# Лабораторная работа № 3

# Тема работы: «Разработка программ с использованием управляющих инструкций»

* + 1. **Цель работы**

Закрепить умение работы с управляющими конструкциями.

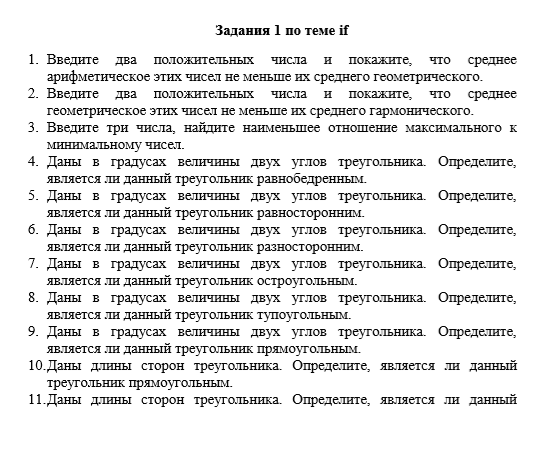
* + 1. **Задание**

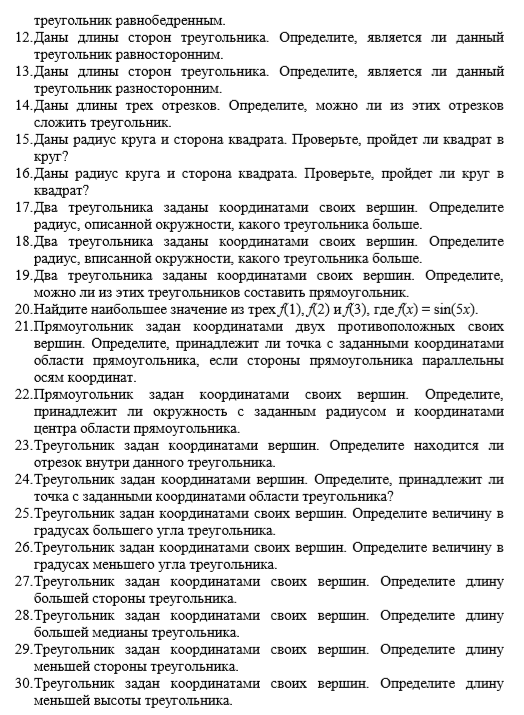
Три задания обязательны для выполнения. Задачи третьего задания необходимо решить двумя способами: с помощью циклов **while** и **do..while**.

Номер варианта соответствует вашему номеру по списку.

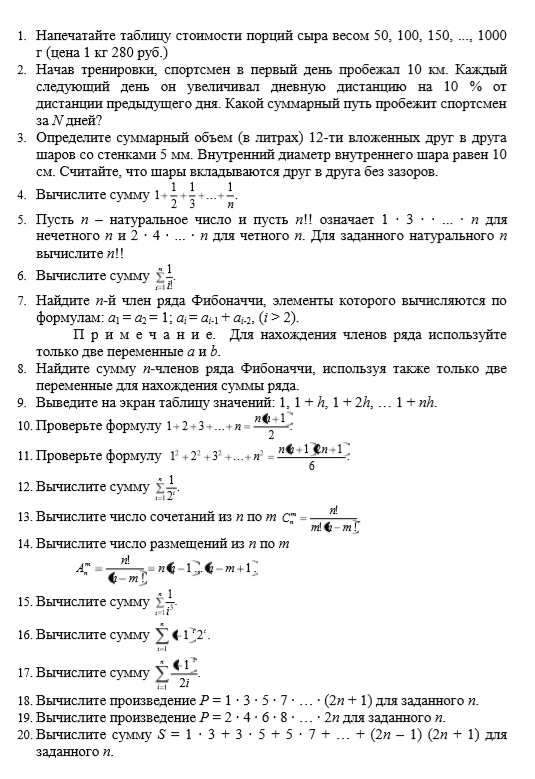
*Примечание: найти сумму бесконечного ряда с заданной степенью точности можно одним из двух способов:*

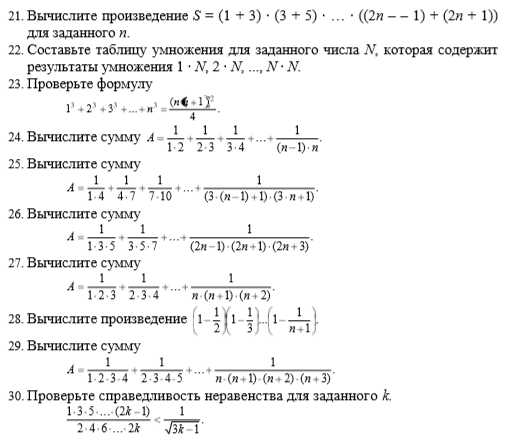
* *если бесконечный ряд убывающий, то необходимо производить вычисления до тех пор, пока очередной член ряда большее введенной точности;*
* *если бесконечный ряд неубывающий, то необходимо производить вычисления до тех пор, пока разность двух соседних членов по модулю больше введенной точности.*

