

## Управление ходом программы

Управление потоком вычислений является фундаментальной основой всего языка программирования. В данной лекции будут рассмотрены основные языковые конструкции и способы их применения.

Синтаксис выражений весьма схож с синтаксисом языка C, что облегчает его понимание для программистов, знакомых с этим языком, и вместе с тем имеется ряд отличий, которые будут рассмотрены позднее и на которые следует обратить внимание.

Порядок выполнения программы определяется операторами. Операторы могут содержать другие операторы или выражения.

## Нормальное и прерванное выполнение операторов

Последовательность выполнения операторов может быть непрерывной, а может и прерываться (при возникновении определенных условий). Выполнение оператора может быть прервано, если в потоке вычислений будут обнаружены операторы

```
break
continue
return
```

Тогда управление будет передано в другое место (в соответствии с правилами обработки этих операторов, которые мы рассмотрим позже).

Нормальное выполнение оператора может быть прервано также при возникновении исключительных ситуаций, которые тоже будут рассмотрены позднее. Явное возбуждение исключительной ситуации с помощью оператора `throw` также прерывает нормальное выполнение оператора и передает управление выполнением программы (далее просто управление) в другое место.

Прерывание нормального исполнения всегда вызывается определенной причиной. Приведем список таких причин:

- `break` (без указания *метки*);
- `break` (с указанием *метки*);
- `continue` (без указания *метки*);
- `continue` (с указанием *метки*);
- `return` (с возвратом значения);
- `return` (без возврата значения);
- `throw` с указанием объекта `Throwable`, а также все исключения, вызываемые виртуальной машиной Java.

Выражения могут завершаться нормально и преждевременно (аварийно). В данном случае термин "аварийно" вполне применим, т.к. причиной необычной последовательности выполнения выражения может быть только возникновение исключительной ситуации.

Если в операторе содержится выражение, то в случае его аварийного завершения выполнение оператора тоже будет завершено преждевременно (т.е. нормальный ход выполнения оператора будет нарушен).

В том случае, если в операторе имеется вложенный оператор и его завершение происходит ненормально, то так же ненормально завершается оператор, содержащий вложенный (в некоторых случаях это не так, что будет оговариваться особо).

## Блоки и локальные переменные

Блок - это последовательность операторов, объявлений локальных классов или локальных переменных, заключенных в скобки. Область видимости локальных переменных и классов ограничена блоком, в котором они определены.

Операторы в блоке выполняются слева направо, сверху вниз. Если все операторы (выражения) в блоке выполняются нормально, то и весь блок выполняется нормально. Если какой-либо оператор (выражение) завершается ненормально, то и весь блок завершается ненормально.

Нельзя объявлять несколько локальных переменных с одинаковыми именами в пределах видимости блока. Приведенный ниже код вызовет ошибку времени компиляции.

```
public class Test {  
  
    public Test() {  
    }  
    public static void main(String[] args) {  
        Test t = new Test();  
        int x;  
        lbl: {  
            int x = 0;  
            System.out.println("x = " + x);  
        }  
    }  
}
```

В то же время не следует забывать, что локальные переменные перекрывают видимость переменных-членов. Так, следующий пример отработает нормально.

```
public class Test {  
    static int x = 5;  
    public Test() { }  
    public static void main(String[] args) {  
        Test t = new Test();  
        int x = 1;  
        System.out.println("x = " + x);  
    }  
}
```

На консоль будет выведено `x = 1`.

То же самое правило применимо к параметрам методов.

```
public class Test {  
    static int x;  
    public Test() {  
    }  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.test(5);  
        System.out.println("Member value x = "  
            + x);  
    }  
    private void test(int x){  
        this.x = x + 5;  
        System.out.println("Local value x = "  
            + x);  
    }  
}
```

В результате работы этого примера на консоль будет выведено:

```
Local value x = 5  
Member value x = 10
```

На следующем примере продемонстрируем, что область видимости локальной переменной ограничена областью видимости блока, или оператора, в пределах которого данная переменная объявлена.

```
public class Test {  
    static int x = 5;  
    public Test() {  
    }  
    public static void main(String[] args) {  
        Test t = new Test();  
        {  
            int x = 1;  
            System.out.println("First block x = "  
                + x);  
        }  
        {  
            int x = 2;  
            System.out.println("Second block x ="  
                + x);  
        }  
        System.out.print("For cycle x = ");  
        for(int x =0;x<5;x++) {  
            System.out.print(" " + x);  
        }  
    }  
}
```

Данный пример откомпилируется без ошибок и на консоль будет выведен следующий результат:

```
First block x = 1  
Second block x =2  
For cycle x = 0 1 2 3 4
```

Следует помнить, что определение локальной переменной есть исполняемый оператор. Если задана инициализация переменной, то выражение выполняется слева направо и его результат присваивается локальной переменной. Использование неинициализированных локальных переменных запрещено и вызывает ошибку компиляции.

