

Глава 5. Управляющие конструкции

Составной оператор

Согласно синтаксису языка Java во многих конструкциях может стоять только один оператор, но часто встречается ситуация, когда надо использовать последовательность из нескольких операторов.

Составной оператор — блок кода между фигурными скобками {}:

Имеется два общепринятых способа форматирования текста с использованием фигурных скобок.

В первом из них скобки пишут друг под другом, а текст, находящийся между ними, сдвигают на 1-2 символа вправо (изредка – больше). Пример:

```
оператор
{
    последовательность простых или составных операторов
}
```

Во втором, более распространённом, открывающую фигурную скобку пишут на той же строке, где должен начинаться составной оператор, без переноса его на следующую строку. А закрывающую скобку пишут под первым словом. Пример:

```
оператор{
    последовательность простых или составных операторов
}
```

Именно такой способ установлен по умолчанию в среде NetBeans, и именно он используется в приводимых в данной книге фрагментах программного кода. Тем не менее, автор предпочитает первый способ форматирования программного кода, так как он более читаемый. Для установки такого способа форматирования исходного кода следует перейти в меню Tools/Options, выбрать Editor/Indentation, после чего в появившейся диалоговой форме отметить галочкой пункт Add New Line Before Brace.

После фигурных скобок по правилам Java, как и в /C++, ставить символ “;” не надо. Но его можно ставить после фигурных скобок в тех местах программы, где разрешается ставить ничего не делающий пустой оператор “;” .

Условный оператор if

У условного оператора if имеется две формы: if и if- else.

По-английски if означает “если”, else - “в противном случае”. Таким образом, эти операторы могут быть переведены как “если...то...” и “если...то...в противном случае...”.

Первая форма:

```
if (условие)
    оператор1;
```

Если условие равно true, выполняется оператор1. Если же условие==false, в операторе не выполняется никаких действий.

Вторая форма:

```
if (условие)
    оператор1;
else
    оператор2;
```

В этом варианте оператора `if` если `условие==false`, то выполняется оператор2.

Обратите особое внимание на форматирование текста. Не располагайте все части оператора `if` на одной строке – это характерно для новичков!

Пример:

```
if(a<b)
    a=a+1;
else if(a==b)
    a=a+1;
else{
    a=a+1;
    b=b+1;
};
```

Из этого правила имеется исключение: если подряд идёт большое количество операторов `if`, уместяющихся в одну строку, для повышения читаемости программ бывает целесообразно не переносить другие части операторов на отдельные строки.

Надо отметить, что в операторе `if` в области выполнения, которая следует после условия, а также в области `else`, должен стоять только один оператор, а не последовательность операторов. Поэтому запись оператора в виде

```
if(условие)
    оператор1;
    оператор2;
else
    оператор3;
```

недопустима. В таких случаях применяют составной оператор, ограниченный фигурными скобками. Между ними, как мы знаем, может стоять произвольное число операторов:

```
if(условие) {
    оператор1;
    оператор2;
}
else
    оператор3;
```

Если же мы напишем

```
if(условие)
    оператор1;
else
    оператор2;
    оператор3;
```

- никакой диагностики ошибки компилятор не выдаст! Оператор3 в этом случае никакого отношения к условию `else` иметь не будет – подобное форматирование текста будет подталкивать к логической ошибке. При следующем форматировании текста программы, эквивалентному предыдущему при компиляции, уже более очевидно, что оператор3 не относится к части `else`:

```
if(условие)
    оператор1;
else
```

```
    оператор2;  
оператор3;
```

Для того, чтобы оператор3 относился к части else, следует использовать составной оператор:

```
if (условие)  
    оператор1;  
else {  
    оператор2;  
    оператор3;  
};
```

В случае последовательности операторов типа:

```
if (условие1) if (условие2) оператор1 else оператор2;
```

имеющийся else относится к последнему if, поэтому лучше отформатировать текст так:

```
if (условие1)  
    if (условие2)  
        оператор1;  
else  
    оператор2;
```

Таким образом, если писать соответствующие if и else друг под другом, логика работы программы становится очевидной.

