Программирование на Java

Лекция 10. Операторы и структура кода. Исключения

20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>

Аннотация

После ознакомления с типами данных в Java, правилами объявления классов и интерфейсов, а также с массивами, из базовых свойств языка остается рассмотреть лишь управление ходом выполнения программы. Вводятся важные понятия, связанные с этой темой, описываются метки, операторы условного перехода, циклы, операторы break и continue и другие.

Следующая тема посвящена более концептуальным механизмам Java, а именно работе с ошибками, или исключительными ситуациями. Рассматриваются причины возникновения сбоев, способы их обработки, объявление собственных типов исключительных ситуаций. Описывается важное разделение всех ошибок на проверяемые и непроверяемые компилятором, а также ошибки времени исполнения.

Оглавление

Лекция 10. Операторы и структура кода

Содержание лекции.

1. Управление ходом программы

Управление потоком вычислений является фундаментальной основой всего языка программирования. В данной главе 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, вместе с тем имеется ряд отличий, которые будут рассмотрены позднее и на которые следует обратить внимание.

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

2. 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

```
brake
continue
return
```

то управление будет передано в другое место (в соответствии с правилами обработки этих операторов, которые будут рассмотрены позже).

Нормальное выполнение оператора может быть прервано, так же, при возникновении исключительных ситуаций. Которые так же будут рассмотрены позже. Явное возбуждение исключительной ситуации с помощью оператора throw, так же 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 Exception, а так же все исключения возбуждаемые виртуальной машиной Java.

Выражения так же могут завершаться нормально и преждевременно (аварийно). В данном случае термин аварийно вполне применим, т.к. причиной последовательности выполнения выражения отличной от нормальной 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 случае если в операторе имеется вложенный оператор, и происходит ненормальное его завершение, то так же не нормально завершается оператора содержащего вложенный (в некоторых случаях это не так, но будет оговариваться особо)

3. 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.

При обращении к локальным переменным не может быть использован квалификатор this или имя класса.

```
1. public class Test {
2.     static int x;
3.     public Test() {
4.     }
5.     public static void main(String[] args) {
6.       Test t = new Test();
7.       lbl: {
8.           int x = this.x;
9.           if ( x > 0) break lbl;
10.       }
11.     }
12. }
```

В строке 18 мы получим ошибку компиляции. Если строку заменить на int x = x;, то тоже будет получена ошибка компиляции. (т.к. локальная переменная не инициализирована переед первым использованием). А вот такое решение будет рабочим int x = (x=2)x;

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

Нельзя объявлять несколько локальных переменных 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. Например, этот 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); } }

И на консоль будет выведено X = 1. Следует напомнить, что перекрытие локальными переменными области видимости глобальных переменных является, частой, но трудно обнаруживаемой ошибкой.

То же самое правило применимо к фактическим параметров 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 следующее.

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

На следующем примере продемонстрируем, что область видимости локальной переменной ограничено 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 переменой, то выражение исполняется слева направо и его результат присваивается локальной переменной. Использование не инициализированных 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, когда 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)

4. Пустой оператор

; Является пустым оператором. Данная конструкция вполне применима там, где не предполагается выполнение никаких действий. Преждевременное завершение пустого оператора невозможно.

5. 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, то метки внешних блоков будут видимы во внутренних.

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 метод goto отсутствует. В некоторых ситуациях (как в рассмотренном примере со вложенными циклами) метки необходимы, но, конечно, их применение следует ограничивать лишь самыми необходимыми случаями.

6. Оператор if

Пожалуй, наиболее часто встречающейся 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. Заметим отличие от языка C, в котором в качестве логического выражения может быть использованы различные типы данных, где отличное от нуля выражение трактуется как истинное значение, а ноль как ложное. В Java возможно использование только логических выражений.

В случае 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") является вполне допустимой.

Операторы if-else могут каскадироваться. . Иногда это называют многозвенный if else.

```
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 в том случае, если не выполнено предыдущее. Заключительная конструкция else относится к самому последнему условию if b,и будет выполнено только в том случае, если ни одно из вышеперечисленных условий не будет истинным. В случае если 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 4");
} else{
 System.out.println("Другое значение");
}
```

Предложение "Равно 4" в данном случае напечатано не будет.

Следует обратить внимание на то, что в условии могут быть использованы только логические выражения.

Например

```
int x = 0;
if(x) ….
```

вызовет ошибку компиляции.

Выражение вот такого типа тоже будет ошибочным

```
int x = 0;
if( x = 5) ….
```

так как здесь происходит не операция сравнения а присвоение значения.

Следует заострить внимание на коротком пути (short circuit) вычисления логических выражений рассмотренных ранее.

В качестве полезного примера можно привести следующий

```
if( null == stringVal  || "" == stringVal )
 System.out.println("Значение stringVal не определено")
```

Данная конструкция выполнится успешно.

```
if( null == stringVal  | "" == stringVal )
 System.out.println("Значение stringVal не определено")
```

В этом случае мы получим ошибку времени выполнения, если stringVal,будет иметь пустое (null) значение.

Следует обратить внимание так же на то, что константы используетсяв левой части оператора сравнения. Если мы будем сравнивать переменную с булевской константой и допустим опечатку

```
if( x = true)
 …
```

то такая конструкция будет всегда истинна, что возможно отличается от того, что этой конструкцией хотел выразить программист. (Такого рода ошибки, могут обнаруживаться с помощью программ автоматического тестирования)

В этом же случае

```
if( true = x)
```

компилятор выдаст ошибку еще во время компиляции.

7. Оператор switch

Оператор switch() 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 быть использована 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 производится выполнение выражения следующего за этим условием. Если выражение выполнено нормально, и нет преждевременного его завершения, то производится выполнение сравнения для последующих case. Если же выражение, следующее за case, завершилось не нормально, 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 выполнено не будет.

Если какое либо условие case выполнено, то все выполнение switch не прекратится, а будут проверяться следующие за ним условия. Что бы избежать этого, после части кода, которая выполнена, и дальнейшее выполнение не является необходимым, применяется break.

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.

Наиболее часто встречается ошибка, когда программист забывает указать break после выполнения блока кода и производится дальнейшее выполнение условий сравнения в операторе switch(). Следует обратить на это внимание. В качестве хорошего стиля программирования можно порекомендовать использование оператора default.

Рассмотрим пример

```
int x = 2;
switch(x){
 case 1:
 case 2:
  System.out.println("Равно 1 или 2");
  break;
 case 2:
 case 3
  System.out.println("Равно 2 или 3");
  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:
  …
  break;
}
```

Если в операторе switch() применяется выражение, следует обратить внимание на результирующий тип выражения. Например

```
float x = 1.0f;
int y = 4;
switch(x*y){
 case 10:
  …
  break;
 case 20:
  …
  break;
 default:
  …
}
```

вызовет ошибку времени компиляции.

Следует обратить внимание так же на следующий ньюанс. Если в операторе switch() указано значение типа byte или short, то соответственно в операторах case должны применяться константы, которые могут быть сохранены в переменной данного типа. Например:

```
byte x = 5;
switch(x){
 case 1:
  …
  break;
 case 132:
  …
  break;
 default:
  …
}
```

вызовет ошибку компиляции. Так как case 132 превышает максимальное значение, которое может быть сохранено в переменной типа byte; Оператор default не обязательно должен замыкать конструкцию switch case. Он может так же комбинироваться с любым оператором case

Например

```
public class Test {
    static int x = 5;
    static {

    }
    public Test() {
    }
    public static void main(String[] args) {
```

```
        Test t = new Test();
        int x = 5;
        switch(x){
            case 1:
                System.out.println("One");
                break;
            case 2:
                System.out.println("Two");
                break;
            default:
            case 3:
                System.out.println("Tree or other");
        }
    }
}
```

откомпилируется без ошибок и на консоль будет выведено

```
Tree or other
```

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");
}
```

не допустима.

Так же в конструкции switch может быть применен только один оператор default.

Можно порекомендовать использование оператора switch вместо многозвенного if else, т.к. switch выполняется быстрее.

8. Управление циклами

В языке Java имеется три основных конструкции управления циклами.

- цикл while
- цикл do
- цикл for

8.1. 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.

Если выражение или блок представляющий тело цикла будет завершен не нормальным образом по причине

- встретился оператор continue, то часть тела цикла следующая за оператором continue будет пропущена и выполнение цикла продолжится с начала. Если continue используется с меткой и метка принадлежит к данному while, то выполнение его будет аналогичным. Если метка не относится к данному while, то его выполнение будет прекращено.
- встретился оператор break, то выполнение цикла будет прекращено
- если выполнение блока будет прекращено по другим причинам (возникла исключительная ситуация), то выполнение while будет прекращено по тем же причинам.

Рассмотрим несколько примеров

```
public class Test {
    static int x = 5;
    static {

    }
    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;
    static {

    }
    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 продолжится с начала.