Следующий пример кода

```
public class Test {  
    static int x = 5;  
    public Test() {  
    }  
    public static void main(String[] args) {  
        Test t = new Test();  
        int x;  
        int y = 5;  
        if( y > 3) x = 1;  
        System.out.println(x);  
    }  
}
```

вызовет ошибку времени компиляции, т.к. возможны условия, при которых переменная `x` может быть не инициализирована до ее использования (несмотря на то,

что в данном случае оператор `if(y > 3)` и следующее за ним выражение `x = 1;` будут выполняться всегда).

## Пустой оператор

Точка с запятой (;) является пустым оператором. Данная конструкция вполне применима там, где не предполагается выполнение никаких действий. Преждевременное завершение пустого оператора невозможно.

## Метки

Любой оператор, или блок, может иметь *метку*. Метку можно указывать в качестве параметра для операторов `break` и `continue`. Область видимости метки ограничивается оператором, или блоком, к которому она относится. Так, в следующем примере мы получим ошибку компиляции:

```
public class Test {
    static int x = 5;
    static {
    }
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int x = 1;
        Lbl1: {
            if(x == 0) break Lbl1;
        }

        Lbl2:{
            if(x > 0) break Lbl1;
        }
    }
}
```

В случае, если имеется несколько вложенных блоков и операторов, допускается обращение из внутренних блоков к *меткам*, относящимся к внешним.

Этот пример является вполне корректным:

```
public class Test {
    static int x = 5;
    static {
    }
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int L2 = 0;
        Test: for(int i = 0; i < 10; i++) {
            test: for(int j = 0; j < 10; j++) {
                if( i*j > 50) break Test;
            }
        }
    }
    private void test() {
        ;
    }
}
```

В этом же примере можно увидеть, что *метки* используют пространство имен, отличное от пространства имен переменных, методов и классов.

Традиционно использование меток не рекомендуется, особенно в объектно-ориентированных языках, поскольку серьезно усложняет понимание порядка выполнения кода, а значит, и его тестирование и отладку. Для Java этот запрет можно считать не столь строгим, поскольку самый опасный оператор `goto` отсутствует. В некоторых ситуациях (как в рассмотренном примере с вложенными циклами) использование меток вполне оправданно, но, конечно, их применение следует ограничивать лишь самыми необходимыми случаями.

## Оператор if

Пожалуй, наиболее распространенной конструкцией в Java, как и в любом другом структурном языке программирования, является оператор условного перехода.

В общем случае конструкция выглядит так:

```
if (логическое выражение) выражение или блок 1
else выражение или блок 2
```

Логическое выражение может быть любой языковой конструкцией, которая возвращает булевский результат. Отметим отличие от языка C, в котором в качестве логического выражения могут использоваться различные типы данных, где отличное от нуля выражение трактуется как истинное значение, а ноль - как ложное. В Java возможно использование только логических выражений.

Если логическое выражение принимает значение "истина", то выполняется выражение или блок 1, в противном случае - выражение или блок 2. Вторая часть оператора (`else`) не является обязательной и может быть опущена. Т.е. конструкция `if(x == 5) System.out.println("Five")` вполне допустима.

Операторы `if-else` могут каскадироваться.

```
String test = "smb";
if( test.equals("value1") {
    ...
} else if (test.equals("value2") {
    ...
} else if (test.equals("value3") {
    ...
} else {
    ...
}
```

Следует помнить, что оператор `else` относится к ближайшему к нему оператору `if`. В данном случае последнее условие `else` будет выполняться, только если не выполнено предыдущее. Заключительная конструкция `else` относится к самому последнему условию `if` и будет выполнена только в том случае, если ни одно из вышеперечисленных условий не будет истинным. Если хотя бы одно из условий выполнено, то все последующие выполняться не будут.