Пример **неправильного** стиля оформления:

```
if (условие1)  
    if (условие2)  
        оператор1;  
else  
    оператор2;
```

Этот стиль подталкивает к **логической ошибке** при чтении программы. Человек, читающий такую программу (как и тот, кто её писал), будет полагать, что оператор2 выполняется, если условие1==false, так как кажется, что он относится к первому if, а не ко второму. Надо отметить, что сама возможность такой ошибки связана с непродуманностью синтаксиса языка Java. Для правильной работы требуется переписать этот фрагмент в виде

```
if (условие1) {  
    if (условие2)  
        оператор1;  
};  
else оператор2;
```

Чтобы избежать такого рода проблем, используйте опцию Reformat code (“переформатировать код”) из всплывающего меню, вызываемого в исходном коде щелчком правой кнопки мыши.

Типичной является ситуация с забытым объединением последовательности операторов в составной с помощью фигурных скобок. Например, пишут

```
if (условие)  
    оператор1;
```

```
оператор2;
```

вместо

```
if (условие)
{ оператор1;
  оператор2;
};
```

Причём такую ошибку время от времени допускают даже опытные программисты. Reformat code не помогает – обычно эта команда вызывается после того, как программист далеко ушёл от проблемного места, и перенос оператора под начало “if” не замечает. Для того, чтобы гарантированно избежать такой ошибки, следует ВСЕГДА ставить фигурные скобки в тех местах операторов Java, где по синтаксису может стоять только один оператор. Конечно, было бы лучше, чтобы разработчики Java предусмотрели такую конструкцию как часть синтаксиса языка. Возможно, в среде NetBeans в дальнейшем будет сделана возможность в опциях редактора делать установку, позволяющую автоматически создавать эти скобки. Точно так же, как автоматически создаётся закрывающая круглая скобка в операторе if после того как вы набираете if(.

Следует отметить разные правила использования точки с запятой при наличии части else в случае использования фигурных скобок и без них:

```
if (i==0)
    i++;
else
    i--;
```

Во втором случае после скобок (перед else) точка с запятой не ставится:

```
if (i==0) {
    i++;
}
else{
    i--;
};
```

Неприятной проблемой, унаследованной в Java от языка C, является возможность использования оператора присваивания “=” вместо оператора сравнения “==”. Например, если мы используем булевы переменные b1 и b2, и вместо

```
if (b1==b2)
    i=1;
else
    i=2;
```

напишем

```
if (b1=b2)
    i=1;
else
    i=2;
```

никакой диагностики ошибки выдано не будет. Дело в том, что по правилам C и Java любое присваивание рассматривается как функция, возвращающая в качестве результата присваиваемое значение. Поэтому присваивание b1=b2 возвратит значение, хранящееся в переменной b2. В результате оператор будет работать, но совсем не так, как ожидалось. Более того, будет испорчено значение, хранящееся в переменной b1.

Проблемы с ошибочным написанием “=” вместо “==” в Java гораздо менее типичны, чем в C/C++, поскольку они возникают только при сравнении булевой переменной с булевым значением. Если же оператор “=” ставится вместо “==” при сравнении числовой переменной, происходит диагностика ошибки, так как такое присваивание должно

возвращать число, а не булевское значение. Жёсткая типизация Java обеспечивает повышение надёжности программ.

Оператор выбора `switch`

Является аналогом `if` для нескольких условий выбора. Синтаксис оператора следующий:

```
switch (выражение) {  
    case значение1: операторы1;  
    .....  
    case значениеN: операторы N;  
    default: операторы;  
}
```

Правда, крайне неудобно, что нельзя ни указывать диапазон значений, ни перечислять через запятую значения, которым соответствуют одинаковые операторы.

Тип выражения должен быть каким-нибудь из целых типов. В частности, недопустимы вещественные типы.

Работает оператор следующим образом: сначала вычисляется выражение. Затем вычисленное значение сравнивается со значениями вариантов, которые должны быть определены ещё на этапе компиляции программы. Если найден вариант, которому удовлетворяет значение выражения, выполняется соответствующий этому варианту последовательность операторов, после чего НЕ ПРОИСХОДИТ выхода из оператора `case`, что было бы естественно. - Для такого выхода надо поставить оператор `break`. Эта неприятная особенность Java унаследована от языка C.

Часть с `default` является необязательной и выполняется, если ни один вариант не найден.

