на използваните в Java операции са дадени по групи:

Аритметични операции

Оператор	*Предназначение*	*Употреба*
* / %	умножение деление остатък при деление	expr1*expr2 expr1/expr2 expr1%expr2
+ -	събиране изваждане	expr1+expr2 expr1-expr2
++ --	Инкрементиране Декраментиране	Var++ (++Var) Var—(--Var)

В таблицата, с **expr1** и **expr2** са означени аритметични изрази, а с **Var** – променлива. Операциите +,-,*,/ се използват както във повечето програмни езици и в алгебрата. Те са приложими към всеки вграден тип числови данни. Могат да се използват и с данни от типа **char**. Както и в език C++, резултатът от деление "/" на цели числа е цяло число (например

резултатът от 10/3 е числото 3). Затова е въведена и допълнителната операция "%", която се използва за получаване на остатък при деление на цели числа.
Операциите инкрементиране (нарастване с единица) и декрементиране (намаляване с единица) също са наследство от C++. Те могат да се прилагат към всички видове числови данни и към данни от типа **char**. Операцията **x++** е идентична с оператора `x = x + 1` , а операцията **x–** с оператора `x = x - 1`. Когато тези операции се използват в изрази, от значение е дали операцията инкрементиране или декрементиране се поставя пред или след съответният операнд. Когато '++' или '—' се разполага пред операнда, Java изпълнява тези операции, преди да извърши действията в израза. Когато '++' или '—' са разположени след операнда, Java изпълнява израза и след това увеличава или намалява стойността на съответната променлива. Ето един пример:

```
x = 8 ; y = ++x ;
```

Стойността на ***y*** и на ***x*** в този случай ще бъде 9, докато ако използване изразите:

```
x = 8 ; y = x++ ;
```

стойността на ***y*** ще бъде 8, а на ***x*** ще бъде 9.

Релационни операции (операции за сравнение)

Оператор	*Предназначение*	*Употреба*
< <= > >=	по-малко по-малко или равно по-голямо по-голямо или равно	expr1<expr2 expr1<=expr2 expr1>expr2 expr1>=expr2
== !=	Равенство неравенство	expr1==expr2 expr1!=expr2

Логически операции

Оператор	*Предназначение*	*Употреба*
!	Логическо отрицание	!expr1
& | ^	Логическо И Логическо ИЛИ Изключващо или	expr1==expr2 expr1!=expr2 expr1^expr2
&&	Съкратено И	expr1 && expr2
||	Съкратено ИЛИ	expr1 || expr2

Стойностите на изразите в които участват операциите от тези два типа са логически (*true u false*), т.е. те са тип **boolean**. Различават се по това, че релационните операции участват в изрази, в които се разглеждат взаимоотношенията между стойности на променливи или изрази, а логическите се използват за образуване на логически изрази. Следователно, релационните операции могат да се прилагат само към обекти и стойности, които могат да се подреждат по големина, Такива са числовите данни и данните от тип **char**. Към логическият тип данни те не могат да се прилагат, тъй като между логическите стойности *true u false* не съществува подредба по отношение на стойността (например в Java израза *true* > *false* е безсмислен).
Java предоставя специални съкратени варианти на логическите операции **И** и **ИЛИ**, които се използват за създаване на по-ефективен код. Разликата между пълните и съкратени форми на логическите операции **И** и **ИЛИ** е в начина на изпълнение на логически изрази с тези операции. При съкратените форми на тези операции, изпълнението на логическите изрази се прекратява, когато стойността на израза стане известна, дори и изпълнението на израза да не е завършило. Така например, при една операция **И,** ако стойността на първия операнд е *false*, резултатът ще бъде *false,* независимо от стойността на втория операнд, както и при операция ИЛИ, ако стойността на първия операнд е *true*, резултатът ще бъде *true*. Следователно, в тези

случаи не е необходимо да се изчислява втория оператор. Съкратените варианти на операциите **И** и **ИЛИ** изчисляват вторите операнди само ако има такава необходимост, докато пълните варианти винаги изчисляват двата операнда.