Советом по применению конструкции может служить использование фигурных скобок, даже если выражение следующее после while(), будет единственным. Т.о. если конструкция в дальнейшем будет расширяться, то можно избежать потенциальных ошибок, т.к. если фигурные скобки опустить, выражение следующее за первым будет трактоваться как независимое, а не связанное с условием while()

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, когда на момент начала его выполнения, логическое выражение будет истинным. Т.о. 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.

8.2. Цикл do

Основная форма цикла do имеет следующий вид

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

В отличии от цикла while цикл do, будет выполняться до тех пор, пока логическое выражение будет ложным. Вторым важным отличием является то, что do будет выполнен как минимум один раз.

Следует еще раз обратить внимание на использование фигурных скобок. Так же следует подчеркунть, что условие выхода из уикла должно изменяться в самом цикле, в противном случае, однажды начавшись, цикл не будет закончен никогда. То же самое следует отметить для цикла while(). В качестве примера следует привести следующую конструкцию.

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

Типичная конструкция цикла do

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

В остальном выполнение цикла do совершенно аналогично выполнению цикла while.

8.3. Цикл for

Довольно часто, необходимо изменять значение какой-либо переменной в заданном диапазоне, и выполнять повторяющуюся последовательность операторов с использованием этой переменной. Для выполнения этой 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 круглых скобках 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 loop(). Тело цикла будет выполняться до тех пор, пока логическое выражение будет истинным. Как и в случае с циклом while() тело цикла может не исполниться ни разу. Это условие срабатывает, если логическое выражение принимает значение ложь до начала выполнения цикла.
- выражение - 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.

Допускается 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 следует обратить внимание, что область видимости переменной cnt будет ограничена телом цикла. Следует помнить о диапазоне значений переменной счетчика.

В качестве примера можно привести следующую конструкцию:

```
for(int I = 0;int < 10;i+=){
 System.out.println("Value i = " + i);
}
System.out.println("After loop i value = " + i);
```

В данном случае воникнет ошибка времени компиляции.

```
for(byte x = 0; x < 256;x++){
```

```
    ...
}
```

А такая конструкция будет выполняться бесконечно … Максимальное значение x = 127. После превышения этого значения x ,будет присвоено значение -128 и т.о. значение 256 никогда достигнуто не будет.

Следует обратить внимание на то, что предложение будет выполнено в любом случае, будет выполнено тело цикла или нет. В качестве примера можно привести такую конструкцию

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

На выхоже будет получено Counter is 0

Напротив выражение выполняется только при выполнении тела цикла.

```
int counter = 0;
for(;cnt < 1;counter++){
    ...
}
System.out.println("Counter is " + counter);
```

На выходе будет получено Counter is 1

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, будут рассмотрены позже.

Возможно так же 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.

Следует отметить, что при использовании в конструкции for() нескольких частей, невозможно определение нескольких переменных или смешение определения и инициализации

нескольких переменных. Так при попытке использования следующих выражений будет получена ошибка компиляции.

```
for(int i = 0, long j = 0,i<10; i++, j += 50) // неверно
```

Нельзя так же использовать выражения в предложении

```
…
int i = 0;
for(i++; int j = 0; i< 10; j++)    // неверно
   ...
```

однако такая конструкция будет верна

```
...
int j = 0;
for(int = 7, j = 0 ;i < 10; i++,j+=10)
   ...
```

9. Операторы break и continue

В некоторых случаях требуется изменить ход выполнения программы. В традиционных языках программирования для этих целей используется оператор goto, однако в Java его использование не предусмотрено. Для этих целей применяются операторы break и continue

9.1. Оператор continue

Оператор continue может применяться 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.

Следует обратить внимание, что инициализация переменной cnt не произойдет, а будет произведена следующая итерация. Если оператор continue будет применен не в контексте оператора цикла, то будет выдана ошибка времени компиляции.

Рассмотрим пример

```
1. public class Test {
2. public Test() {
3. }
4. public static void main(String[] args) {
5. Test t = new Test();
6.   for(int j=0; j < 10; j++){
7.     if(i* % 2 == 0) continue;
8.        System.out.print("i=" + i);
9.      }
10.   }
11. }
```

в результате работы на консоль будет выведено

```
1 3 5 7 9
```

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

9.2. Оператор break

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

```
1. public class Test {
2.     public Test() {
3.     }
4.     public static void main(String[] args) {
5.         Test t = new Test();
6.         int [] x = {1,2,4,0,8};
7.         int y  = 8;
8.         for(int cnt=0;cnt < x.length;cnt++){
9.             if(0 == x[cnt]) break;
10.             System.out.println("y/x = " + y/x[cnt]);
11.         }
12.     }
13. }
```

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 данный оператор break не содержится.

10. 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 разрешает этой задачи, однако в Java возможно именовать блок кода и явно указать этим операторам, к какому из них относится делаемое действие. Делается 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, что бы прервать выполнение обоих циклов, используется ранее упомянутая языковая конструкция

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

Оператор continue так же может быть использован с именованными блоками.

Следует обратить внимание, что при использовании перехода continue на внешний блок, его действие схоже с действием break. Т.е. исполнение текущего блока будет прервано и управление передастся на внешний блок.

Пример.

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

и

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

Результат исполнения обоих вариантов кода будет идентичным

Между операторами break и continue есть еще одно существенное отличие. Оператор break может быть использован 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. Т.е. 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, т.е. управление будет передано на оператор (выражение), следующий непосредственно за закрывающей фигурной скобкой.

Следует отметить, что использование оператора goto JAVA запрещено, хотя это слово и является зарезервированным. Метки используют пространство имен, отличное от пространства имен классов, и методов.

Например, этот пример кода будет вполне работоспособным.

```
1.
2. public class Test {
```

```
3.      public Test() {
4.      }
5.      public static void main(String[] args) {
6.          Test t = new Test();
7.          t.test();
8.      }
9.      void test(){
10.          Test:{
11.              test: for(int i =0;true;i++){
12.                  if(i % 2 == 0) continue test;
13.                  if(i > 10) break Test;
14.                  System.out.print(i + " ");
15.              }
16.          }
17.      }
18. }
```

Однако следует акцентировать внимание, что хотя пространства имен и не совпадают, не рекомендуется без крайней на то необходимости использовать имена меток совпадающие с именами методов или классов.