Пример:

```
switch (i/j) {  
    case 1:  
        i=0;  
        break;  
    case 2:  
        i=2;  
        break;  
    case 10:  
        i=3;  
        j=j/10;  
        break;  
    default:  
        i=4;  
};
```

У оператора `switch` имеется две особенности:

- Можно писать произвольное число операторов для каждого варианта `case`, что весьма удобно, но полностью выпадает из логики операторов языка Java.
- Выход из выполнения последовательности операторов осуществляется с помощью оператора `break`. Если он отсутствует, происходит “проваливание” в блок операторов, соответствующих следующему варианту за тем, с которым совпало значение выражения. При этом никакой проверки соответствия очередному значению не производится. И так продолжается до тех пор, пока не встретится оператор `break` или не кончатся все операторы в вариантах выбора. Такие правила проверки порождают типичную ошибку,

называемую “забытый break”.

Условное выражение ...?... : ...

Эта не очень удачная по синтаксису функция унаследована из языка C. Её синтаксис таков:
условие?значение1 : значение2

В случае, когда условие имеет значение true, функция возвращает значение1, в противном случае возвращается значение2.

Например, мы хотим присвоить переменной j значение, равное i+1 при i<5, и i+2 в других случаях. Это можно сделать таким образом:

```
j=i<5?i+1:i+2
```

Иногда при вычислении громоздких выражений этот оператор приходится использовать: без него программа оказывается ещё менее прозрачной, чем с ним. Приоритет разделителей “?” и “:” очень низкий – ниже только приоритет оператора присваивания (в любых его формах). Поэтому можно писать выражения без использования скобок. Но лучше всё-таки использовать скобки:

```
j=(i<5)?(i+1):(i+2)
```

Операторы инкремента ++ и декремента --

Оператор “++” называется инкрементным (“увеличивающим”), а “--” декрементным (“уменьшающим”). У этих операторов имеются две формы, постфиксная (наиболее распространённая, когда оператор ставится после операнда) и префиксная (используется очень редко, в ней оператор ставится перед операндом).

Для любой числовой величины x выражение x++ или ++x означает увеличение x на 1, а выражение x-- или --x означает уменьшение x на 1.

Различие двух форм связано с тем, когда происходит изменение величины – после вычисления выражения, в котором используется оператор, для постфиксной формы, или до этого вычисления – для префиксной.

Например, присваивания $j=i++$ и $j=++i$ дадут разные результаты. Если первоначально $i=0$, то первое присваивание даст 0, так как i увеличится на 1 *после* выполнения присваивания. А второе даст 1, так как *сначала* выполнится инкремент, и только потом будет вычисляться выражение и выполняться присваивание. При этом в обоих случаях после выполнения присваивания i станет равно 1.

Оператор цикла for

```
for (блок инициализации; условие выполнения тела цикла;  
     блок изменения счётчиков)  
оператор;
```

В блоке инициализации через запятую перечисляются операторы задания локальных переменных, область существования которых ограничивается оператором for. Также могут быть присвоены значения переменным, заданным вне цикла. Но инициализация может происходить только для переменных одного типа.

В блоке условия продолжения цикла проверяется выполнение условия, и если оно выполняется, идёт выполнение тела цикла, в качестве которого выступает оператор. Если же не выполняется – цикл прекращается, и идёт переход к оператору программы, следующему за оператором for.

После каждого выполнения тела цикла (очередного шага цикла) выполняются операторы блока изменения счётчиков. Они должны разделяться запятыми.

Пример:

```
for(int i=1,j=5; i+j<100; i++,j=i+2*j){  
    ...  
};
```

Каждый из блоков оператора for является необязательным, но при этом разделительные “;” требуется писать.

Наиболее употребительное использование оператора for – для перебора значений некоторой переменной, увеличивающихся или уменьшающихся на 1, и выполнения последовательности операторов, использующих эти значения. Переменная называется счетчиком цикла, а последовательности операторов – телом цикла.

Пример1: вычисление суммы последовательно идущих чисел.

Напишем цикл, в котором производится суммирование всех чисел от 1 до 100. Результат будем хранить в переменной result.

```
int result=0;  
for(int i=1; i<=100; i++){  
    result=result+i;  
};
```

Цикл (повторное выполнение одних и тех же действий) выполняется следующим образом:

- До начала цикла создается переменная result, в которой мы будем хранить результат. Одновременно выполняется инициализация – присваивается начальное значение 0.
- Начинается цикл. Сначала выполняется блок инициализации - счётчику цикла i присваивается значение 1. Блок инициализации выполняется только один раз в самом начале цикла.
- Начинается первый шаг цикла. Проверяется условие выполнения цикла. Значение i сравнивается со 100.
- Поскольку сравнение $1 \leq 100$ возвращает true, выполняется тело цикла. В переменной result хранится 0, а значение i равно 1, поэтому присваивание $result=result+i$ эквивалентно $result=1$. Таким образом, после первого шага цикла в переменной result будет храниться значение 1.
- После выполнения тела цикла выполняется секция изменения счётчика цикла, то есть оператор i++, увеличивающий i на 1. Значение i становится равным 2.
- Начинается второй шаг цикла. Проверяется условие выполнения тела цикла. Поскольку сравнение $2 \leq 100$ возвращает true, идёт очередное выполнение тела цикла, а затем – увеличение счётчика цикла.
- Шаги цикла продолжают до тех пор, пока счётчик цикла не станет равным 101. В этом случае условие выполнения тела цикла $101 \leq 100$ возвращает false, и происходит выход из цикла. Последнее присваивание $result=result+i$, проведённое в цикле, это $result=result+100$.

Если бы нам надо было просуммировать числа от 55 до 1234, в блоке инициализации i надо присвоить 55, а в условии проверки поставить 1234 вместо 100.

Пример 2: вычисление факториала.

```
double x=1;
```

```

for (i=1; i<=n; i++) {
    x=x*i;
};

```

Заметим, что в приведённых примерах можно сделать некоторые усовершенствования - операторы присваивания записать следующим образом:

```

result+=i; вместо result=result+i;

```

для первого примера и

```

x*=i; вместо x=x*i;

```

для второго. На начальных стадиях обучения так лучше не делать, поскольку текст программы должен быть понятен программисту – все алгоритмы должны быть “прозрачны” для понимания.

Наиболее распространённая ошибка при работе с циклами, в том числе – с циклом `for` - использование вещественного счётчика цикла. Разберём эту ошибку на примере.

Пример 3. Вычисление площади под кривой.

Пусть надо вычислить площадь S под кривой, задаваемой функцией $f(x)$, на промежутке от a до b (провести интегрирование). Разобьём промежуток на n интервалов, при этом длина каждого интервала будет $h=(b-a)/n$. Мы предполагаем, что величины a, b, n и функция $f(x)$ заданы. Площадь под кривой на интервале с номером j будем считать равной значению функции на левом конце интервала. Такой метод численного нахождения интеграла называется методом левых прямоугольников.

На первый взгляд можно записать алгоритм вычисления этой площади в следующем виде:

```

double S=0;
double h=(b-a)/n;
for (double x=a; x<b; x=x+h) {
    S=S+f(x)*h;
};

```

И действительно, ИНОГДА такой алгоритм правильно работает. Но изменение числа интервалов n или границ a или b может привести к тому, что будет учтён лишний интервал, находящийся справа от точки b . Это связано с тем, что в циклах с вещественным счётчиком ВСЕГДА проявляется нестабильность, если последняя точка попадает на границу интервала. Что будет происходить на последних шагах нашего цикла?

На предпоследнем шаге мы попадаем в точку $x=b-h$, при этом условие $x<b$ всегда выполняется, и никаких проблем не возникает. Следующей точкой должна быть $x=b$. Проверяется условие $x<b$, и в идеальном случае должен происходить выход из цикла, поскольку $x=b$, и условие не должно выполняться. Но ведь все операции в компьютере для чисел в формате с плавающей точкой проводятся с конечной точностью. Поэтому практически всегда значение x для этого шага будет либо чуть меньше b , либо чуть больше. Отличие будет в последних битах мантиссы, но этого окажется достаточно для того, чтобы сравнение $x<b$ иногда давало `true`. Хотя для заданных a, b и n результат будет прекрасным образом воспроизводиться, в том числе – на других компьютерах. Более того, при увеличении числа разбиений n погрешность вычисления площади даже для “неправильного” варианта будет убывать, хотя и гораздо медленнее, чем для “правильного”. Это один из самых неприятных типов ошибок, когда алгоритм вроде бы работает правильно, но для получения нужной точности требуется гораздо больше времени или ресурсов.

Рассмотрим теперь правильную реализацию алгоритма. Для этого будем использовать ЦЕЛОЧИСЛЕННЫЙ счётчик цикла. Нам потребуется чуть больше рассуждений, и алгоритм окажется немного менее прозрачным, но зато гарантируется его устойчивость. Значение функции в начале интервала с номером j будет равно $f(a+j*h)$. Первый интервал будет иметь

номер 0, второй – номер 1, и так далее. Последний интервал будет иметь номер n-1.