****

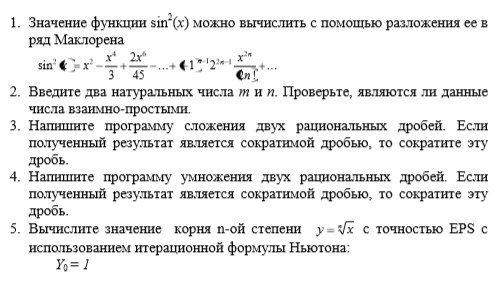


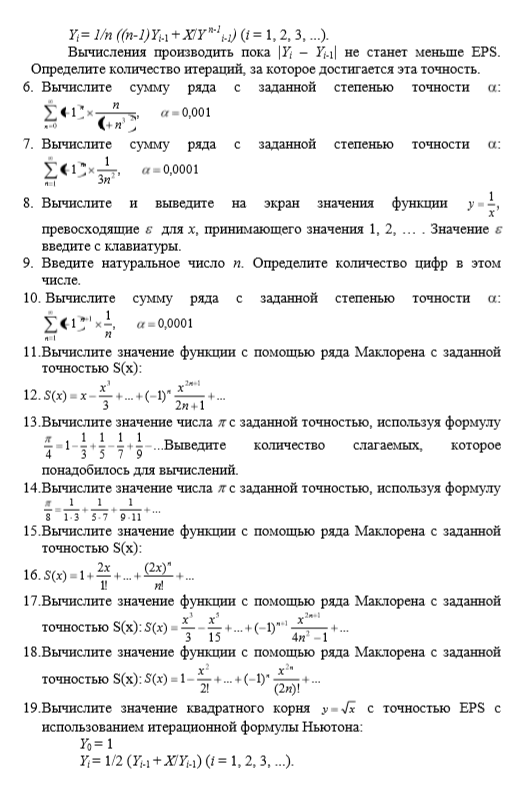
**Задание 2 по теме for**

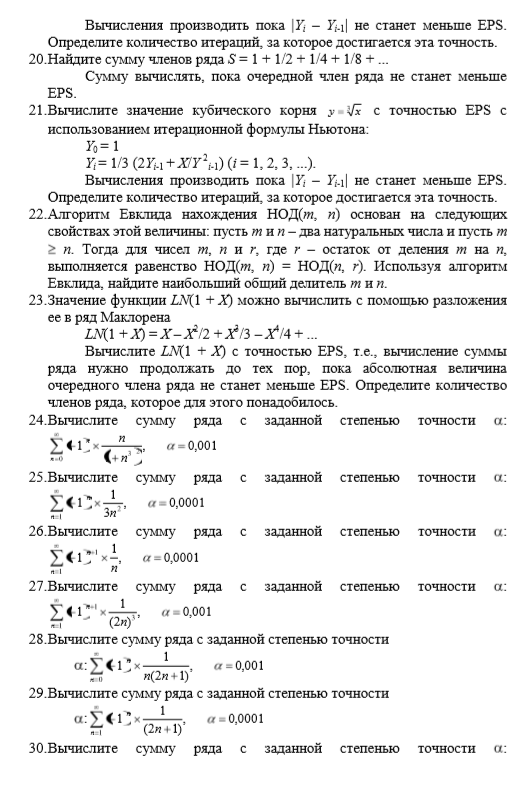


****

**Задание 3 по теме while и do..while**

****

****



* + 1. **Оснащение работы**

Задание по варианту, ЭВМ, среда разработки **IntelliJ IDEA**.

* + 1. **Основные теоретические сведения Оператор if**

Условный оператор **if** часто применяется программистами и имеется во всех языках программирования. Оператор **if** позволяет вашей программе в зависимости от условий выполнить оператор или группу операторов, основываясь на значении булевой переменной или выражения. Оператор **if** является основным оператором выбора в Java и позволяет выборочно изменять ход выполнения программы - и это одно из основных отличий между программированием и простым вычислением.