Побитови операции

Оператор	*Предназначение*	*Употреба*
~	Унарно отрицание (NOT)	~expr
<< >>	изместване наляво изместване надясно	expr1<<expr2 expr1>>expr2
>>>	Беззнаково изместване надясно	expr1>>>expr2
&	побитово И (AND)	expr1&expr2
^	изключващо ИЛИ (XOR)	expr1^expr2
|	побитово ИЛИ (OR)	expr1 | expr2

Побитовите операции действат директно върху битовете на целочислените типове **long**, **int**, **short**, **char** и **byte**. Те не могат да се използват с **boolean**, **float** или **double**, както и с типовете класове. Те се наричат побитови операции, защото се използват за проверка, задаване или изместване на битовете, които образуват целочислени стойности. Важна е разликата между операциите ‘изместване надясно’ и ‘беззнаково изместване надясно’. При положителни цели числа между тях няма разлика. Когато се използват отрицателни цели числа, в най старшия бит, обикновено се поставя ‘1’ (бит за знака). При побитовото изместване надясно, всички битове се изместват с една позиция надясно, а в най-лявата позиция се поставя ‘0’. Ако числото е отрицателно, то в позицията за знак трябва да се постави ‘0’ и числото ще се разглежда като положително. За да не се сменя знака на числото, при операцията ‘изместване надясно’ действията се извършват само върху битовете на числото, а бита за знак не участва в операцията. При операцията ‘беззнаково изместване надясно’ бита за знак участва в действията по преместване на битовете.

Примери на побитови операции:

char bits = 0151

1	0	0	1	0	1	1	1

bits = ~bits 0 1 1 0 1 0 0 0

char bits = 1

0	0	0	0	0	0	0	1

bits = bits << 1 0 0 0 0 0 0 1 0

bits = bits << 2 0 0 0 0 1 0 0 0

bits = bits >>3 0 0 0 0 0 0 0 1

char b1 = 145

0	1	1	0	0	1	0	1

char b2 = 175

1	0	1	0	1	1	1	1

result = b1&b2 0 0 1 0 0 1 0 1

result = b1^b2 1 1 0 0 1 0 1 0

result = b1 | b2 1 1 1 0 1 1 1 1

Прости и съставни оператори.

Единичните (прости) оператори в Java, представляват завършени синтактични конструкции, чрез които се извършва обработка на данни или се осъществява контрол и управление на процеса на изпълнение на програмата. Операторите в Java завършват със символа ‘;’ (точка и запетая), което осигурява свобода на запис на програмните редове. Един оператор в Java може да се запише на повече от един ред, както и няколко оператора могат да се запишат на един ред.

Освен единични, в Java съществуват и съставни оператори (блокове). Съставните оператори представляват съвкупност от оператори, които се изпълняват заедно и образуват някаква

обособена част от програмата. Те се задават, като се ограждат с логически скоби. Ролята на логически скоби в Java играят фигурните скоби ‘{‘ и ‘}’ (както е при език C++). В език Pascal тази роля се изпълнява от запазените думи **begin** и **end**. Съставните оператори (блоков) могат да влагат един в друг. Типичен пример на съставен оператор, включително и с влагане на блокове беше демонстриран по-горе в примера за обхват на променливи.

Оператор за присвояване.

Операторът за присвояване в език Java, е единичният знак за равенство ‘=’. Неговата обща форма е:

променлива = израз ; или **Var = expr1** ;

В оператора за присвояване типът на променливата (Var) трябва да е съвместим с типа на израза (expr1).

Операторът за присвояване позволява да се организира последователност (верига) от присвоявания:

```
int i, j, k ;
i = j = k = 10 ;
```

В този фрагмент на целочислените променливи `i, j, k` се присвоява стойност 10, посредством една конструкция на оператор за присвояване. Присвояването на стойностите се извършва отдясно наляво.