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 использовать 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:{
  …
 }
 …
}
```

11. Оператор return

Этот оператор предназначен для возврата управления из вызываемого метода в вызываемый. Если в последовательности операторов выполняется return то управление немедленно (если это не оговорено особо) передает управление в вызывающий метод.

Далее будут рассмотрены особенности выполнения retun в конструкции try catch finally

return может иметь, а может не иметь аргументов. Если аргументы отсутствуют, то этот оператор может быть использован для возврата управления только в методах с квалификатором void. Если при объявлении метода использован тип void, то return не может иметь аргументов, в противном случае, будет получена ошибка времени компиляции.

В качестве аргумента return может быть использовано выражение.

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

В этом случае сначала будет выполнено выражение, а затем результат его выполнения будет передан в вызывающий метод. В случае если выражение будет завершено не естественным образом, то и оператор return, будет завершен не естественным способом. Например, если во время выполнения выражения в операторе return возникнет исключение, то и return не будет выполнен так, как это ожидалось. Т.е. исключительная ситуация будет обработана в самом методе, или будет возбуждена в вызывающем методе.

В методе может быть более одного оператора return.

12. Оператор synchronized

Этот оператор применяется для исключения взаимного влияния нескольких потоков при выполнении кода, и будет подробно рассмотрен в главе 12, посвященной потокам исполнения.

13. Ошибки при работе программы. Исключения (Exceptions).

При выполнении программы зачастую могут возникать ошибки. В одних случаях это вызвано ошибками программиста, в других внешними причинами. Например, может возникнуть ошибка ввода/вывода при работе с файлом или сетевым соединением. В классических языках программирования, например в C, требовалось проверять некое условие которое указывало на наличие ошибки и, в зависимости от этого предпринимать определенные действия.

Например

```
…
int statusCode = someAction();
if (statusCode){
 … обработка ошибки
}else{
```

```
 statusCode = anotherAction();
 if(statusCode){
  … обработка ошибки …
}
}
…
```

В Java появилось более простое и элегантное решение - обработка исключительных ситуаций.

```
try{
 someAction();
 anotherAction()
}catch(Exception e){
 … обработка исключительной ситуации
}
```

Легко заметить, что такой подход является не только элегантным, но и более надежным и простым для понимания.

13.1. Причины возникновения ошибок

Существует три причины возникновения исключительных ситуаций.

- Попытка выполнить некорректное выражение.

  Например, деление на ноль, или обращение к объекту по ссылке, равной null, попытка использовать класс, описание которого (class-файл) отсутствует, и т.д.

  В таких случаях всегда можно точно указать, в каком месте произошла ошибка - именно в некорректном выражении.

- Выполнение оператора throw.

  Очевидно, что и здесь можно легко указать место возникновения исключительной ситуации.

- Асинхронные ошибки во время исполнения программы.

  Причиной таких ошибок могут быть сбои внутри самой виртуальной машины (ведь она также является программой), или вызов метода stop() у потока выполнения thread).

  В этом случае невозможно указать точное место программы, где происходит исключительная ситуация. Если мы пытаемся остановить поток выполнения (вызвав метод stop()), то мы не можем предсказать, при выполнении какого именно выражения этот поток остановится.

Таким образом, все ошибки в Java делятся на синхронные и асинхронные. Первые сравнительно проще, так как принципиально возможно найти точное место в коде, которое является причиной возникновения исключительно ситуации. Конечно, Java является строгим языком в том смысле, что все выражения до точки сбоя обязательно будут выполнены, в то же время ни одно последующее выражение никогда выполнено не будет. Важно помнить, что ошибки могут возникать как по причине недостаточной внимательности программиста (отсутствует нужный класс, или индекс массива вышел за допустимые границы), так и по

независящим от него причинам (произошел разрыв сетевого соединения, сбой аппаратного обеспечения, например, жесткого диска, и др.).

Асинхронные ошибки гораздо сложнее в обнаружении и исправлении. Обычному разработчику очень затруднительно выявить причины сбоев в виртуальной машине. Это могут быть ошибка создателей JVM, несовместимость с операционной системой, аппаратный сбой и многое другое. Все же современные виртуальные машины реализованы довольно хорошо, и подобные сбои происходят крайне редко (при условии использования качественных комплектующих).

Аналогичная ситуация наблюдается и в случае с принудительной остановкой потоков исполнения. Поскольку это действие выполняется операционной системой, никогда нельзя предсказать, в каком именно месте остановится поток. Это означает, что программа может многократно отработать корректно, а потом неожиданно дать сбой просто из-за того, что поток остановился в каком-то другом месте. По этой причине принудительная остановка крайне не рекомендуется. В соответствующей лекции приводятся примеры корректного управления жизненным циклом потока.

При возникновении исключительной ситуации управление передается от кода, вызвавшего исключительную ситуацию, на ближайший блок catch (или вверх по стеку), и создается объект, унаследованный от класса Throwable или его потомков (см. диаграмму иерархии классов исключений), который содержит информацию об исключительной ситуации и используется при ее обработке. Собственно в блоке catch указывается именно класс обрабатываемой ситуации. Подробно обработка ошибок рассматривается ниже.

Иерархия по которой передается информация об исключительной ситуации зависит от того, где эта исключительная ситуация возникла. Если это

- метод, то управление будет передаваться в то место, где этот метод был вызван;
- конструктор, то управление будет передаваться туда, где попытались создать объект (как правило, применяя оператор new);
- если это статический инициализатор, то управление будет передано туда, где произошло первое обращение классу, потребовавшее его инициализацию.

Допускается создание собственных классов исключительных ситуаций. Осуществляется это с помощью механизма наследования, т.е. класс пользовательской исключительной ситуации, должен быть унаследован от класс Throwable или его потомков.

13.2. Обработка исключительных ситуаций

13.2.1. Конструкция try-catch

В общем случае конструкция выглядит так.

```
try{
 …
}catch(SomeExceptionClass e){
 …
}catch(AnotherExceptionClass e){
 …
}
```

Работает она следующим образом. Сначала выполняется код заключенный в фигурные скобки оператора try. Если во время его выполнения не происходит никаких нештатных ситуаций, то далее управление передается, за закрывающую фигурную скобку, последнего оператор catch ассоциированного с данным оператором try.

В случае если в пределах try возникает исключительная ситуация, то далее выполнение кода производится по одному из ниже перечисленных сценариев.

- возникла исключительная ситуация, класс которой указан в качестве параметра одного из блоков catch. В этом случае производится выполнение блока кода ассоциированного с этим catch (заключенного в фигурные скобки). Далее если код в этом блоке завершается нормально, то и весть оператор try завершается нормально и управление передается на оператор (выражение) следующий за закрывающей фигурной скобкой последнего catch ассоциированного с данным try. Если код в catch завершается нештатно, то и весь try завершается нештатно по той же причине.
- если возникла исключительная ситуация, которая класс которой не указан в качестве аргумента, ни в одном catch, то выполнение всего try завершается нештатно.

Если в последовательности операторов могут возникнуть как ошибки ввода/вывода так и ошибки арифметических вычислений, вовсе нет нужды помещать различные фрагменты кода в разные операторы try{}catch(){}. Достаточно обеспечить несколько catch() для различных типов исключений.

13.2.2. Конструкция try-catch-finally

Оператор finally предназначен для того, что бы обеспечить гарантированное выполнение какого-либо фрагмента кода.

Последовательность выполнения такой конструкции будет следующей: Если оператор try выполнен нормально, то будет выполнен блок finally. В свою очередь, если оператор finally выполняется нормально, то весь оператор try выполняется нормально. Если происходит преждевременное окончание выполнения блока finally, то весть оператор try завершается предварительно по тем же причинам.

- существует оператор catch, который перехватывает данный тип исключения, происходит выполнение связанного с catch блока.
  - Если блок catch выполняется нормально, то выполняется блок finally
    - в свою очередь если блок finally завершается нормально, то весь try завершается нормально.
    - Если finally завершается предварительно, то и весь оператор try завершается предварительно по той же причине.
  - Если блок catch завершается ненормально, то выполняется блок finally
    - в свою очередь, если блок finally завершается нормально, то оператор try завершается не нормально, по той же причине, по которой не нормально завершился блок catch
    - если блок finally завершается ненормально, то весь блок try завершается ненормально, по той же причине, что и блок finally

- в списке операторов catch не находится такого, который обработал бы возникшее исключение. Все равно выполняется блок finally. В этом случае, если
  - finally завершится нормально, весь try завершится не нормально по той же причине по которой было нарушено исполнение try.
  - finally завершится ненормально, то try завершится ненормально по той же причине, по которой ненормально завершился finally

Если оператор try завершился нормально, то выполнится блок finally. И если

- блок finally завершится нормально, то весь try завершится нормально
- блок finally завершится не нормально, то весь try завершится не нормально, по той же причине.

Следует обратить внимание, что при использовании конструкции finally, блок кода ассоциированный с ним будет выполнятся всегда. Если во время обработки исключительной ситуации, возникнет новая исключительная ситуация, то исключительная ситуация, которая послужила первопричиной будет потеряна.

Рассмотрим пример применения конструкции try-catch-finally.