Оператор **if** начинается с ключевого слова **if**. Ключевое слово **if** должно сопровождаться булевым выражением, заключѐнным в скобки. Самая простая форма выглядит так:

if (условие) оператор; // если условие истинно, то выполняется оператор

Здесь условие - это булево выражение, имеющее значение *true* или *false*. Если условие истинно, то оператор или группа операторов выполняется, если ложно, то оператор не выполняется. Очень часто булево выражение в операторе **if** содержит какое-нибудь сравнение, но можно использовать булеву переменную или константу

// Если кот голоден if (isHungry)

Оператор **if** продолжается заключенным в фигурные скобки фрагментом, который называют блоком операторов. Если используется только один оператор, то фигурные скобки можно опустить. Но практика показывает, что лучше их всегда использовать, особенно в сложных проектах, когда постоянно приходится что-то переделывать.

Напишем следующий пример:

if (2 \* 2 == 5)

{

mResultEditText.setText("Дважды два равно пяти!);

}

Как вы думаете, что появится на экране? Правильно, ничего, так как оператор не будет выполняться, потому что условие 2 \* 2 == 5 является ложным.Обратите внимание, что оператор равенства состоит из двух символов знака равно. Об этом часто забывают начинающие программисты.

Вот список операторов, которые можно использовать в условных выражениях:

|  |  |
| --- | --- |
| **Оператор** | |
| < | Меньше чем |
| <= | Меньше или равно |
| > | Больше чем |
| >= | Больше или равно |
| == | Равно |
| != | Не равно |

Результат сравнения удобно использовать для изменения логики программы. Например, если кот голоден, то накормить его. Или если число нечѐтное, то сделать его чѐтным.

if(number % 2 != 0)

++number;

Существует расширенный вариант оператора **if** с использованием ключевого слова **else**:

if (условие) оператор; // если условие истинно, то выполняется первый оператор else оператор; // если условие ложно, то выполняется оператор после else

В этом случае при выполнении условия оператора **if** инициируется только один оператор, если условие не выполняется, то также инициируется только один оператор, который относится к **else**. Также можно использовать блоки операторов, тогда синтаксис будет выглядеть так:

if (условие)

{

оператор1; оператор2;

}

else

{

оператор1; оператор2;

}

Обе части оператора **if** и **else** не могут выполняться одновременно. А условное

выражение, управляющее оператором **if** должно возвращать булево значение.

Например, необходимо вычисления функции: Y(x) = x\*x, при x < 0 и Y(x)

= 2\*x, при x >= 0:

int x, y;

// Чтение значений x, y из консоли Scanner in = new Scanner(System.in); System.out.println("Enter x :");

x = in.nextLine(); System.out.println("Enter y :"); y = in.nextLine();

if (x < 0)

y = x\*x;

else

y = 2\*x;

System.out.printf ("Результат: %i\n", y);

**Вложенные операторы if**

Вложенный оператор **if** используется для дальнейшей проверки данных после того, как условие предыдущего оператора **if** принимает значение **true**. Иными словами, вложенный оператор применяется в тех случаях, когда для выполнения действия требуется соблюдение сразу нескольких условий, которые не могут быть указаны в одном условном выражении. Необходимо помнить, что во вложенных операторах **if-else** вторая часть **else** всегда относится к ближайшему оператору **if**, за условным выражением которого следует оператор ; или блок операторов. Вот небольшой пример:

if(i == 10)

{

if(j < 20) a = b; if(k > 100) c = d;

else a = c; // else относится к if(k > 100)

}

else a = d; // else относится к if(i == 10)

**Цепочка операторов if-else-if**

Часто используется цепочка операторов if-else-if - конструкция, состоящая из вложенных операторов if:

if (condition) statement;

else if (condition) statement;

else if (condition) statement;

.

.

.

else

statement;

Условные выражения оцениваются сверху вниз. Как только найдено условие, принимающее значение true, выполняется ассоциированный с этим условием оператор, а остальная часть цепочки пропускается. Если ни одно из условий не принимает значение *true*, то выполняется последний оператор **else**, который можно рассматривать как оператор по умолчанию. Если же последний оператор **else** отсутствует, а все условные выражения принимают значение *false*, то программа не выполняет никаких действий.

Напишем пример, вычисляющий время года, когда коты поют свои мартовские песни.

int month = 3; // март String season; // время года

if(month == 1 || month == 2 || month == 12) season = "Зимушка-зима";

else if (month == 3 || month == 4 || month == 5) season = "Весна";

else if (month == 6 || month == 7 || month == 8) season = "Лето";

else if (month == 9 || month == 10 || month == 11) season = "Осень";

else

season = "Вы с какой планеты?";

mInfoTextView.setText("Мартовские песни коты поют, когда на дворе " + season);

**Тернарный оператор**

Продвинутые программисты часто используют тернарный оператор ?