Java предоставя възможност за **съкратен запис** на операторите за присвояване (пренесена от C++), чрез които се опростява кодирането при някои специфични конструкции. Например, операторът

```
x = x + 10 ;
```

може да се запише в съкратен вид като:

```
x += 10 ;
```

Операторът ‘**+=**’ казва на компилатора да присвои на променливата x, стойността на x преди изпълнение на оператор и да прибави към нея стойност 10. Общият вид на съкратения формат на оператора за присвояване изглежда така:

```
Var op= expr1 ;
```

Тук е `Var` променлива, ‘**op**’ - аритметична или логическа операция, която допуска съкратен запис и `expr1` е израз с чиято стойност се променя стойността на променливата `Var`. Допустими са следните съкратени формати на оператора за присвояване:

+= -= *= /= %= &= != ^=

Съкратеният запис на оператора за присвояване, освен по-компактен запис на кода на програмата, реализират по-ефективно run-time изпълнение в машината на Java.

Преобразуване на типовете в Java.

В оператора за присвояване често се използват променливи от различен тип. Когато в един оператор за присвояване се смесват съвместими типове, стойността на израза в дясната страна автоматично се преобразува към типа на променливата в лявата страна. Така например, в следния програмен фрагмент има смесване на целочислен и реален тип данни, които са съвместими типове.

```
int ii, j ;
float x, y ;
ii = 12 ;
x = ii*2 ; //На реалната променлива x се присвоява целочислен тип
```

Тъй като променливата в лявата страна е реален тип, то стойностите на израза в дясната част се преобразуват в реален тип. Така стойността на израза `ii*2` (24) ще се преобразува в реално число (24.0) и тази стойност ще се присвои на променливата ***x.***

Когато на един тип променлива се присвоява стойност от друг тип, се извършва автоматично преобразуване на типа, ако са изпълнени следните условия: типовете да са съвместими и

типът на получателя (променливата в лявата страна) да е по-голям по размер от типът на присвояваната стойност. В този случай се извършва разширяващо преобразуване. Например, типът int, винаги е достатъчно голям (4 байта) за да съхрани стойности от типове **byte** и **short,** но не може да съхрани данни от тип **long**. Когато преобразуването на типовете не е разширяващо, то не разрешено в езика Java. Такова преобразуване се предполага в следния програмен фрагмент:

```
int ii, j ;
float x, y ;
x = 12.15 ;
ii = x*2 ; // Не е разрешено !!
```

В случая, реалната променлива ***x*** трябва да се преобразува към целочислен тип. Въпреки че в Java, **int** и **float** заемат еднаква памет (4 байта), това не е разширяващо преобразуване, тъй като при преобразуването на реална стойност в целочислена се губи част от информацията (дробната част трябва да се отреже или закръгли). Това правило, е едно от многото различия със езика C++, които определят Java, като един от най-сигурните програмни езици.

Въпреки че автоматичните преобразувания на типовете са полезни, те не могат да разрешат всички нужди на програмистите, тъй като се отнасят за малък на брой случаи на преобразуване на типовете данни. За всички останали случаи на преобразуване на типовете на данните се изискват специални инструкции към компилаторите, които уточняват начина на преобразуването (casting). Този начин на преобразуване се нарича явен и се използва специална заявка за извършването му. Общата форма на явното преобразуване на типовете изглежда по следния начин:

```
(целеви тип) израз ;
```

Тук '**целеви тип**' задава желания тип данни към който да бъде преобразувана стойността на израза зададен в горната конструкция с '**израз**'. Например, ако искаме последният оператор в горният фрагмент да бъде изпълнен, трябва да укажем явно преобразуване на израза '`x*2`' в целочислен тип и след това да се извърши присвояването:

```
ii = int (x*2) ;
```

Горното преобразуване е стесняващо преобразуване и в него се губи част от информацията. Преобразуването на `(x*2)` към целочислен тип е съпроводено с отрязване на дробната част от стойността.

В рамките на един израз също може да има смесване на типовете. Така е например в горният израз (`x*2`), където се извършва аритметична операция между реално число **`x`** и целочислена стойност **2**. За да се изпълни този израз, най-напред трябва да се извърши преобразуване на типовете на данните, така че те да станат от един тип. Правилата за преобразуване тук са, че всички променливи се преобразуват към типа носещ най-голяма информация. В случая, реалната променлива е с най-голям обем информация и целочислената стойност се преобразува в реална. Трябва да се има пред вид, че типът се преобразува само за конкретната операция, но като цяло величината си остава от типа с който е декларирана.