Например:

```
...
int x = 5;
if( x < 4) {
    System.out.println("Меньше 4");
} else if (x > 4) {
    System.out.println("Больше 4");
}
```

```
} else if (x == 5) {  
    System.out.println("Равно 5");  
} else{  
    System.out.println("Другое значение");  
}
```

Предложение "Равно 5" в данном случае напечатано не будет.

## Оператор switch

Оператор `switch()` удобно использовать в случае необходимости множественного выбора. Выбор осуществляется на основе целочисленного значения.

Структура оператора:

```
switch(int value) {  
    case const1:  
        выражение или блок  
    case const2:  
        выражение или блок  
    case constn:  
        выражение или блок  
    default:  
        выражение или блок  
}
```

Причем, фраза `default` не является обязательной.

В качестве параметра `switch` может использоваться переменная типа `byte`, `short`, `int`, `char` или выражение. Выражение должно в конечном итоге возвращать параметр одного из указанных ранее типов. В операторе `case` не могут применяться значения примитивного типа `long` и ссылочных типов `Long`, `String`, `Integer`, `Byte` и т.д.

При выполнении оператора `switch` производится последовательное сравнение значения `x` с константами, указанными после `case`, и в случае совпадения выполняется выражение следующего за этим условием. Если выражение выполнено нормально и нет преждевременного его завершения, то производится сравнение для последующих `case`. Если же выражение, следующее за `case`, завершилось ненормально, то будет прекращено выполнение всего оператора `switch`.

Если не выполнен ни один оператор `case`, то выполнится оператор `default`, если он имеется в данном `switch`. Если оператора `default` нет и ни одно из условий `case` не выполнено, то ни одно из выражений `switch` также выполнено не будет.

Следует обратить внимание, что, в отличие от многозвенного `if-else`, если какое-либо условие `case` выполнено, то выполнение `switch` не прекратится, а будут проверяться следующие за ним условия. Если этого необходимо избежать, то после кода следующего за оператором `case` используется оператор `break`, прерывающий дальнейшее выполнение оператора `switch`.

После оператора `case` должен следовать литерал, который может быть интерпретирован как 32-битовое целое значение. Здесь не могут применяться выражения и переменные, если они не являются `final static`.

Рассмотрим пример:

```
int x = 2;  
switch(x) {  
    case 1:
```

```

        case 2:
            System.out.println("Равно 1 или 2");
            break;
        case 3:
        case 4:
            System.out.println("Равно 3 или 4");
            break;
        default:
            System.out.println(
                "Значение не определено");
    }

```

В данном случае на консоль будет выведен результат "Равно 1 или 2". Если же убрать операторы `break`, то будут выведены все три строки.

Вот такая конструкция вызовет ошибку времени компиляции.

```

int x = 5;
switch(x) {
    case y:    // только константы!
        ...
        break;
}

```

В операторе `switch` не может быть двух `case` с одинаковыми значениями.

Т.е. конструкция

```

switch(x) {
    case 1:
        System.out.println("One");
        break;
    case 1:
        System.out.println("Two");
        break;
    case 3:
        System.out.println("Tree or other value");
}

```

недопустима.

Также в конструкции `switch` может быть применен только один оператор `default`.

## Управление циклами

В языке Java имеется три основных конструкции управления циклами:

- цикл `while`;
- цикл `do`;
- цикл `for`.

### Цикл `while`

Основная форма цикла `while` может быть представлена так:

```

while(логическое выражение)
    повторяющееся выражение, или блок;

```

В данной языковой конструкции повторяющееся выражение, или блок будет исполняться до тех пор, пока логическое выражение будет иметь истинное значение. Этот многократно исполняемый блок называют телом цикла

Операторы `continue` и `break` могут изменять нормальное исполнение тела цикла. Так, если в теле цикла встретился оператор `continue`, то операторы, следующие за ним, будут пропущены и выполнение цикла начнется сначала. Если `continue` используется с *меткой* и *метка* принадлежит к данному `while`, то выполнение его будет аналогичным. Если *метка* не относится к данному `while`, его выполнение будет прекращено и управление будет передано на оператор, или блок, к которому относится *метка*.

Если встретился оператор `break`, то выполнение цикла будет прекращено.

Если выполнение блока было прекращено по какой-то другой причине (возникла исключительная ситуация), то выполнение всего цикла будет прекращено по той же причине.