: вместо if-else. Тернарный оператор использует три операнда и записывается в форме:

логическое\_условие ? выражение1 : выражение2

Если **логическое\_условие** истинно, т.е. возвращает **true**, то берѐтся (или вычисляется) первое выражение слева от двоеточия, если возвращается **false**, то берѐтся второе выражение справа от двоеточия.

Например, нужно вычислить, какое из двух чисел больше и занести результат в третью переменную:

int largerNum; int lowNum = 9; int highNum = 27;

if(lowNum < highNum) // если первое число меньше второго

{

largerNum = highNum;

} else { // иначе largerNum = lowNum;

}

При тернарном варианте код будет следующим:

int lowNum = 9; int highNum = 27;

int largerNum = lowNum < highNum ? highNum : lowNum;

**Логические операторы**

Логические операторы работают только с операндами типа boolean. Все логические операторы с двумя операндами объединяют два логических значения, образуя результирующее логическое значения. Не путайте с побитовыми логическими операторами.

**Таблица логических операторов в Java**

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| & | Логическое AND (И) |
| && | Сокращѐнное AND |
| | | Логическое OR (ИЛИ) |
| || | Сокращѐнное OR |
| ^ | Логическое XOR (исключающее OR (ИЛИ)) |
| ! | Логическое унарное NOT (НЕ) |
| &= | AND с присваиванием |
| |= | OR с присваиванием |
| ^= | XOR с присваиванием |
| == | Равно |
| != | Не равно |
| ?: | Тернарный (троичный) условный оператор |

Логические операторы **&**, **|**, **^** действуют применительно к значениям типа **boolean** точно так же, как и по отношению к битам целочисленных значений. Логический оператор **!** инвертирует (меняет на противоположный) булево состояние: *!true == false* и *!false == true*.

**Таблица. Результаты выполнения логических операторов**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **A | B** | **A & B** | **A ^ B** | **!A** |
| false | False | false | false | false | true |
| true | False | true | false | true | false |
| false | True | true | false | true | true |
| true | True | true | true | false | false |

**Сокращѐнные логические операторы**

Кроме стандартных операторов **AND (&)** и **OR (|)** существуют сокращѐнные операторы **&&** и **||**.

Если взглянуть на таблицу, то видно, что результат выполнения оператора **OR** равен **true**, когда значение операнда A равно **true**, независимо от значения операнда B. Аналогично, результат выполнения оператора **AND** равен **false**, когда значение операнда A равно **false**, независимо от значения операнда B. Получается, что нам не нужно вычислять значение второго операнда, если результат можно определить уже по первому операнду. Это становится удобным в тех случаях, когда значение правого операнда зависит от значения левого.

Рассмотрим следующий пример. Допустим, мы ввели правило - кормить или не кормить кота в зависимости от числа пойманных мышек в неделю. Причѐм число мышек зависит от веса кота. Чем больше кот, тем больше он должен поймать мышей.

int mouse; // число мышек

int weight; // вес кота в граммах mouse = 5;

weight = 4500;

if (mouse != 0 & weight / mouse < 1000) { mInfoTextView.setText("Можно кормить кота");

}

Если запустить программу, то пример будет работать без проблем - пять мышей в неделю вполне достаточно, чтобы побаловать кота вкусным завтраком. Если он поймает четырѐх мышей, то начнутся проблемы с питанием кота, но не с программой - она будет работать, просто не будет выводить сообщение о разрешении покормить дармоеда.

Теперь возьмѐм крайний случай. Кот обленился и не поймал ни одной мышки. Значение переменной **mouse** будет равно 0, а в выражении есть оператор деления. А делить на 0 нельзя и наша программа закроется с ошибкой. Казалось бы, мы предусмотрели вариант с 0, но Java вычисляет оба выражения **mouse != 0** и **weight / mouse < 1000**, несмотря на то, что уже в первом выражении возвращается **false**.

Перепишем условие следующим образом (добавим всего лишь один символ):

if (mouse != 0 && weight / mouse < 1000) { mInfoTextView.setText("Можно кормить кота");

}

Теперь программа работает без краха. Как только Java увидела, что первое выражение возвращает **false**, то второе выражение с делением просто игнорируется.

Сокращѐнные варианты операторов **AND** и **OR** принято использовать в тех ситуациях, когда требуются операторы булевой логики, а их односимвольные родственники используются для побитовых операций.

**Оператор switch**

В отличие от операторов **if-then** и **if-then-else**, оператор **switch** применим к известному числу возможных ситуаций. Можно использовать простые типы **byte**, **short**, **char**, **int**. Также можно использовать **Enum** и **String** (начиная с JDK7), и специальные классы, которые являются обѐрткой для примитивных типов: **Character**, **Byte**, **Short**, **Integer**.

Дублирование значений **case** не допускается. Тип каждого значения должен быть совместим с типом выражения.