Условен оператор.

Пълният формат на условния оператор в Java има вида:

```
if (Log_expr) expr1 else expr2 ;
```

В тази конструкция **`Log_expr`** е условен израз (израз за сравнение или логически израз). Стойността на този израз е от типа **boolean**. Ако стойността на условния израз `L1` е **true**, ще се изпълни операторът **`expr1`** (блок на оператора **if**), а в противен случай ще се изпълни оператора **`expr2`** (блок на оператор **else**). Операторите **`expr1`** и **`expr2`** могат да бъдат или единични оператори или съставни оператори (блокове). В общият случай, операторът **`else`** не е задължителен. Когато този оператор липсва, условният оператор с непълен

формат. Изразите **expr1** и **expr2** са алтернативни възможности и затова те не могат да се изпълнят едновременно при никакви съчетания на управляващите условия.
Тъй като цялата конструкцията на условния оператор се разглежда също като оператор в Java, той може да бъде вграждан в условни оператори на мястото на **expr1** или **expr2.** В този случай се получава влагане на условен оператор в друг условен оператор. Основното правило при вложените условни оператори е че една **else** конструкция винаги се отнася за най-близката **if** конструкция, която е в рамките на същия блок и не е свързана вече с друга **else** конструкция. Това може да се види в следния пример:

```
if (ii == 10) {
    if (j<15) x = z -1 ;
    if (k>20) w = v +2 ;
     else w = v - 11 ; // Този else се отнася за (k>20)
}
else { x = z +2 ; w = 100 ; } // Този else се отнася за (ii == 10)
```

Тук в условният оператор (ii == 10) е включен съставен оператор (във фигурните скоби), в който има два вложени условни оператора. Последната конструкция **else** не се асоциира с (j<15), въпреки че е най-близкият **if** без **else,** тъй като не в същият блок (извън фигурните скоби).

Оператор switch.

Другата конструкция за избор в езика Java е операторът **switch**. Той предоставя възможност за избор от много възможности. Въпреки че с няколко условни оператори може да се реализира многовариантен избор, използването на оператора **switch** позволява създаването на по-ефикасен програмен код.
Пример за използване на оператора **switch** е показан в следващия програмен фрагмент:

```
switch (icase) {
      case 0:
         System.out.println (“ icase is zero”) ;
         break ;
      case 1:
      case 2:
      case 3:
        System.out.println (“ icase is more than 0 or less than 4“) ;
        break ;
      case 4:
      case 5:
        System.out.println (“ icase is more than 3 or less than 6”) ;
        break ;
      default:
        System.out.println (“ icase is more than 5”) ;
}
```

Представеният пример, е елементарен, но съдържа някои характерни особености, които илюстрират възможностите на оператора (цикъла) **switch**. Заглавието на оператора съдържа служебната дума **switch,** след която в скоби се записва управляващата величина **icase.** Тя може да бъде променлива или израз. Стойността на тази величина трябва да е от тип **char**, **byte**, **short** или **int**.
Тялото на оператора е съвкупност от оператори, оградени с фигурни скоби. Основни елементи в тялото на оператора са конструкциите **case**. В тялото на оператора **switch** има толкова конструкции **case**, колкото алтернативни възможности за изпълнение предлага операторът. Всеки раздел **case** има етикет, който представлява възможна стойност на управляващата величина. Етикетите за различните **case** раздели, трябва да са различни.

Всеки **case** раздел има група оператори, които се изпълняват, когато стойността на управляващата величина съвпадне със етикета. Обикновено, групата оператори за даден раздел **case,** завършва с оператора **break**. При достигането до този оператор се извършва излизане от цикъла **switch.** Ако го няма този оператор, програмата ще продължи да изпълнява операторите от следващият раздел **case.**
Тази особеност е използвана в примера, за да се организира обща група изпълними оператори за няколко възможни стойности на управляващата величина. Така за стойности на **icase** 1,2 или 3 се използва обща група изпълними оператори `System.out.println (“ icase is more than 0 or less than 4“) ; break ;.` За целта, разделите **case** с етикети 1 и 2 са оставени без изпълними оператори (празни). Когато стойността на управляващата променлива е 1 или 2, изпълнението се прехвърля към съответният **case** раздел. Тъй като в него няма оператор break, изпълнението продължава със следващите **case** раздели, докато се достигне **case** раздела с етикет 3 и се изпълнят неговите изпълними оператори.