```
try{
 byte [] buffer = new byte[128];
 FileInputStream fis = new FileInputStream("file.txt");
 while(fis.read(buffer) > 0){
  … обработка данных
}
}catch(IOException es){
  … обработка исключения …
}finally{
 fis.flush();
 fis.close();
}
```

Следует обратить внимание, что использование flush() не является обязательным в данном контексте, т.к. буфер ввода/вывода будет очищен при вызове close() указания, и здесь использован для большей наглядности.

Если в данном примере поместить операторы очистки буфера и закрытия файла сразу после окончания обработки данных, то при возникновении ошибки ввода вывода, корректного закрытия файла не произойдет. Следует отметить, что блок finally, будет выполнен в любом случае, вне зависимости от того произошла обработка исключения или нет, возникло это исключение или нет.

В конструкции try-catch-finally обязательным является использование одной из частей оператора catch или finally. То есть, конструкция

```
try{
 ...
}finally{
```

```
 ...
}
```

является вполне допустимой. В этом случае блок finally при возникновении исключительной ситуации будет выполнен, хотя сама исключительная ситуация обработана не будет и будет передана для обработки на более высокий уровень иерархии.

Следует рассмотреть два специальных случая использования finally.

```
try{
 …
}catch(Exception e){
 …
 System.exit(0);
}finally{
 …
}
```

в этом случае блок finally выполнен НЕ БУДЕТ т.к. выполнение данного потока будет прекращено.

Пример применения оператора return:

```
try{
 …
 return 0;
}
catch(MyException ex){
 …
 System.out.println("Exception");
 return -1;
}
finally{
 …
 System.out.println("Finally");
}
```

В этом случае последовательность действий будет следующей.

Если исключения не произойдет будет исполнен лишь блок finally метод вернет значение 0

На консоль будет выведено Finally

Если произойдет исключение MyException, то будет выполнен блок finally и метод вернет значение -1.

На консоль будет выведено

```
Exception
Finally
```

То, есть сначала отработает блок finally и только после этого будет произведен возврат управления в вызвавший код.

Если,будет возбуждено исключение Exception, которое не обрабатывается в данной конструкции, то будет выполнен блок finally и исключительная ситуация будет передана на обработку вызываемому коду.

В случае, если обработка исключительной ситуации в коде не предусмотрена, то при ее возникновении выполнение метода будет прекращено, и исключительная ситуация будет передана для обработки коду более высокого уровня. Таким образом, если исключительная ситуация произойдет в вызываемом методе, то управление будет передано вызываемому методу и обработку исключительной ситуации должен произвести он. Если исключительная ситуация возникла в коде самого высокого уровня (методе main()), то управление будет передано исполняющей системе Java и выполнение программы будет прекращено.

13.3. Использование оператора throw

Помимо того, что предопределенная исключительная ситуация могла быть возбуждена исполняющей системой Java, программист сам может сгенерировать это условие. Делается это с помощью оператора throw.

Например

```
….
public int  calculate(int theValue){
if( theValue < 0){
throw new Exception("Параметр для вычисления не должен быть отрицательным");
}
}
…
```

В данном случае, предполагается, в качестве параметра методу может быть передано только положительное значение, если это условие не выполнено, то с помощью оператора throw возбуждается исключительная ситуация. В действительности данный код не будет откомпилирован, т.к. компилятор выдаст сообщение об ошибке. Если в методе возбуждается исключительная ситуация, то должно быть выполнено одно из двух правил

- исключительная ситуация должна быть обработана в теле метода (т.е. код должен возбуждающий исключительную ситуацию, должен быть помещен в блок try {} catch(UserException ue){})
- метод должен делегировать обработку исключительной ситуации вызвавшему его коду. Для этого в сигнатуре метода применятся ключевое слово throws, после которого должны быть перечислены через запятую все исключительные ситуации, которые может вызывать данный метод. Т.е. приведенный выше пример должен быть приведен к следующему виду

```
….
public int  calculate(int theValue) throws Exception{
if( theValue < 0){
throw new Exception("Some descriptive info");
```

```
}
}
…
```

Т.о. возбуждение исключительной ситуации в программе производится с помощью оператора throw, слева от которого указывается объект, который может быть приведен к типу Throwable. (Как правило этот объект создается в этом же месте с помощью оператора new, хотя это условие и не является обязательным)

В некоторых случаях после обработки исключительной ситуации, возможно, возникнет необходимость передать информацию о ней в вызывающий код.

В этом случае throw используется вторично.

Например

```
…
try{
 …
}catch(IOException ex){
 …
 // Обработка исключительной ситуации
 …
 // Повторное возбуждение исключительной ситуации
 throw ex;
}
```

Рассмотрим еще один случай.

Предположим, что оператор throw применяется внутри конструкции try catch.

```
try{
 …
 throw new IOException();
 …
}catch(Exception e){
 …
}
```

В этом случае, исключение возбужденное в блоке try не будет передано для обработки на более высокий уровень иерархии, а обработается в пределах блока try catch., т.к. тут содержится оператор, который может это исключение перехватить.Т.е. как бы произойдет неявная передача управления, на соответствующий блок cacth

Следует обратить внимание слушателей, что такой способ использования конструкции try catch не рекомендуется, т.к. затраты на обработку будут несравнимо выше, чем при использовании операторов if. Следует обратить так же внимание на то, что блок catch может быть пустым, т.е. не производить никакой обработки, тем не менее исключение будет считаться перехваченным и далее по иерархии передано не будет.