Правильно работающий алгоритм может выглядеть так:

```
double S=0;
double h=(b-a)/n;
for(int j=0;j<=n-1;j++){
    S=S+f(a+j*h)*h;
};
```

Проверить неустойчивость первого алгоритма и устойчивость второго можно на примере $f(x)=x^2$, $a=0$, $b=1$.

n	S для первого алгоритма	S для второго алгоритма
8	0.2734375	0.2734375
9	0.2798353909465021	0.279835390946502
10	0.3849999999999999	0.28500000000000001
11	0.28925619834710753	0.28925619834710750
12	0.29282407407407407	0.292824074074074
13	0.3727810650887573	0.2958579881656805
...
100	0.328350000000000036	0.32835000000000014
101	0.32839917655131895	0.3283991765513185
102	0.33825131359733385	0.3284473920287069
103	0.3382034121971908	0.3284946743331133
...
1000	0.332833500000000095	0.33283350000000034
1001	0.3338330001666631	0.33283399916766554
1002	0.3328344973393309	0.33283449733932

В таблице жирным выделены первые значащие цифры неправильных значений, получающихся в результате неустойчивости алгоритма при $n=10$, $n=13$, $n=102$, $n=103$, $n=1001$. При отсутствии неустойчивости оба алгоритма при всех n должны были бы давать одинаковые результаты (с точностью до нескольких младших бит мантиссы). Очень характерной особенностью такого рода неустойчивостей является скачкообразное изменение результата при плавном изменении какого-либо параметра. В приведённой выше таблице меняется число n , но такая же ситуация будет наблюдаться и при плавном изменении чисел a или b . Например, при $n=10$ и $a=0$ получим следующие результаты в случае очень малых изменений b :

b	S для первого алгоритма	S для второго алгоритма
1.00000	0.3849999999999999	0.28500000000000001
1.00001	0.28500855008550036	0.28500855008550036
1.00002	0.2850171003420023	0.2850171003420022
1.00003	0.38503465103951023	0.28502565076950764
1.00004	0.2850342013680182	0.2850342013680184
1.00005	0.2850427521375357	0.2850427521375357

Вещественный счётчик цикла не всегда приводит к проблемам. Рассмотрим вариант численного нахождения интеграла методом средних прямоугольников. Для этого площадь под кривой на интервале с номером j будем считать равной значению функции в середине интервала. Алгоритм идентичен описанному выше для метода левых прямоугольников, за исключением выбора в качестве начальной точки $a+h/2$ вместо a .

```
double S=0;
h=(b-a)/n;
for(double x=a+h/2;x<b;x=x+h){
    S=S+f(x)*h;
};
```

Для данного алгоритма проблем не возникает благодаря тому, что все точки x , в которых производится сравнение $x<b$, отстоят достаточно далеко от значения b – по крайней мере на $h/2$. Заметим, что метод средних прямоугольников гораздо точнее метода левых прямоугольников, и в реальных вычислениях лучше использовать либо его, либо метод Симпсона, который несколько сложнее, но обычно ещё более точен.

Отметим ещё один момент, важный для эффективной организации циклов. Предложенные выше реализации циклов не самые эффективные по скорости, поскольку в них много раз повторяется достаточно медленная операция – “вещественное” умножение (умножение в формате с плавающей точкой). Лучше написать алгоритм метода средних прямоугольников так:

```
double S=0;
double h=(b-a)/n;
for(double x=a+h/2;x<b;x=x+h){
    S=S+f(x);
};
S=S*h;
```

При этом суммируются значения $f(x)$, а не $f(x)*h$, а умножение делается только один раз – уже после выхода из цикла для всей получившейся суммы. Если время выполнения “вещественного” умножения $t_{\text{умнож}}$, то экономия времени работы программы по сравнению с первоначальным вариантом будет $(n-1)*t_{\text{умнож}}$. Для больших значений n экономия времени может быть существенной даже для мощных компьютеров.

Такого рода действия очень характерны при написании программ с использованием циклов, и особенно важны при большом количестве шагов: следует выносить из циклов все операции, которые могут быть проделаны однократно вне цикла. Но при усовершенствованиях часто теряется прозрачность алгоритмов. Поэтому полезно сначала написать реализацию алгоритма “один к одному” по формулам, без всяких усовершенствований, и убедиться при не очень большом числе шагов, что всё работает правильно. А уже затем можно вносить исправления, повышающие скорость работы программы в наиболее критических местах. Не следует сразу пытаться написать программу, которая максимально эффективна по всем параметрам. Это обычно приводит к гораздо более длительному процессу поиска неочевидных ошибок в такой программе.