Рассмотрим несколько примеров:

```
public class Test {
    static int x = 5;
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int x = 0;
        while(x < 5) {
            x++;
            if(x % 2 == 0) continue;
            System.out.print(" " + x);
        }
    }
}
```

На консоль будет выведено

1    3    5

т.е. вывод на печать всех четных чисел будет пропущен.

```
public class Test {
    static int x = 5;
    public Test() { }
    public static void main(String[] args) {
        Test t = new Test();
        int x = 0;
        int y = 0;
        lbl: while(y < 3) {
            y++;
            while(x < 5) {
                x++;
                if(x % 2 == 0) continue lbl;
                System.out.println("x=" + x + " y="+y);
            }
        }
    }
}
```

На консоль будет выведено



```
x=1 y=1
x=3 y=2
x=5 y=3
```

т.е. при выполнении условия `if(x % 2 == 0) continue lbl;` цикл по переменной `x` будет прерван, а цикл по переменной `y` начнет новую итерацию.

Типичный вариант использования выражения `while()`:

```
int i = 0;
while( i++ < 5) {
    System.out.println("Counter is " + i);
}
```

Следует помнить, что цикл `while()` будет выполнен только в том случае, если на момент начала его выполнения логическое выражение будет истинным. Таким образом, при выполнении программы может иметь место ситуация, когда цикл `while()` не будет выполнен ни разу.

```
boolean b = false;
while(b) {
    System.out.println("Executed");
}
```

В данном случае строка `System.out.println("Executed");` выполнена не будет.

## Цикл do

Основная форма цикла `do` имеет следующий вид:

```
do
    повторяющееся выражение или блок;
while(логическое выражение)
```

Цикл `do` будет выполняться до тех пор, пока логическое выражение будет истинным. В отличие от цикла `while`, этот цикл будет выполнен, как минимум, один раз.

Типичная конструкция цикла `do`:

```
int counter = 0;
do {
    counter ++;
    System.out.println("Counter is " + counter);
} while(counter < 5);
```

В остальном выполнение цикла `do` аналогично выполнению цикла `while`, включая использование операторов `break` и `continue`.

## Цикл for

Довольно часто бывает необходимо изменять значение какой-либо переменной в заданном диапазоне и выполнять повторяющуюся последовательность операторов с использованием этой переменной. Для выполнения такой последовательности действий как нельзя лучше подходит конструкция цикла `for`.

Основная форма цикла `for` выглядит следующим образом:

```
for(выражение инициализации; условие; выражение обновления)
```

повторяющееся выражение или блок;

Ключевыми элементами данной языковой конструкции являются предложения, заключенные в круглые скобки и разделенные точкой с запятой.

Выражение инициализации выполняется до начала выполнения тела цикла. Чаще всего используется как некое стартовое условие (инициализация, или объявление переменной).

Условие должно быть логическим выражением и трактуется точно так же, как логическое выражение в цикле `while()`. Тело цикла выполняется до тех пор, пока логическое выражение истинно. Как и в случае с циклом `while()`, тело цикла может не исполниться ни разу. Это происходит, если логическое выражение принимает значение "ложь" до начала выполнения цикла.

Выражение обновления выполняется сразу после исполнения тела цикла и до того, как проверено условие продолжения выполнения цикла. Обычно здесь используется выражение инкрементации, но может быть применено и любое другое выражение.

Пример использования цикла `for()`:

```
...
for(counter=0;counter<10;counter++)
{
    System.out.println("Counter is " + counter);
}
```

В данном примере предполагается, что переменная `counter` была объявлена ранее. Цикл будет выполнен 10 раз и будут напечатаны значения счетчика от 0 до 9.

Разрешается определять переменную прямо в предложении:

```
for(int cnt = 0;cnt < 10; cnt++) {
    System.out.println("Counter is " + cnt);
}
```

Результат выполнения этой конструкции будет аналогичен предыдущему. Однако нужно обратить внимание, что область видимости переменной `cnt` будет ограничена телом цикла.

Любая часть конструкции `for()` может быть опущена. В вырожденном случае мы получим оператор `for` с пустыми значениями

```
for(;;) {
    ...
}
```

В данном случае цикл будет выполняться бесконечно. Эта конструкция аналогична конструкции `while(true){}`. Условия, в которых она может быть применена, мы рассмотрим позже.

Возможно также расширенное использование синтаксиса оператора `for()`. Предложение и выражение могут состоять из нескольких частей, разделенных запятыми.

```
for(i = 0, j = 0; i<5;i++, j+=2) {
    ...
}
```

Использование такой конструкции вполне правомерно.