```
try{
```

```
 …
 throw new IOException();
 …
}catch(Exception e){
 ;
}
```

в данном случае оператор try завершится нормально.

13.4. Обрабатываемые и необрабатываемые исключения

Все исключительные ситуации можно разделить на две категории обрабатываемые (checked) и не обрабатываемые (unchecked).

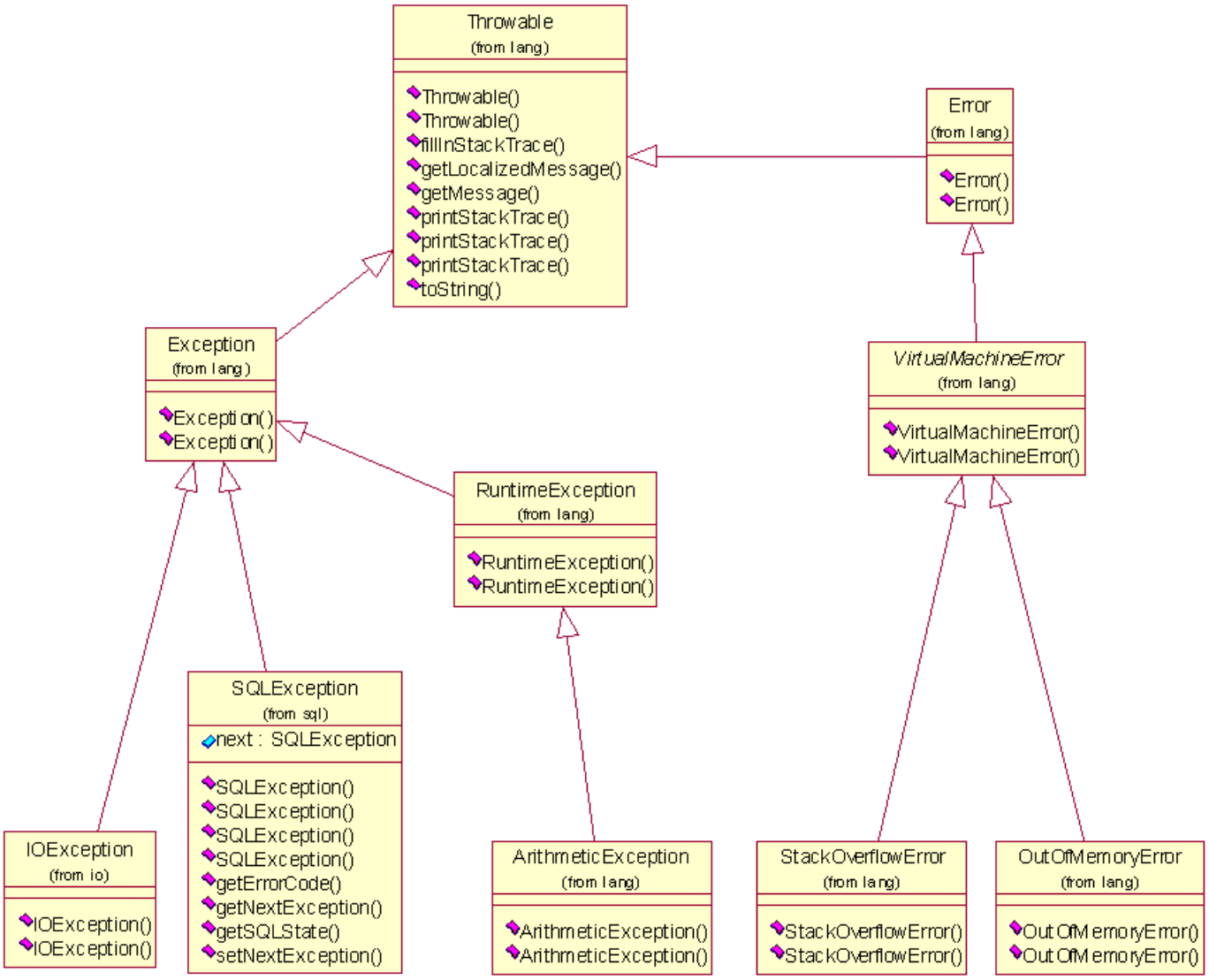
Все исключения, порожденные от java.lang.Exception являются обрабатываемыми. Т.е. во время компиляции проверяется - предусмотрена ли обработка возможных исключительных ситуаций.

Исключения, порожденные от java.lang.RunTimeException, являются необрабатываемыми, и компилятор не требует обязательной их обработки.

Как правило, обрабатываемые исключения предназначены для обработки ситуаций связанных с окружением программы (сетевым, файловым вводом-выводом и др.), которые могут возникнуть вне зависимости от того, корректно написан код или нет. Например, открытие сетевого соединения или файла может привести к возникновению ошибки, и компилятор требует от программиста предусмотреть некие действия для обработки возможных проблем. (Возможно никаких действий предпринято не будет, т.е. блок catch(){} можно оставить пустым, однако компилятор это трактует как обработку исключения и не выдаст сообщения об ошибке компиляции) Все пользовательские исключения, порожденные от java.lang.Exception (или его потомков) являются обрабатываемыми.

Необрабатываемые исключения, это ошибки программы, которые при правильном кодировании возникать не должны (например, java.lang.IndexOfBoundException, java.lang.ArifmeticException возникают соответственно при указании индекса выходящего за границы массива и при делении на ноль). Поэтому, чтобы не загромождать программу, компилятор разрешает не обрабатывать с помощью блоков try{} catch() исключения этого типа.

Исключения, порожденные от Error, так же не являются обрабатываемыми. Эти ошибки предназначены для того что бы уведомить программу о возникновении сбоев которые программным способом устранить сложно, или невозможно вообще. В качестве примера можно привести StackOverflowError, OutOfMemoryError.



Следует обратить внимание, что хотя необрабатываемые ошибки не обязательно помещать в блоки try{} catch() …, они обрабатываются точно так же как и обычные ошибки. Т.е. если вы не можете гарантировать, что Вов время выполнения не будет нарушена граница массива (например при динамическом определении размера массива), то код в котором производится обращение к элементам массива следует поместить в блок try{} catch(java.lang.IndexOfBoundException ex) { …}.

Далее, в случае если в подпрограмме могут возникать необрабатываемые ошибки, то указание ключевого слова throws не является обязательным.

Ошибки (Error), говорят о наличии фатальных ситуаций и нет необходимости обрабатывать их. Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые ошибки виртуальной машины Java. Например, исчерпание свободной памяти, переполнение стека и т.д.

В разработке и проектировании программного обеспечения используется понятие контракта. Более полно это понятие рассматривалось ранее. Одним из аспектов является четкое разграничение ответственности между вызывающим и вызываемым методами. Например при вызове метода производится вычисление квадратного корня (предполагается, что речь идет о натуральных числах), тогда на вход метода не может быть передано отрицательное число. Если это происходит, то метод должен вызвать исключительную ситуацию, так как контракт нарушен. Этот подход наиболее понятен и упрощает программирование, так как метод должен обрабатывать только правильно заданные параметры и не должен знать ничего об обработке неправильных. Кроме того он должен уведомить вызывающий метод о возникновении этой ситуации, сразу по ее возникновению. Для реализации этого подхода рекомендуется использовать обрабатываемее (checked) исключения.

Как было сказано ранее, возможно задание нескольких операторов catch для одного блока try. Если программист желает перехватывать некий набор исключительных ситуаций, он может указать в блоке catch родительский класс вместо перечисления нескольких подклассов. Например можно перехватывать все исключения порожденные от java.lang.Exception следующим образом

```
  try{
  …
 }
 catch(Exception e){
  …
 }
```

Так как родителем всех исключений является java.lang.Throwable, то если необходимо в каком - либо месте кода перехватывать все исключительные ситуации, то вполне применима вот такая конструкция

```
  try{
  …
 }
 catch(Throwable e){
  …
 }
```

однако следует избегать использования такой конструкции без явных на то причин. Т.к. в блоке catch будут обрабатываться самые разные ошибки, что нарушит структуру кода и усложнит его понимание.

В случае если в конструкции обработки исключений используется несколько операторов catch, то классы исключений нужно перечислять в них последовательно, от менее общих к более общим. Рассмотрим два примера

```
try{
 …
}
catch(Exception e){
 …
```

```
}
catch(IOException ioe){
 …
}
catch(UserExcetion ue){
 …
}
```

В данном примере при возникновении исключительной ситуации (класс которой порожден от Exception) будет выполняться всегда только первый блок catch. Остальные не будут выполнены ни при каких условиях. Эта ситуация отслеживается компилятором, который сообщает об UnreachableCodeException (ошибка - недостижимый код). Правильно данная конструкция будет выглядеть так

```
try{
 …
}
catch(UserExcetion ue){
 …
}
catch(IOException ioe){
 …
}
catch(Exception e){
 …
}
```

В этом случае будет произведена последовательная обработка исключений. И в случае если не предусмотрена обработка того типа исключения, которое возникло (например, AnotherUserException), будет выполнен блок catch(Exception e){…}

Если срабатывает один из блоков catch, то остальные блоки в данной конструкции try-catch выполняться не будут.

13.5. Создание пользовательских классов исключений

Как уже отмечалось ранее, допускается создание собственных классов исключений. Для этого достаточно создать свой класс, унаследовав его от любого класса являющегося дочерним по отношению к ajav.lang.Throwable. (Или от самого Throwable)

Пример.

```
  public class UserException extends Exception{
  public UserException(){
   super();
  }
  public UserException(String descry){
super(descr);
}
 }
```

соответственно возбуждаться данное исключение будет следующим образом:

```
  throw new UserException("Дополнительное описание");
```

Рассмотрим другой пример. В языке С существует конструкция assert, которая зачастую используется для целей отладки. Рассмотрим как в JAVA можно воспроизвести подобную конструкцию с помощью механизма исключений.