Команду **switch** часто называют командой выбора. Выбор осуществляется в зависимости от целочисленного выражения. Форма команды выглядит так:

switch(ВыражениеДляСравнения) { case Совпадение1:

команда; break;

case Совпадение2: команда;

break;

case Совпадение3: команда;

break; default:

оператор; break;

}

Параметр *ВыражениеДляСравнения* - выражение, в результате вычисления которого получается целое число (как правило). Команда **switch** сравнивает результат *ВыражениеДляСравнения* с каждым последующим *Совпадением*. Если обнаруживается совпадение, то исполняется команда или набор команд, которые прописаны за данным оператором. Если совпадений не будет, то исполняется команда после ключевого слова **default**. Однако оператор **default** не является обязательным. В этом случае при отсутствии совпадений программа не выполняет никаких действий.

Каждая секция **case** обычно заканчивается командой **break**, которая передаѐт управление к концу команды **switch**.

Рассмотрим простейший пример с месяцами. Запустим наш учебный проект и добавим код в обработчик нажатия кнопки:

int month = 3; String monthString; switch (month) {

case 1: monthString = "Январь"; break;

case 2: monthString = "Февраль"; break;

case 3: monthString = "Март"; break;

case 4: monthString = "Апрель"; break;

case 5: monthString = "Май";

break;

case 6: monthString = "Июнь"; break;

case 7: monthString = "Июль"; break;

case 8: monthString = "Август"; break;

case 9: monthString = "Сентябрь"; break;

case 10: monthString = "Октябрь"; break;

case 11: monthString = "Ноябрь"; break;

case 12: monthString = "Декабрь"; break;

default: monthString = "Не знаем такого"; break;

}

mInfoTextView.setText(monthString);

Запустите проект и нажмите кнопку - в текстовом поле появится слово *Март.*

При желании, можно переписать пример с использованием **if-then-else**:

int month = 3;

if (month == 1) { mInfoTextView.setText("Январь");

} else if (month == 2) { mInfoTextView.setText("Февраль");

}

... // и так далее

В каждом блоке **case** имеется оператор **break**, который прерывает свой блок кода. Его нужно использовать обязательно, иначе выполнение кода продолжится. Хотя иногда это и используется.

java.util.ArrayList<String> futureMonths = new java.util.ArrayList<String>();

int month = 8;

switch (month) { case 1:

futureMonths.add("January"); case 2:

futureMonths.add("February"); case 3:

futureMonths.add("March"); case 4:

futureMonths.add("April"); case 5:

futureMonths.add("May"); case 6:

futureMonths.add("June"); case 7:

futureMonths.add("July"); case 8:

futureMonths.add("August"); case 9:

futureMonths.add("September"); case 10:

futureMonths.add("October"); case 11:

futureMonths.add("November"); case 12:

futureMonths.add("December"); break;

default:

break;

}

if (futureMonths.isEmpty()) { mInfoTextView.setText("Invalid month number");

} else {

for (String monthName : futureMonths) { mInfoTextView.setText(monthName);

}

}

Следующий пример случайным образом генерирует английские буквы.

Программа определяет, гласные они или согласные:

Random random = new Random(); for (int i = 0; i < 100; i++) {

int c = random.nextInt(26) + 'a'; mInfoTextView.setText((char)c + ", " + c + ": "); switch (c) {

case 'a':

case 'e':

case 'i':

case 'o':

case 'u':

mInfoTextView.setText("Гласная"); break;

case 'y':

case 'w':

mInfoTextView.setText("Условная гласная"); break;

default:

mInfoTextView.setText("Согласная"); break;

}

}

Так как метод **Random.nextInt(26)** генерирует значение между 0 и 26, для получения символа нижнего регистра остаѐтся прибавить смещение 'a', при этом символ a автоматически преобразуется к типу **int**. Символы в секциях **case** также представляют собой целочисленные значения, используемые для сравнения.

Чтобы вывести переменную **c** в символьном виде, еѐ необходимо преобразовать к типу **char**, иначе значение будет выведено в числовом виде.