## Операторы `break` и `continue`

В некоторых случаях требуется изменить ход выполнения программы. В традиционных языках программирования для этих целей применяется оператор `goto`, однако в Java он не поддерживается. Для этих целей применяются операторы `break` и `continue`.

### Оператор `continue`

Оператор `continue` может использоваться только в циклах `while`, `do`, `for`. Если в потоке вычислений встречается оператор `continue`, то выполнение текущей последовательности операторов (выражений) должно быть прекращено и управление будет передано на начало блока, содержащего этот оператор.

```
...
int x = (int) (Math.random() * 10);
int arr[10] = {....}
for(int cnt=0; cnt<10; cnt++) {
    if(arr[cnt] == x) continue;
    ...
}
```

В данном случае, если в массиве `arr` встретится значение, равное `x`, то выполнится оператор `continue` и все операторы до конца блока будут пропущены, а управление будет передано на начало цикла.

Если оператор `continue` будет применен вне контекста оператора цикла, то будет выдана ошибка времени компиляции. В случае использования вложенных циклов оператору `continue`, в качестве адреса перехода, может быть указана *метка*, относящаяся к одному из этих операторов.

Рассмотрим пример:

```
public class Test {
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        for(int i=0; i < 10; i++){
            if(i % 2 == 0) continue;
            System.out.print(" i=" + i);
        }
    }
}
```

В результате работы на консоль будет выведено:

```
i=1 i=3 i=5 i=7 i=9
```

При выполнении условия в строке 7 нормальная последовательность выполнения операторов будет прервана и управление будет передано на начало цикла. Таким образом, на консоль будут выводиться только нечетные значения.

### Оператор `break`

Этот оператор, как и оператор `continue`, изменяет последовательность выполнения, но не возвращает исполнение к началу цикла, а прерывает его.

```

public class Test {
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int [] x = {1,2,4,0,8};
        int y = 8;
        for(int cnt=0;cnt < x.length;cnt++) {
            if(0 == x[cnt]) break;
            System.out.println("y/x = " + y/x[cnt]);
        }
    }
}

```

На консоль будет выведено:

```

y/x = 8
y/x = 4
y/x = 2

```

При этом ошибки, связанной с делением на ноль, не произойдет, т.к. если значение элемента массива будет равно 0, то будет выполнено условие в строке 9 и выполнение цикла `for` будет прервано.

В качестве аргумента `break` может быть указана *метка*. Как и в случае с `continue`, нельзя указывать в качестве аргумента *метки* блоков, в которых оператор `break` не содержится.

## Именованные блоки

В реальной практике достаточно часто используются вложенные циклы. Соответственно, может возникнуть ситуация, когда из вложенного цикла нужно прервать внешний. Простое использование `break` или `continue` не решает этой задачи, однако в Java можно именовать блок кода и явно указать операторам, к какому из них относится выполняемое действие. Делается это путем присвоения *метки* операторам `do`, `while`, `for`.

*Метка* - это любая допустимая в данном контексте лексема, оканчивающаяся двоеточием.

Рассмотрим следующий пример:

```

...
int array[][] = {...};
for(int i=0;i<5;i++) {
    for(j=0;j<4; j++) {
        ...
        if(array[i][j] == caseValue) break;
        ...
    }
}
...

```

В данном случае при выполнении условия будет прервано выполнение цикла по `j`, цикл по `i` продолжится со следующего значения. Для того, чтобы прервать выполнение обоих циклов, используется *метка*:

```

...
int array[][] = {...};
outerLoop: for(int i=0;i<5;i++) {
    for(j=0;j<4; j++){

```

```

        ...
        if(array[i][j] == caseValue)
            break outerLoop;
        ...
    }
}
...

```

Оператор `break` также может использоваться с именованными блоками.

Между операторами `break` и `continue` есть еще одно существенное отличие. Оператор `break` может использоваться с любым именованным блоком, в этом случае его действие в чем-то похоже на действие `goto`. Оператор `continue` (как и отмечалось ранее) может быть использован только в теле цикла. То есть такая конструкция будет вполне приемлемой:

```

lbl:{
    ...
    if( val > maxVal) break lbl;
    ...
}

```

В то время как оператор `continue` здесь применять нельзя. В данном случае при выполнении условия `if` выполнение блока с меткой `lbl` будет прервано, то есть управление будет передано на оператор (выражение), следующий непосредственно за закрывающей фигурной скобкой.