Замечание: В Java отсутствует специальная форма оператора `for` для перебора в цикле элементов массивов и коллекций (или, что то же, наборов). Тем не менее оператор `for` позволяет последовательно обработать все элементы массива или набора. Пример поочерёдного вывода диалогов со значениями свойств компонентов, являющихся элементами массива компонентов главной формы приложения:

```
java.util.List components= java.util.Arrays.asList(this.getComponents());
```

```

for (Iterator iter = components.iterator();iter.hasNext();) {
    Object elem = (Object) iter.next();
    javax.swing.JOptionPane.showMessageDialog(null,"Компонент: "+
                                                elem.toString());
}

```

Оператор цикла *while* – цикл с предусловием

```

while (условие)
    оператор;

```

Пока условие сохраняет значение `true` — в цикле выполняется оператор, иначе — действие цикла прекращается. Если условие с самого начала `false`, цикл сразу прекращается, и тело цикла не выполнится ни разу.

Цикл `while` обычно применяют вместо цикла `for` в том случае, если условия продолжения достаточно сложные. В отличие от цикла `for` в этом случае нет формально заданного счётчика цикла, и не производится его автоматического изменения. За это отвечает программист. Хотя вполне возможно использование как цикла `for` вместо `while`, так и наоборот. Многие программисты предпочитают пользоваться только циклом `for` как наиболее универсальным.

Пример:

```

i=1;
x=0;
while (i<=n) {
    x+=i;//эквивалентно x=x+i;
    i*=2;//эквивалентно i=2*i;
};

```

В операторе `while` очень часто совершают ошибки, приводящие к неустойчивости алгоритмов из-за сравнения чисел с плавающей точкой на неравенство. Как мы знаем, сравнивать их на равенство в подавляющем большинстве случаев некорректно из-за ошибок представления таких чисел в компьютере. Но большинство программистов почему-то считает, что при сравнении на неравенство проблем не возникает, хотя это не так. Например, если организовать с помощью оператора `while` цикл с вещественным счётчиком, аналогичный разобранный в разделе, посвящённому циклу `for`. Пример типичной ошибки в организации такого цикла приведён ниже:

```

double a=...;
double b=...;
double dx=...;
double x=a;
while (x<=b) {
    ...
    x=x+dx;
};

```

Как мы уже знаем, данный цикл будет обладать неустойчивостью в случае, когда на интервале от `a` до `b` укладывается целое число шагов. Например, при `a=0`, `b=10`, `dx=0.1` тело цикла будет выполняться при `x=0`, `x=0.1`, ..., `x=9.9`. А вот при `x=10` тело цикла может либо выполниться, либо не выполниться – как повезёт! Причина связана с конечной точностью выполнения операций с числами в формате с плавающей точкой. Величина шага `dx` в двоичном представлении чуть-чуть отличается от значения `0.1`, и при каждом цикле

систематическая погрешность в значении x накапливается. Поэтому точное значение $x=10$ достигнуто не будет, величина x будет либо чуть-чуть меньше, либо чуть-чуть больше. В первом случае тело цикла выполнится, во втором – нет. То есть пройдёт либо 100, либо 101 итерация (число выполнений тела цикла).

Оператор цикла `do...while` – цикл с постусловием

```
do
    оператор;
while (условие);
```

Если условие принимает значение `false`, цикл прекращается. Тело цикла выполняется до проверки условия, поэтому оно всегда выполнится хотя бы один раз.

Пример:

```
int i=0;
double x=1;
do{
    i++; // i=i+1;
    x*=i; // x=x*i;
}
while (i<n);
```

Если с помощью оператора `do...while` организуется цикл с вещественным счётчиком или другой проверкой на равенство или неравенство чисел типа `float` или `double`, у него возникают точно такие же проблемы, как описанные для циклов `for` и `while`.