Оператори за цикъл.
Java подържа три типа оператори за цикъл. Първата конструкция на оператор за цикъл е от типа **цикъл с предусловие**. Организира се с помощна на служебната дума **while**. Затова тази конструкция е известна и като цикъл **while**. Общата форма на цикъла е:

```
while (Log_expr) expr1 ;
```

Както и при условния оператор, **Log_expr** е логически израз (условие). Ако стойността на **Log_expr** е **true** (истина), ще се изпълни оператора `expr1` (тяло на цикъла), който може да бъде единичен оператор или блок (съставен оператор). След изпълнението на тялото на цикъла, управлението се прехвърля в началото на оператора за нова проверка на условието **Log_expr**. Когато стойността на **Log_expr** стане **false**, изпълнението на цикъла се прекратява.
Ето един пример за използване на цикъл `while:`

```
class WhileDemo {
  public static void mine(String args[]) {
    char ch ;
// Отпечатват се буквите от a до z
   ch = 'a' ;
   while (ch <= 'z') {
     System.out.println(ch) ; ch++; }
   }
}
```

Основното при цикъла `while` е, че проверката на условието за продължаване на цикъла се извършва преди изпълнението на тялото на цикъла (цикъл с предусловие). Това означава, че операторите от тялото на цикъла, могат да не се изпълнят нито веднъж (ако при влизане в цикъла **Log_expr** е **false**).
Втората конструкция е **цикъл с постусловие**. Тя е известна като цикъл **do-while**. Общата форма на цикъла има вида:

```
do expr1 while (Log_expr);
```

При изпълнението на цикъла, най-напред се изпълнява тялото на цикъла `expr1`, след което се проверява условието `Log_expr`. Ако стойността на `Log_expr` е **true**, управлението на цикъла се предава в началото за ново изпълнение на операторите от тялото на цикъла. Когато `Log_expr` е със стойност false, се извършва излизане от цикъла. Характерното за този цикъл е че условието се проверява след изпълнение на операторите от тялото (цикъл с постусловие). Това означава, че поне веднъж тялото на цикъла ще бъде изпълнено, независимо от стойността на логическото условие `Log_expr`. Следва пример за използване на цикъл **do-while:**

```
class While_doDemo {
```

```
    public static void main(String args[ ])
          throws java.io.IOExeption {
      char ch ;
// Въвеждат се символи от клавиатурата докато се натисне символ 'q'
      do {
          System.out.println(' Натисни клавиш + Enter') ;
          ch = (char) System.in.read () ; // Въвежда символ
          } while (ch <= 'q')
    }
  }
```

В този пример е използване една непозната конструкция от Java: **`throws`** `java.io.IOExeption`. Това е синтаксис на клаузата **`throws`**, отнасяща се към системата за обработка на изключенията. Изключенията са грешки, които възникват по време на изпълнение на програмите. С помощта на системата за обработка на изключенията могат да се обработват грешки по време на изпълнение на програмата по контролиран начин. Конструкцията **`throws`** се използва за деклариране на методи които генерират изключения, не обработвани от самите тях, а от извикващите ги методи. Такива изключения могат да възникнат например, при реализиране на входно-изходни операции. Тъй като входно-изходните операции не обработват изключения възникнали при въвеждане или извеждане на информацията, трябва да се укаже системата, която да обработи изключенията. Това е **IOExeption**, която се съдържа в обекта **java.io** на стандартния пакет **java.lang**.