```
public class AssertionException extends RuntimeException{

public AssertionException(){
super("Asertion Exception");
}

  public AssertionException(String descr){
super(descr);
}
}

public class Assertion {
 public static Boolean ASSERTION_ON = true;

 private Assertion(){};

public static void assert(boolean flag)
throws AssertionException{
if(ASSERTION_ON && flag){
 throw new AssertionException()
}
}

public static void assert(boolean flag,Sring msg)
throws AssertionException{
if(ASSERTION_ON && flag){
 throw new AssertionException(msg)
}
}

}
```

Основная идея использования данного класса заключается в том, что бы в критичных участках программы встроить проверку некоторых граничных условий и, в случае их невыполнения возбуждать исключительную ситуацию.

Например

```
Assertion.assert(x <= xMinValue,"X too large");
```

В данном случае, если величина переменной x будет меньше некоего минимума будет возбуждена исключительная ситуация AssertionException. Т.к. это необрабатываемое

исключение, то использование блока try{} catch() не является обязательным. Однако, если мы все таки обработаем эту исключительную ситуацию, то можем выдать (например на консоль, или в лог-файл) сообщение об ошибке.

```
try{
 …
 // вызов кода использующего Assertion
 …
}cacth(AssertionException ae){
 System.err.println(ae);

}
```

13.6. Переопределение методов и исключения

При переопределении методов следует помнить что, если переопределяемый метод возбуждает исключение, то переопределяющий метод не может расширять класс этих исключений. Рассмотрим пример

```
public class BaseClass{
public void method () throws IOException{
 ...
 }
}

public class LegalOne extends BaseClass{
 public void method () throws IOException{
  ...
 }
}

public class LegalTwo extends BaseClass{
 public void method () {
  ...
 }
}

public class LegalTree extends BaseClass{
 public void method ()
  throws EOFException,MalformedURLException  {
  ...
 }
}

public class IllegalOne extends BaseClass{
 public void method ()
  throws IOException,IllegalAccessException  {
  ...
```

```
 }
}

public class IllegalTwo extends BaseClass{
 public void method () {
  ...
  throw new Exception();
 }
}
```

в данном случае

определение класса LegalOne будет корректным, т.к. переопределение метода method() будет верным.

определение класса LegalTwo будет корректным, т.к. переопределение метода method() будет верным. (Переопределяемый метод не возбуждает исключений и поэтому не создает конфликта с переопределяемым методом)

определение класса LegalTree будет корректным, т.к. переопределение метода method() будет верным. (Метод может возбуждать исключения, которые являются подклассами исключения возбуждаемого в переопределяемом методе)

определение класса IlegalOne будет некорректным, т.к. переопределение метода method() неверно. (IllegalAccessException не является подклассом IOException)

определение класса IlegalTwo будет некорректным, method() переопределен верно, но он возбуждает исключение не указанное в throws.

13.7. Особые случаи

Во время исполнения кода могут возникать ситуации, которые редко или вообще не описаны в литературе.

Рассмотрим такую ситуацию.

```
import java.io.*;
public class Test  {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        try {
            test.doFileInput("bogus.file");
        }
        catch (IOException ex) {
            System.out.println("Second exception hadle starck trace");
            ex.printStackTrace();
        }
    }
```

```
    private String doFileInput(String fileName)
throws FileNotFoundException,IOException{
        String retStr = "";
        java.io.FileInputStream fis = null;
        try {
             fis = new java.io.FileInputStream(fileName);
        }
        catch (FileNotFoundException ex) {
            System.out.println("First exception hadle starck trace");
            ex.printStackTrace();
            throw ex;
        }
        return retStr;
    }
}
```

Результат работы будет выглядеть следующим образом.

```
java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:33)
 at experiment.Test.main(Test.java:21)
First exception hadle starck trace
java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:33)
 at experiment.Test.main(Test.java:21)
Second exception hadle starck trace
```

Так как при вторичном возбуждении используется один и тот же один объект Exception, то стек в обоих случаях будет содержать одну и ту же последовательность вызовов. Т.е. при повторном возбуждении исключения, если мы используем тот же объект, изменения его параметров не происходит.

Рассмотрим другой пример.

```
import java.io.*;

public class Test  {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        try {
            test.doFileInput();
```

```
        }
        catch (IOException ex) {
            System.out.println("Exception hash code " + ex.hashCode());
            ex.printStackTrace();
        }
    }

    private String doFileInput() throws FileNotFoundException,IOException{
        String retStr = "";
        java.io.FileInputStream fis = null;
        try {
             fis = new java.io.FileInputStream("bogus.file");
        }
        catch (FileNotFoundException ex) {
            System.out.println("Exception hash code " + ex.hashCode());
            ex.printStackTrace();
            fis = new java.io.FileInputStream("anotherBogus.file");
    throw ex;
        }
        return retStr;
    }
}

java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:33)
 at experiment.Test.main(Test.java:21)
Exception hash code 3214658

java.io.FileNotFoundException:  (The system cannot find the path specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:38)
 at experiment.Test.main(Test.java:21)
Exception hash code 6129586
```

Несложно заметить, что, несмотря на то, что последовательность вызовов одна и та же, в вызываемом методе и вызывающем обрабатываются разные объекты исключений.

Здесь следует обратить внимание, что если при обработке исключения произойдет в свою очередь новое исключение и оно не будет обработано в данном методе, то собственно информация об исключении, которое послужило первоисточником нештатной ситуации будет утеряно и информация о нем в вызывающий метод передана не будет. В случае, если в коде обрабатывающем исключение тоже может возникнуть внештатная ситуация, следует использовать вложенные блоки try{} catch(). Если преобразовать код к следующему виду, то программа будет вести себя ожидаемым образом.

```
import java.io.*;

public class Test  {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        try {
            test.doFileInput();
        }
        catch (IOException ex) {
            System.out.println("Exception hash code " + ex.hashCode());
            ex.printStackTrace();
        }
    }

    private String doFileInput() throws FileNotFoundException,IOException{
        String retStr = "";
        java.io.FileInputStream fis = null;
        try {
             fis = new java.io.FileInputStream("bogus.file");
        }
        catch (FileNotFoundException ex) {
            try {
                System.out.println("Exception hash code " + ex.hashCode());
                ex.printStackTrace();
                fis = new java.io.FileInputStream("");
            }
            catch (FileNotFoundException ex2) {
            }
            throw ex;
        }
        return retStr;
    }
}
```

```
java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:24)
 at experiment.Test.main(Test.java:12)

Exception hash code 3214658

java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
```

```
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:24)
 at experiment.Test.main(Test.java:12)