В Java SE 7 появилась возможность использовать объект **String** в операторе **switch**. Возможно, это будет работать и на Android в будущем (Upd.: вроде уже работает):

public class StringSwitchDemo {

public static int getMonthNumber(String month) { int monthNumber = 0;

if (month == null) { return monthNumber;

}

switch (month.toLowerCase()) { case "january":

monthNumber = 1; break;

case "february": monthNumber = 2; break;

case "march":

monthNumber = 3; break;

case "april":

monthNumber = 4; break;

case "may":

monthNumber = 5; break;

case "june":

monthNumber = 6; break;

case "july":

monthNumber = 7; break;

case "august": monthNumber = 8; break;

case "september": monthNumber = 9; break;

case "october": monthNumber = 10; break;

case "november": monthNumber = 11; break;

case "december": monthNumber = 12; break;

default:

monthNumber = 0; break;

}

return monthNumber;

}

public static void main(String[] args) { String month = "August";

int returnedMonthNumber = StringSwitchDemo.getMonthNumber(month);

if (returnedMonthNumber == 0) { System.out.println("Invalid month");

} else {

System.out.println(returnedMonthNumber);

}

}

}

Важные свойства оператора **switch**:

* Оператор **switch** отличается от оператора **if** тем, что может выполнять проверку только равенства, а оператор **if** может вычислять результат булева выражения любого типа.
* Две константы **case** в одном и том же операторе **switch** не могут иметь одинаковые значения
* Оператор **switch** эффективнее набора вложенных операторов **if**

**Цикл for**

Конструкция for управляет циклами. Команда выполняется до тех пор, пока управляющее логическое выражение не станет ложным.

Блок-схема.

Цикл **for** является наиболее распространѐнным циклом в программировании, поэтому его следует изучить. Цикл **for** проводит инициализацию перед первым шагом цикла. Затем выполняется проверка условия цикла, и в конце каждой итерации происходит изменение управляющей переменной. Выглядит следующим образом:

for(инициализация; логическое выражение (условие); шаг (итерация)) команда

Любое из трѐх выражений цикла (инициализация, логическое выражение или шаг) можно пропустить. Перед выполнением каждого шага цикла

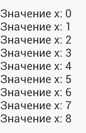
проверяется условие цикла. Если условие окажется ложным, то выполнение продолжится с инструкции, следующей за конструкцией **for**.

Помните, что выражение инициализации выполняется один раз, затем вычисляется условие, которое должно быть булевым выражением.

Как правило, цикл **for** используют для перебора. В качестве имени первой переменной часто используют **i** (сокр. от init), но вы можете использовать любое имя.

Простейший пример:

for (int x = 0; x < 9; x = x + 1) System.out.println("\nЗначение x: " + x);



В этом примере переменной x присваивается начальное значение, равное нулю. Затем выполняется проверка условия в логическом выражении (x < 9), если результат проверки истинен, то выполняется оператор после выражения цикла. После чего процесс повторяется. Процесс продолжается до тех пор, пока результат проверки условия не станет ложным.

Третье выражение в цикле - шаг, то есть, на какое значение нужно изменить переменную. Строго говоря, в таком виде (x = x + 1) современные программисты не пишут, так как есть укороченная форма записи (x++). Предыдущий пример можно переписать по другому:

for (int x = 0; x < 9; x++)

Эта запись является классической и правильной, если нам нужно посчитать от 0 до 8. Может возникнуть соблазн написать, например, так:

for (int x = 0; x <= 8; x++)

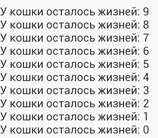
Результат будет таким же, но такой код нежелателен. Старайтесь писать традиционно. Особенно это проявляется при работе с массивами.

Увеличение значения переменной на единицу - весьма распространѐнный случай. Но это не является обязательным условием цикла, вы можете установить шаг и с помощью умножения, вычитания и других действий.

Например, мы хотим вывести процесс уменьшения жизней у очень любопытной кошки:

for (int life = 9; life >= 0; life--)

System.out.println ("\nУ кошки осталось жизней: " + life);



Например, выводим чѐтные числа.

for (int x = 0; x < 9; x += 2)

System.out.println(" " + x); // между кавычками пробел

Получим:

0 2 4 6 8

Если нужно выполнить несколько операторов в цикле, то используют фигурные скобки.

for (int kitten = 1; kitten < 10; kitten++) { mInfoTextView.append("\nСчитаем котят: " + kitten); mResultEditText.setText("Ура! Нас подсчитали");

}

В этом примере выполняется цикл с выводом числа подсчитанных котят, а также выводится текст в текстовом поле. Кстати, это один из примеров неправильного кода, когда в текстовом поле девять раз подряд выводится одна и та же строка. Мы этого не замечаем, но в реальности процессор выполняет лишнюю работу и второй оператор безусловно нужно вынести за пределы блока кода, который относится к циклу. Подобные ошибки очень часто встречаются у начинающих программистов, которые забывают, как работает цикл.

Когда мы объявляем переменную в первой части оператора **for(int i = 0;**

**...)**, то область видимости переменной ограничена телом цикла и эта переменная не доступна на другом участке кода. Это подходящий вариант, если переменная больше нигде не используется. При этом переменная имеет область видимости и продолжительность существования, совпадающие с видимостью и продолжительностью жизни самого цикла. Вне цикла переменная прекратит своѐ существование.