При необходимости организовать бесконечный цикл (с выходом изнутри тела цикла с помощью оператора прерывания) часто используют следующий вариант:

```
do{
    ...
}
while (false);
```

Операторы прерывания `continue`, `break`, `return`, `System.exit`

Довольно часто требуется при выполнении какого-либо условия прервать цикл или подпрограмму и перейти к выполнению другого алгоритма или очередной итерации цикла. При неструктурном программировании для этих целей служил оператор `goto`. В Java имеются более гибкие и структурные средства для решения этих проблем - операторы `continue`, `break`, `return`, `System.exit`:

`continue`; – прерывание выполнения тела цикла и переход к следующей итерации (проверке условия) текущего цикла;

`continue имя метки`; – прерывание выполнения тела цикла и переход к следующей итерации (проверке условия) цикла, помеченного меткой (`label`);

`break`; – выход из текущего цикла;

`break имя метки`; – выход из цикла, помеченного меткой;

`return`; – выход из текущей подпрограммы (в том числе из тела цикла) без возврата значения;

`return значение;` – выход из текущей подпрограммы (в том числе из тела цикла) с возвратом значения;

`System.exit(n)` – выход из приложения с кодом завершения `n`. Целое число `n` произвольно задаётся программистом. Если `n=0`, выход считается нормальным, в других случаях - аварийным. Приложение перед завершением сообщает число `n` операционной системе для того, чтобы программист мог установить, по какой причине произошёл аварийный выход.

Операторы `continue` и `break` используются в двух вариантах – без меток для выхода из текущего (самого внутреннего по вложенности) цикла, и с меткой - для выхода из помеченного ей цикла. Меткой является идентификатор, после которого стоит двоеточие. Метку можно ставить непосредственно перед ключевым словом, начинающим задание цикла (`for`, `while`, `do`).

Пример использования `continue` без метки:

```
for(int i=1;i<=10;i++){
    if(i==(i/2)*2){
        continue;
    };
    System.out.println("i="+i);
};
```

В данном цикле не будут печататься все значения `i`, для которых `i==(i/2)*2`. То есть выводиться в окно консоли будут только нечётные значения `i`.

Ещё один пример использования `continue` без метки:

```
for(int i=1;i<=20;i++){
for(int j=1;j<=20;j++){
    if(i*j==(i*j/2)*2){
        continue;
    };
    System.out.println("i="+i+" j="+j+ " 1.0/(i*j-20)="+ (1.0/(i*j-20)) );
};
};
```

В этом случае будут выводиться значения `i`, `j` и `1.0/(i*j-20)` для всех нечётных `i` и `j` от 1 до 19. То есть будут пропущены значения для всех чётных `i` и `j`:

```
i=1 j=1 1.0/(i*j-20)=-0.05263157894736842
i=1 j=3 1.0/(i*j-20)=-0.058823529411764705
i=1 j=5 1.0/(i*j-20)=-0.066666666666666667
i=1 j=7 1.0/(i*j-20)=-0.07692307692307693
i=1 j=9 1.0/(i*j-20)=-0.09090909090909091
i=1 j=11 1.0/(i*j-20)=-0.11111111111111111
i=1 j=13 1.0/(i*j-20)=-0.14285714285714285
```

```

i=1 j=15 1.0/(i*j-20)=-0.2
i=1 j=17 1.0/(i*j-20)=-0.3333333333333333
i=1 j=19 1.0/(i*j-20)=-1.0
i=3 j=1 1.0/(i*j-20)=-0.058823529411764705
i=3 j=3 1.0/(i*j-20)=-0.09090909090909091
i=3 j=5 1.0/(i*j-20)=-0.2
i=3 j=7 1.0/(i*j-20)=1.0
...
i=19 j=9 1.0/(i*j-20)=0.006622516556291391
i=19 j=11 1.0/(i*j-20)=0.005291005291005291
i=19 j=13 1.0/(i*j-20)=0.004405286343612335
i=19 j=15 1.0/(i*j-20)=0.0037735849056603774
i=19 j=17 1.0/(i*j-20)=0.0033003300330033004
i=19 j=19 1.0/(i*j-20)=0.002932551319648094

```

Пример использования continue с меткой:

```

label_for1:
    for(int i=1;i<=20;i++){
        for(int j=1;j<=20;j++){
            if(i*j==(i*j/2)*2){
                continue label_for1;
            };
            System.out.println("i="+i+" j="+j+ " 1.0/(i*j-20)="+ (1.0/(i*j-20)) );
        };
    };