Exception hash code 3214658
```

14. Заключение

В данной главе рассмотрены основные языковые конструкции.

Циклы

- Три основных конструкции для выполнения циклов while(), do(), for().
- Выражение управляется выражением которое должно иметь булевский тип.
- В циклах for() while() условие выполнения цикла проверяется в начел цикла и до его выполнения. Таким образом тело цикла может быть не выполнено ни разу.
- В do циклах условие выполнения проверяется в конце цикла, таким образом цикл будет выполнен как минимум один раз.
- Для цикла for в скобках указывается три выражения. Первое выполняется до первой итерации цикла. Как правило здесь определяются и инициализируются переменные использующиеся в теле цикла. Второй параметр должен иметь булевское значение и определяет условие продолжения цикла. Третье выражение вычисляется сразу после выполнения тела цикла и до того как будет протестировано условие продолжения выполнения.
- Область видимости переменных объявленных в конструкции for ограничена телом цикла.
- Все три выражения цикла for не являются обязательными. Если условие продолжения цикла опущено, то оно трактуется как истинное.
- Предложение continue отменяет выполнение оставшейся части цикла для циклов while и do. В случае цикла for производится выполнение третьего выражения в конструкции и далее вычисляется условие продолжения цикла.
- Оператор break прекращает выполнение цикла. Для цикла for не производится тестирование условия продолжения цикла, выполнение третьего выражения не производится.
- Break и continue могут иметь в идее параметра метку, что позволяет прерывать выполнение вложенных циклов. Метка должна быть расположена перед объявлением цикла и заканчиваться двоеточием.

Операторы ветвления

- Оператор if() принимает на вход логическое (булевское) выражение
- else является необязательной частью оператора if()
- оператор switch принимает на входе целочисленное значение. Это может быть один из типов byte,short,char,int.
- Аргумент оператора case должен быть константой или выражением которое может быть вычислено во время компиляции
- В операторе case может быть только одна метка, в случае необходимости использовать несколько меток, следует применять несколько операторов case и не использовать brake для прекращения выполнения конкретного блока

- Если не выполнено ни одно из условий case, будет выполнен блок default (если он имеется)

Последовательность выполнения исключений

- В случае возникновения исключения управление передается за закрывающую скобку блока try, даже если исключение было вызвано из другого метода, находящегося в переделах блока try. В последнем случае выполнение вызываемого метода будет прекращено.
- Если встречается блок catch связанный с блоком try, и в качестве аргумента catch указан класс возникшего исключения или его родительский класс, будет выполнен первый встретившийся такой блок. Если это произошло, исключение считается обработанным, если этого не произошло, исключение считается не обработанным и передается для обработки вызвавшему коду.
- Блок finally выполняется вне зависимости от того возникло ли исключение, было - ли оно обработано или нет.
- Если исключения не возникло или исключение было обработано выполнение продолжается после блока finally
- Если исключение не обработано в текущем блоке, оно передается на обработку вышестоящему блоку try если оно не обрабатывается и там, то передается вверх по иерархии до тех пор, пока не достигнет самого верхнего уровня (главный поток приложения). В этом случае поток будет завершен и в System.err будет выведен дамп стека.

Возбуждение исключений

- Для возбуждения исключительной ситуации используется конструкция throw new XXXException()
- Любой подкласс java.lang.Exception будет обрабатываемым исключением, за исключением классов порожденных от java.lang.RuntimeException
- В методах, код которых может вызывать исключения, должен быть обрамлен блоками try{} catch(){} или в объявлении метода должно быть указано
- Метод не может возбуждать исключения если они не указаны после ключевого слова throws, за исключением RuntimeException и Error
- Если метод может возбуждать исключение (указано ключевое слово throws в объявлении метода), то вовсе не обязательно, что бы код возбуждающий это исключение был обязательно включен в этот метод
- Переопределенный (overridden) не может возбуждать исключение, если метод который он переопределяет не может возбуждать этого исключения или родительского класса исключения.

15. Контрольные вопросы

10-1. Приведенная ниже программа должна вывести на консоль Hello World! Выберите строки, которые нужно модифицировать в вашей программе, что бы получить правильный результат.

```
1. public class Test  {
2.     public Test() {
3.     }
4.     public static void main(String[] args) {
```

```
5.      Test test = new Test();
6.      String [] arr = {"H","e","l","l","o","
","w","o","r","l","d","!"};
7.      String result = "";
8.      int i= 0;
9.      for(;;;){
10.         result += arr[i++];
11.      }
12.      System.out.println(result);
13.    }
14. }
```

- a Заменить строку 9 на for(i = 0; i < arr.length;){
- b Заменить строку 9 на for(int int i = 0; i < arr.length;){
- c Заменить строку 9 на for(i = 0; i < arr.length;i++){
- d Заменить строку 9 на for(i = 1; i <= arr.length;i++){

a.) Правильный ответ a

Ответ b не верен так как переменная i уже определена в методе main Здесь следует еще раз напомнить, что область видимости переменной (если она объявлена в цикле for) будет ограничиваться лишь телом цикла. Однако в данном случае перемена с таким именем уже объявлена в теле метода и соответственно находится в той же области видимости. Если переменная была бы объявлена, как переменная класса, то этот код откомпилировался бы вполне успешно.

Ответ c не является верным так как, увеличение значения i в теле цикла будет произведено дважды, т.о. на печать будут выведены лишь четные элементы массива.

Ответ d не является верным по двум причинам. Первая – элементы массива нумеруются с 0, соответственно первым будет выбран второй элемент массива, вторая – когда выполнится условие окончания цикла, будет нарушена граница массива и будет вызвано исключение IndexOf-BoundException

10-2. Каков будет результат выполнения программы

```
1. public class Test  {
2.    public Test() {
3.    }
4.
5.    public static void main(String[] args) {
6.      Test test = new Test();
7.      int i = 5;
8.      while(i = 5){
9.         System.out.println(i++);
10.      }
11.    }
12. }
```

- a Компилятор выдаст сообщение об ошибке в строке 8
- b На консоль будут последовательно выведены значения 01234
- c На консоль будут последовательно выведены значения 43210
- d Программа откомпилируется,но на консоль ничего выведено не будет

a.) Правильный ответ a. В операторе while может быть использовано только булево значение. В данном случае используется оператор присваивания,а не сравнения, т.о. компилятор выдаст ошибку.

Так как здесь предложено выбрать только один ответ остальные ответы неверные

10-3. В данном случае выберите все правильные ответы.

```
private void say(int digit){
    switch(x){
        case 1: System.out.print("ONE");
              break;
        case 2: System.out.print("TWO");
        case 3: System.out.print("TREE");
        default:System.out.pritn("Unknown value")
    }
}
```

- a digit = 1 ONE
- b digit = 0 TWO TREE
- c digit = 2 TWO Unknown value
- d digit = 3 TREE Unknown value

a.) В данном случае правильными будут ответы a,d.

Рассмотрим последовательно вывод для всех возможных значений.

```
digit = 0 Unknown value.
```

В данном случае ни одно из значений case выполнено не будет и будет исполнен default.

```
digit = 1 ONE
```

В данном случае будет выполнено условие case 1: на консоль будет выведена соответствующая надпись. Далее следует оператор break и управление будет передано за фигурную скобку закрывающую блок switch

```
digit = 2 TWO TREE Unknown value
```

В данном случае будет выполнено условие case 2: на консоль будет выведено TWO. Т.к. далее не встречается оператор break, то выполнение switch будет продолжено, и на экран будут выведены TREE Unknown value

```
digit = 3  TREE Unknown value
```

В данном случае будет выполнено условие будет выполнено условие case 3: на консоль будет выведено TREE, как и в предыдущем случае оператор break здесь не используется, соответственно так же будет выполнен оператор default.

```
digit = 4 Unknown value
```

В данном случае ни одно из условий case выполнено не будет, поэтому отработает только оператор default.

10-4. Какая строка будет выдана на консоль после выполнения фрагмента кода приведенного ниже.