Если управляющую переменную цикла нужно использовать в других частях приложения, то еѐ не следует объявлять внутри цикла.

int i; // эта переменная нам понадобится не только в цикле for(i = 0; i < 10; i++){

// что-то делаем

}

// можем использовать переменную где-то ещѐ x = i + 10;

С другой стороны, если видимость переменной ограничена в пределах цикла, то не будет никакого конфликта, если вы будете использовать одинаковые имена переменных в разных циклах **for**, так как они не будут друг другу мешать.

**Использование нескольких переменных**

Иногда требуется указать несколько переменных в инициализационной части цикла. Для начала посмотрим на стандартный пример:

int a, b; b = 4;

for(a = 1: a < b; a++) { mInfoTextView.append("a = " + a + "\n"); mInfoTextView.append("b = " + b + "\n"); b--;

}

В данном цикле используются две переменные. Можно включить обе переменные в оператор **for**, чтобы не выполнять обработку второй переменной вручную:

int a, b;

for(a = 1, b = 4; a < b; a++, b--) { mInfoTextView.append("a = " + a + "\n"); mInfoTextView.append("b = " + b + "\n");

}

Как видно из кода, запятая служит разделителем для двух переменных. Теперь оба разделѐнных запятой оператора в итерационной части цикла выполняются при каждом выполнении цикла. Данный код считается более эффективным, хотя на практике встречается редко.

Части цикла могут быть пустыми. Оставим пустым первое выражение.

int i = 0;

for(; i < 10; i++){

// что-то делаем mInfoTextView.append("\n" + i);

}

В следующем примере нет инициализационного и итерационного выражения:

int i;

boolean kind = false;

i = 0;

for( ; !kind; ) {

mInfoTextView.append("i равно " + i + "\n"); if(i == 10) done = true;

i++;

}

А можно вообще все три части оператора оставить пустыми:

for( ; ;) {

//...

}

В этом случае создаѐтся бесконечный цикл, который никогда не завершится. Практического смысла данный код не имеет.

**Цикл while**

Оператор цикла **while** есть практически во всех языках программирования. Он повторяет оператор или блок операторов до тех пор, пока значение его управляющего выражения истинно.

Форма цикла **while** следующая:

while(условие) {

// тело цикла

}

Здесь *условие* должно быть любым булевым выражением. Тело цикла будет выполняться до тех пор, пока условное выражение истинно. Когда *условие* становится ложным, управление передаѐтся строке кода, которая идѐт после цикла. Если в цикле используется только один оператор, то фигурные скобки можно опустить (но лучше так не делать).

Логическое выражение вычисляется перед началом цикла, а затем каждый раз перед выполнением очередного повторения оператора.

Напишем пример с использованием цикла **while**, который выполняет обратный отсчѐт от 10 до 0:

int counter = 10; while (counter > 0) {

mInfoTextView.append("Осталось " + counter + " сек.\n"); counter--;

}

Программа выведет десять строк:

Осталось 10 сек.

Осталось 9 сек.

Осталось 8 сек.

Осталось 7 сек.

Осталось 6 сек.

Осталось 5 сек.

Осталось 4 сек.

Осталось 3 сек.

Осталось 2 сек.

Осталось 1 сек.

Если нужно увеличивать от единицы до 10, то код будет следующим.

int counter = 1; while(counter < 11){

System.out.println(counter); counter++;

}

Поскольку цикл **while** вычисляет своѐ условное выражение в начале цикла, то тело цикла не будет выполняться, если условие с самого начала было ложным.

boolean isHungry; // голоден ли кот

isHungry = true; // где вы видали сытого кота? while(!isHungry) {

mInfoTextView.setText("Случилось чудо - кот не голоден");

}

Вы никогда не увидите сообщение, так как сытый кот - это из области

фантастики.

Тело цикла **while** может быть пустым. Например:

int i, j; i = 10;

j = 30;

// вычисляем среднее значение двух переменных while (++i < --j)

; // у цикла нет тела mInfoTextView.setText("Среднее значение равно " + i);

Пример работает следующим образом. Значение переменной **i** увеличивается, а значение переменной **j** уменьшается на единицу. Затем программа сравнивает два новых значения переменных. Если новое значение переменной **i** меньше нового значения переменной **j**, то цикл повторяется. На каком-то этапе значения обоих переменных сравняются и цикл прекратит свою работу. При этом переменная **i** будет содержать среднее значение исходных значений двух переменных. Достаточно изуверский способ вычисления среднего значения, но здесь главное увидеть пример цикла без тела. Все действия выполняются внутри самого условного выражения. Учтите, если значение первой переменной с самого начала будет больше второй переменной, то код пойдѐт коту под хвост.