Третата форма на оператора за цикъл е конструкцията **for**. Цикълът **for** е известен още и като цикъл управляван с променлива. Той се използва, когато е известен броят на итерациите, които трябва да се изпълнят в цикъла. Общата форма на цикъла **for** има вида:

```
for (init_expr; Log_expr; iter_expr) expr1 ;
```

Инициализиращият оператор `init_expr` се използва за задаване на начална стойност на управляващата променлива в цикъла, която работи като контролиращ брояч на цикъла. `Log_expr` е логически израз (условие), който определя дали да се изпълнява тялото на цикъла `expr1`. Както и в другите случаи, тялото на цикъла `expr1` може да бъде единичен оператор или блок. Изразът за итерация се използва да се извършва актуализация на управляващата променлива (променя се стойността на брояча). Актуализацията се извършва след проверка на условието и изпълнение на тялото на цикъла. Използването на елементите на цикъла **for** може да се види в следващият елементарен пример:

```
Class Decrease_For {
  public static void main(String args[]) {
     int x ;
     for (x = 100 ; x >= 0 ; x -= 5)
        System.out.println(x) ;
     }
}
```

Управляваща променлива на цикъла е ***x***, а началната ѝ стойност се установява от инициализиращия оператор и тя е 100. Цикълът се изпълнява, докато стойността на управляващата променлива е по-голяма или равна на 0. Актуализацията на променливата се извършва от оператора `x -= 5`, който представлява съкратен запис на операцията изваждане на стойност 5 от стойността на променливата. Тялото на цикъла е единичен оператор и затова не се изискват фигурни скоби.

Характерното за цикъла **for** в Java е, че той е наследил голяма част от свободата на записа от езика C++. Ще отбележим някои от вариациите, които се допускат при използването на цикъла.

Операторът **for** може да използва повече от една управляваща променлива. Това означава, че в инициализиращият израз и в израза за итерация могат да се съдържат повече от един оператори. Ето един допустим формат на оператора за цикъл **for**:

```
for (ii=1, j=100, k=0; ii < j ; ii++, j--) {…
```

Тук в инициализиращата част има три оператора за присвояване, разделени със запетаи, а в израза за итерации - две операции. Оператора к=0, няма отношение към управлението на цикъла и мястото му по принцип не би трябвало да е там, но синтаксиса на езика го допуска.
Друга допустима разновидност на цикъла **for** е възможността условният оператор да не съдържа условие за управляващата променлива. То може да бъде произволен логически израз. Освен това, елементите в заглавието на цикъла не са задължителни. Може да се организира цикъл без инициализиращ оператор:

```
for (; ii < 100 ; ii++) {…
```

В този случай началната стойност на управляващата променлива трябва да се зададе извън цикъла, преди влизането в него. По същият начин, може да се запише цикъл **for** без израз за итерации или без логически израз (условие). За да работи правилно такъв цикъл, е необходимо актуализацията на управляващите променливи или изхода от цикъла да се организира в тялото на цикъла. Както и в език C++ и в Java е допустим оператор за цикъл без нито един от управляващите елементи на цикъла:

```
for (;;) {…
```

В този случай всички действия по управлението на цикъла трябва да се извършват в тялото на цикъла или преди началото на изпълнението му. Така може да се организира безкраен цикъл, ако няма проверка за завършване на цикъла.
В Java тялото на един цикъл може да бъде празно. Понякога това е полезна конструкция. Ето един пример за използване на празен оператор в цикъл **for**:

```
class Empty_For {
    public static void main (String args[]) {
     int ii ;
     int sum = 0 ;
     for (ii = 1 ; ii <= 10; sum += ii++) ;
     System.out.println (" Sum is :",sum) ;
     }
}
```

В този пример се извършва сумиране на целите числа от 1 до 10, но всички действия се извършват изцяло в управляващата част на оператора за цикъл. Тялото на цикъла е празно (веднага след заглавието на цикъла има ‘;’). Следващият оператор “`System.out.println (" Sum is :",sum) ;`” е извън цикъла. Сумирането в случая се извършва в израза за итерации `sum += ii++`, който представлява съкратен запис на сумиране (прибавяне) към променливата ***sum*** и същевременно инкрементиране на променливата ***ii***.

Управляващи оператори.