```

В отличие от предыдущего случая, после каждого достижения равенства $i*j==(i*j/2)*2$ будет производиться выход из внутреннего цикла (по j), и все последующие j для таких значений i будут пропущены. Поэтому будут выведены только значения

```

i=1 j=1 1.0/(i*j-20)=-0.05263157894736842
i=3 j=1 1.0/(i*j-20)=-0.058823529411764705
i=5 j=1 1.0/(i*j-20)=-0.06666666666666667
i=7 j=1 1.0/(i*j-20)=-0.07692307692307693
i=9 j=1 1.0/(i*j-20)=-0.09090909090909091
i=11 j=1 1.0/(i*j-20)=-0.11111111111111111
i=13 j=1 1.0/(i*j-20)=-0.14285714285714285
i=15 j=1 1.0/(i*j-20)=-0.2
i=17 j=1 1.0/(i*j-20)=-0.3333333333333333

```

```
i=19 j=1 1.0/(i*j-20)=-1.0
```

Пример использования break без метки:

```
for(int i=1;i<=10;i++){
    if( i+6== i*i ){
        break;
    };
    System.out.println("i="+i);
};
```

Данный цикл остановится при выполнении условия $i+6==i*i$. То есть вывод в окно консоли будет только для значений i , равных 1 и 2.

Ещё один пример использования break без метки:

```
for(int i=1;i<=20;i++){
for(int j=1;j<=20;j++){
    if(i*j==(i*j/2)*2){
        break;
    };
    System.out.println("i="+i+" j="+j+ " 1.0/(i*j-20)="+ (1.0/(i*j-20)) );
};
};
```

В этом случае будут выводиться все значения i и j до тех пор, пока не найдётся пара i и j , для которых $i*j==(i*j/2)*2$. После чего внутренний цикл прекращается – значения i , j и $1.0/(i*j-20)$ для данного и последующих значений j при соответствующем i не будут выводиться в окно консоли. Но внешний цикл (по i) будет продолжаться, и вывод продолжится для новых i и j . Результат будет таким же, как для continue с меткой для внешнего цикла.

Пример использования break с меткой:

```
label_for1:
for(int i=1;i<=20;i++){
for(int j=1;j<=20;j++){
    if(i*j==(i*j/2)*2){
        break label_for1;
    };
    System.out.println("i="+i+" j="+j+ " 1.0/(i*j-20)="+ (1.0/(i*j-20)) );
};
};
```

В этом случае также будут выводиться все значения i и j до тех пор, пока не найдётся пара i и j , для которых $i*j==(i*j/2)*2$. После чего прекращается внешний цикл, а значит – и внутренний тоже. Так что вывод в окно консоли прекратится. Поэтому вывод будет только

для $i=1, j=1$.

Краткие итоги по главе 5

- ✓ В Java имеются: условный оператор `if`, оператор выбора `case`, условное выражение `...?... : ...`, операторы инкремента `++` и декремента `--`,
- ✓ В Java имеются операторы цикла: `for`, `while` – цикл с предусловием, `do...while` – цикл с постусловием. А также операторы прерывания циклов `continue` и `break`, подпрограмм `-return`, программы - `System.exit`.
- ✓ Сравнение на равенство чисел в формате с плавающей точкой практически всегда некорректно. Но даже сравнение на неравенство таких чисел нельзя выполнять в случае, когда в точной арифметике эти числа должны быть равны. По этой причине нельзя использовать циклы с вещественным счётчиком в случаях, когда на интервале изменения счётчика укладывается целое число шагов счётчика.

Задания

- Написать приложение с графическим пользовательским интерфейсом, в котором по нажатию на кнопку иллюстрируется действие операторов цикла `for`, `while`, `do...while` – в зависимости от того, какая из кнопок `JRadioButton` нажата. С помощью этих циклов должна вычисляться площадь под кривой, задаваемой функцией $f(x)=a \cdot x^2+b \cdot x+c$, при x меняющемся от x_1 до x_2 , где величины a, b, c, x_1 и x_2 должны вводиться пользователем в соответствующих пунктах ввода.
- Добавить в это приложение вычисление с вещественным счётчиком цикла для какого-либо из операторов (`for`, `while` или `do...while`) в случае, когда отмечена соответствующая опция с помощью кнопки `JCheckBox`. Показать наличие неустойчивости при очень малом изменении входных параметров (какой-либо из величин a, b, c, x_1, x_2). Объяснить, почему изменение одних параметров приводит к неустойчивости, а других – нет.
- Написать приложение с графическим пользовательским интерфейсом, в котором по нажатию на кнопку `JButton` иллюстрируется действие операторов прерывания `continue`, `break`, `return`, `System.exit` – в зависимости от того, какая из кнопок `JToggleButton` нажата.