Профессиональные программисты часто используют циклы без тела, в которых само по себе управляющее выражение может выполнять все необходимые действия.

**Цикл do-while**

Конструкция цикла:

do

// команда (тело цикла)

while(условие-логическое выражение)

Отличие цикла **do-while** от цикла [while](http://developer.alexanderklimov.ru/android/java/while.php) состоит в том, что цикл **do- while** выполняется по крайней мере один раз, даже если условие изначально ложно. В цикле **while** такое не произойдѐт, так как тело цикла не отработается. Цикл **do-while** используется реже, чем **while**.

Бывают ситуации, когда проверку прерывания цикла желательно выполнять в конце цикла, а не в его начале. И данный цикл позволяет это сделать. При каждом повторении цикла **do-while** программа сначала выполняет тело цикла, а затем вычисляет условное выражение. Если это выражение истинно, то цикл повторяется. В противном случае выполнение цикла прерывается. Как и в других циклах Java, условие должно иметь булево значение.

Перепишем пример из урока по циклу **while** на новый лад с использованием цикла **do-while**:

int counter = 10; do {

mInfoTextView.append("Осталось " + counter + " сек.\n"); counter--;

} while (counter > 0);

Если теперь изменить условие **counter < 0**, то цикл отработает один раз и выведет одну строку:

Осталось 10 сек.

Пример можно переписать следующим образом:

int counter = 10; do {

mInfoTextView.append("Осталось " + counter + " сек.\n");

} while (--counter > 0);

Мы объединили декремент счѐтчика и сравнение с нулѐм в одном выражении. Программа работает следующим образом. Вначале она выполняет операцию уменьшения на единицу и возвращая новое значение счѐтчика. Затем сравнивается значение с нулѐм. Если оно больше нуля, выполнение цикла продолжается. В противном случае цикл прерывается.

**Оператор break**

Оператор **break** завершает последовательность операторов в операторе **switch**, позволяет выйти из цикла и в качестве оператора безусловного перехода (goto).

Рассмотрим пример выхода из цикла. Используя оператор **break**, можно вызвать немедленное завершение цикла, пропуская условное выражение и другой код в теле цикла. Когда приложение встречает оператор **break** внутри цикла, оно прекращает выполнение цикла и передаѐт управление оператору, следующему за циклом.

for(int i = 0; i < 100; i++) {

if(i == 5) break; // выходим из цикла, если i равно 5 mInfoTextView.append("i: " + i + "\n");

}

mInfoTextView.append("Цикл завершѐн");

Получим следующий результат:

i: 0

i: 1

i: 2

i: 3

i: 4

Цикл завершѐн.

**Оператор continue**

Иногда требуется, чтобы повторение цикла начиналось с более раннего оператора его тела. В циклах **while** и **do-while** оператор **continue** вызывает передачу управления непосредственно управляющему условному выражению цикла. В цикле **for** управление передаѐтся вначале итерационной части цикла **for**, а потом условному выражению. При этом во всех циклах промежуточный код пропускается.

В следующем примере выводим два числа в каждой строке:

for (int i = 0; i < 10; i++) {

mInfoTextView.append(i + " "); if (i % 2 == 0)

continue; mInfoTextView.append("\n");

}

В этом коде оператор % служит для проверки чѐтности значения переменной *i*. Если значение чѐтное, выполнение цикла продолжится дальше, а если нечѐтное, то произойдѐт переход на новую строку. В результате мы получим следующий текст:

0 1

2 3

4 5

6 7

8 9

Как и оператор **break**, оператор **continue** может содержать метку содержащего его цикла, который нужно продолжить. Создадим пример вывода треугольной таблицы умножения чисел от 0 до 9.

outer: for (int i = 0; i < 10; i++) { for (int j = 0; j < 10; j++) {

if (j > i) {

mInfoTextView.append("\n"); continue outer;

}

mInfoTextView.append(" " + (i \* j));

}

}

mInfoTextView.append("\n");

В этом примере оператор **continue** прерывает цикл подсчѐта значений переменной *j* и продолжает его со следующей итерации цикла подсчѐта переменной *i*. На экране увидим текст в таком виде:

0

0 1

0 2 4

0 3 6 9

0 4 8 12 16

и т.д.

Данный оператор в практике встречается достаточно редко. Но на всякий случай помните о нѐм. Иногда он может пригодиться.

* + 1. **Порядок выполнения работы**

1. построить объектную модель;
2. построить алгоритм решения данной задачи;
3. реализовать приложение;
4. провести тестирование полученного приложения.