Допълнителни възможности за управление и изход от конструкциите за цикъл могат да се реализират с помощта на операторите **break (прекъсни)** и **continue (продължи)**. Първият от тези оператори осигурява цялостно напускане на цикъла, а вторият реализира преустановяване на текущата итерация на цикъла и предава управлението в началото на цикъла за следваща итерация. Използвайки оператор **break**, примерът за въвеждане на символи, вместо с цикъл **do-while**, може да се изпълни с безкраен цикъл for:

```
class BreakDemo {
  public static void mine(String args[])
    throws java.io.IOExeption {
    char ch ;
    for (;;) {
     System.out.println(‘ Натисни клавиш + Enter’) ;
     ch = (char) System.in.read () ; // Въвежда символ
     if (ch == ‘q’) break ;
```

```
        }
    }
}
```

Операторът **break** може да се използва и като ‘интелигентен’ заместител на оператора **goto**, който се използва в езика C++. Програмите, които използват **goto,** обикновено са трудни за разбиране и управление. Поради тази причина той се избягва от употреба, а в Java е премахнат. Съществуват обаче ситуации, при които използването на оператор подобен на **goto** са доста полезни. Така е например, когато трябва да се напусне блок намиращ се във вложени блокови структури. За тази цел в Java се използва оператор **break** работещ с етикет. Ето как изглежда синтактичната конструкция при използване на **break** с етикет:

```
class Break_Lab {
   public static void main(String args[]) {
       int ii ;
       for (ii = 1; ii < 4 ; ii++) {
lab1:  {
lab2:    {
lab3:       {
            if (ii == 1) break lab1 ;
            if (ii == 2) break lab2 ;
            if (ii == 3) break lab3 ;
            }
             System.out.println (“След блок lab3”) ;
          }
         System.out.println (“След блок lab2”) ;
         }
      System.out.println(“След блок lab1”) ;
      }
  }
```

В примера има три вложени един в друг блока и оператора **break** се намира в най-вътрешния блок. За да се осигури възможност за напускане на блоковете трябва да се укаже кой блок да се напусне - най-вътрешния, средния или най-външния. За тази цел отделните блокове се означават с етикети (**lab1, lab2, lab3**) и в оператора **break** се указва кой блок се напуска. Тук стават ясни и различията между оператора goto в C++ и **break** с етикет. С оператора **break** не може да се прехвърли управлението в произволно място в програмата (както е при goto), а само се избира блока които се напуска. Това означава, че оператора **break** трябва да се намира в блока за който се отнася (не непременно най-вътрешния). Етикетите могат да именоват не произволен оператор, а само блокове в програмите.

Изпълнението на дадена итерация в цикъл може да бъде форсирано посредством оператора **continue (продължи)**. Конструкцията **continue** ускорява изпълнението на определена итерация от цикъла, като пропуска кода до края на тялото на цикъла и предава управлението в началото на цикъла за изпълнение на следващата итерация.

```
class ContinueDemo {
   pulic static void main (String args[]) {
        int ii ;
// отпечатва четните числа между 0 и 100
       for (ii = 0 ; ii <= 100 ; ii++) {
            if (ii%2 != 0 ) continue ;
     System.out.println(ii) ;  }
   }}
```

Оператор ?

Един от най-интересните оператори в Java, пренесени от език C++ е операторът **?**. Той се използва за съкратен запис на някои специфични форми на условният оператор. Ако се налага използването на условен оператор от типа:

```
if (Log_expr) var1 = expr1
   else var1 = expr2 ;
```

той може да се замени с оператор от вида:

```
var1 = Log_expr ? expr1 : expr2 ;
```

Общата форма на оператора **?** има вида: `Log_expr ? expr1 : expr2.` Операторът **?** се нарича тернарен, защото изисква три операнда. При него `Log_expr` е логически израз със стойност **true** или **false**. Другите аргументи в оператора, `expr1` и `expr2` са произволни изрази от един и същ тип. Стойността на оператора е равна на стойността на `expr1`, ако `Log_expr` е **true** или на стойността на `expr2`, ако ако `Log_expr` е **false.** Ето един пример за използване на оператора **?**:

```
absval = val > 0 ? val : -val ;
```

Тук на променливата `absval` се присвоява абсолютната стойност на променливата `val`.