Метки используют пространство имен, отличное от пространства имен классов и методов.

Так, следующий пример кода будет вполне работоспособным:

```

public class Test {
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.test();
    }
    void test() {
        Test: {
            test: for(int i =0;true;i++) {
                if(i % 2 == 0) continue test;
                if(i > 10) break Test;
                System.out.print(i + " ");
            }
        }
    }
}

```

Для составления меток применяются те же синтаксические правила, что и для переменных, за тем исключением, что метки всегда оканчиваются двоеточием. Метки всегда должны быть привязаны к какому-либо блоку кода. Допускается использование меток с одинаковыми именами, но нельзя применять одинаковые имена в пределах видимости блока. Т.е. такая конструкция допустима:

```

lbl: {
    ...
    System.out.println("Block 1");
    ...
}

```

```
...
lbl: {
    ...
    System.out.println("Block 2");
    ...
}
```

А такая нет:

```
lbl: {
    ...
    lbl: {
        ...
    }
    ...
}
```

## Оператор return

Этот оператор предназначен для возврата управления из вызываемого метода в вызывающий. Если в последовательности операторов выполняется `return`, то управление немедленно (если это не оговорено особо) передается в вызывающий метод. Оператор `return` может иметь, а может и не иметь аргументов. Если метод не возвращает значений (объявлен как `void`), то в этом и только этом случае выражение `return` применяется без аргументов. Если возвращаемое значение есть, то `return` обязательно должен применяться с аргументом, чье значение и будет возвращено.

В качестве аргумента `return` может использоваться выражение

```
return (x*y +10) /11;
```

В этом случае сначала будет выполнено выражение, а затем результат его выполнения будет передан в вызывающий метод. Если выражение будет завершено ненормально, то и оператор `return` будет завершен ненормально. Например, если во время выполнения выражения в операторе `return` возникнет исключение, то никакого значения метод не вернет, будет обрабатываться ошибка.

В методе может быть более одного оператора `return`.

## Заключение

В данной лекции рассмотрены основные языковые конструкции.

Для организации циклов в Java предназначены три основных конструкции: `while`, `do`, `for`. Для изменения порядка выполнения операторов применяются `continue` и `break` (с меткой или без). Также существуют два оператора ветвления: `if` и `switch`.

## Пример из лекции:

```
package Exercises;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ex2 {
```

```

public static void main(String[] args)
{
    BufferedReader r = new BufferedReader(new
InputStreamReader(System.in));
    int n=1, numDigits=1; // переменная, в которую считываем
    try
    {
        System.out.println("Введите кол-во разрядов числа на билете: ");
        numDigits = Integer.parseInt(r.readLine());

        System.out.println("Введите " + numDigits + "-значное число на
билете: ");
        n = Integer.parseInt(r.readLine());
    }
    catch (NumberFormatException e)
    {
        //e.printStackTrace();
        System.out.println("Wrong format number");
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
//123456
if((n>Math.pow(10.0,numDigits-1))&&(n<Math.pow(10.0,numDigits)-1))
{

    int [] digits = new int[numDigits];
    for (int i=0; i<numDigits; i++)
        digits[i] = (n / (int)Math.pow(10.0, (double)i) ) % 10;
/* // Вариант с разложением строго на 6 разрядов
    int e, d, s, t, dt, st;
    e = (n / 1      ) % 10;
    d = (n / 10     ) % 10;
    s = (n / 100    ) % 10;
    t = (n / 1000   ) % 10;
    dt = (n / 10000 ) % 10;
    st = (n / 100000) % 10;
    //123321
    int n1 = e*100000 + d*10000 + s*1000 + t*100+dt*10+st;
    if (n1 == n)
        System.out.println("Число - палиндром!");
    else
        System.out.println("Число - не палиндром!");
*/
    // System.out.println(e + " | " + d + " | " + s + " | " + t + "
| " + dt + " | " + st);
    // if ((e + d + s) == (t + dt + st))
    if ((digits[0] + digits[1] + digits[2]) == (digits[3] +
digits[4] + digits[5]))
        System.out.println("Билет счастливый: можно есть!");
    else
        System.out.println("Билет обычный: не есть!");
}
else
{
    System.out.println("Это не " + numDigits + "-значный номер
билета");
}
}

```

```
/* // Ещё один пример
   for (int a=1;a<=10;a++)
   {
       System.out.println(a*a + " | " + a*a*a);
   }
*/
}
```