```
1. public class Test {
2.    public Test() {
3.    }
4.    public static void main(String[] args) {
5.       int i,j;
6.       lab: for(i = 0; i < 6; i++){
7.            for (j = 3; j > 1; j--){
8.                 if(i == j){
9.                   System.out.println(" " + j);
10.                   break lab;
11.                 }
12.            }
13.       }
14.    }
15. }
```

1. 2345
2. 234
3. 3
4. 2

a.) Правильный ответ d

Условие if в данном примере будет выполнено, когда переменные i и j будут равны 2. После чего на консоль будет выведено 2 и выполнится оператор break. Т.к. break содержит ссылку на метку, то будет прерван не текущий цикл (внутренний, по переменной j), а цикл по переменной i (внешний), т.о. образом выполнение программы будет прекращено.

10-5. Выберите все правильные варианты ответов в этом примере

```
1. public class Test {
2.   float fVal = 0.0f;
3.   public Test() {
4.   }
5.   public static void main(String[] args) {
6.      Test t = new Test();
7.      String testVal = "0.123";
8.      System.out.println("Was returned " + t.testParse(testVal) + "
with value " + t.fVal);
9.    }
10.   private boolean testParse(String val){
11.      try {
12.        fVal = Float.parseFloat(val);
13.        return true;
14.      }
15.      catch (NumberFormatException ex) {
16.        System.out.println("Test.testParse() Bad number -> " + val);
17.        fVal = Float.NaN;
18.      } finally{
19.        System.out.println("Finally part executed");
20.      }
21.      return false;
22.   }
23. }
```

1. testVal=“0.123”; Finally part executed Was returned true with value 0.123
2. testVal = “0,123”; Finally part executed Was returned false with value 0.123
3. testVal = null; Finally part executed Далее будет вызвано исключение NullPointer
4. testVal = “0.123”; Finally part executed Was returned false with value null

a.) Правильные ответы a,c

Вариант b не верен потому что, будет вызван оператор return в строке 13. Ответ d будет неверным т.к. при выполнении строки 12 будет возбуждено исключение, которое не обрабатывается ни в процедуре, ни в вызывающей программе, поэтому сообщение об ошибке будет выведено на консоль и выполнение программы прекратится.

10-6. Рассмотрим следующий пример.

Эти исключения имеют следующую иерархию наследования StringIndexOutOfBoundsException и ArrayIndexOutOfBoundsException

```
java.lang.Object
  |
  +--java.lang.Throwable
```

```
|
+--java.lang.Exception
      |
      +--java.lang.RuntimeException
            |
            +--java.lang.IndexOutOfBoundsException
                  |
                  +--java.lang.StringIndexOutOfBoundsException
                  |
                  +--java.lang.ArrayIndexOutOfBoundsException
```

Предположим, что в методе testSomeValue могут быть возбуждены оба вида этих исключений, при этом они не обрабатываются в блоке try – catch. Какое из ниже перечисленных суждений будет верным ?

- a Определение метода testSomeValue должно включать throws StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException
- b Если метод вызывающий testSomeValue перехватывает IndexOutOfBoundsException, то исключения StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException тоже будут перехыватываться.
- c Так как в определении метода указано throws StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException, то любой вызывающий его метод должен перехватывать эти типы исключений, вне зависимости возбуждается во время работы исключение или нет.
- d При объявлении метода testSomeValue не обязательно указывать возбуждаемые исключения

a.) Правильные ответы b,d

Ответ b будет правильным потому, что StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException являются потомками IndexOutOfBoundsException. И если перехватывается родительский класс исключений, то будут перехвачены и все его потомки.

Ответ d верен так, как StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException унаследованы от RuntimeException, то они являются необрабатываемыми исключениями и их указание в разделе throws определения метода не является обязательным

Ответ a неверен т.к. если метод возбуждает исключения унаследованные от RuntimeException, то указывать их в разделе throws нет необходимости, хотя это и не будет ошибкой.

Ответ c неверен по тем же причинам, что и вопрос 1. Даже если мы укажем в разделе throws StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException все равно нет необходимости указывать их в разделе catch.

10-7. Предположим необходимо создать собственную иерархию исключений.

Рассмотрим следующий пример.

```
Exception
  |
  +--LengthException
     |
     +--TooLongException
     |
     +--TooShortException

class BaseMeasurer{
    public BaseMeasurer(){
    }

    int measureLength(Dimension d) throws LengthException{
    }
}

class DerivedMeasurer extends BaseMeasurer{
    public BaseMeasurer(){
   }

  XXX {
  }
}
```

Какое из ниже перечисленных выражений можно использовать в строке 13 с тем, что бы код успешно откомпилировался

- a int measureLength(Dimension d) throws LengthException
- b int measureLength(Dimension d) throws Exception
- c int measureLength(Dimension d) throws TooLongException
- d int measureLength(Dimension d)

a.) Правильные ответы a,c,d

Ответ a верен, т.к. переопределенный метод может возбуждать тот же тип исключений, что и переопределяемый.

Ответ c верен, т.к. переопределенный метод возбуждает исключение которое является подклассом, исключения возбуждаемого в переопределяемом методе

Ответ d верен, т.к. переопределяемый метод не возбуждает исключений.

Ответ b неверен т.к. переопределенный метод расширяет список исключений возбуждаемых в переопределяемом методе.

10-8. Как и в предыдущем примере создадим собственную иерархию классов исключений.

```
Exception
  |
```

```
  +--LengthException
     |
     +--TooLongException
     |
     +--TooShortException
```

```
1. class TooShortException extends Exception{
2.     public TooShortException(String description){
3.         super(description);
4.     }
5. }
6.
7. class Measurer{
8.     public Measurer(){
9.         super();
10.     }
11.
12.     int measureLength(Dimension d) throws LengthException{
13.         XXX
14.     }
15. }
```

В строке 13 необходимо вызвать исключение. Какой из предложенных вариантов будет правильным ?

- a new TooShortException(“Shhhhort”);
- b throws new TooShortException(“Shhhhort”);
- c throw new TooShortException(“Shhhhort”);
- d throw TooShortException(“Shhhhort”);

a.) Правильным будет вариант c. В этом случае создается и возбуждается исключение.

Ответ a неверен т.к. создается экземпляр класса TooShortException, но возбуждения исключения не производится (т.е. не используется оператор throw)

Ответ b неверен т.к. вместо оператора throw используется throws

Ответ d неверен т.к. возбуждается исключение, но при этом экземпляр класса исключения не создается

10-9. Каков будет результат работы следующего кода

```
1. public class Test {
2.
3.    public Test() {
4.    }
5.    public static void main(String[] args) {
6.       Test t = new Test();
```

```
7.       XXX
8.    }
9.    private int check(String x,int n){
10.       if( n ==0 )return n;
11.       else if(n == 1){
12.          if ( x != null) return 5;
13.       }
14.       else if ( n == 2 && x != null){
15.          if(x.equals("YES")) return 3;
16.          else if ( x.equals("NO")) return 4;
17.       }
18.       return -1;
19.    }
20. }
```

Если в строке 7 поместить код вызова метода check, то какое из предложений будет верным ?

- a t.check(“ANY”,1) в этом случае обязательно будет выполнена строка 14.
- b t.check(“NO”,2) в этом случае функция вернет значение 4.
- c t.check(“YES”,1) в этом случае функция вернет значение 3
- d else в строке 14 относится к if в строке 11.

a.) Верными ответами будут b,d

Предложение a неверно, т.к. если n=1 и x = null то функция вернет значение 5 в строке 12.

Предложение c неверно потому, что если n = 1, то строки 15, 16 не выполняются, а значение 3 возвращается именно из этих строк.

10-10. Рассмотрим пример связанный с конструкцией switch…case

```
1. public class Test {
2.   public Test() {
3.   }
4.   public static void main(String[] args) {
5.     Test t = new Test();
6.     XXX
7.   }
8.
9.   private String check(int n){
10.     String retStr = "x";
11.     if (n < 3) n--;
12.     switch(n){
13.       case 1:
14.          return "one";
15.       case 2:
16.         n = 3;
17.       case 3:
```

```
18.         break;
19.       case 4:
20.       default:
21.         return retStr;
22.     }
23.     return "Result " + n;
24.   }
25. }
```

Если в строке 6 поместить вызов метода check, то выражения из ниже перечисленных можно считать верными ?

- a t.check(1) “one”
- b t.check(2) “Result 3”
- c t.check(3) “Result 3”
- d t.check(4) “X”
- e t.check(5) “X”

a.) Правильные ответы c,d,e. Рассмотрим логику работы метода при различных входных параметрах.

n=1 В строке 11 производится декремент n т.о. он примет значение 0 и из оператора switch case будет выбрано default

n=2 В строке 11 производится декремент n т.о. он примет значение 1 и будет выполнено условие case 1 и метод вернет значение “one”

n=3 Условие if в строке 11 выполнено не будет т.о. значение n останется без изменений и будет выполнено условие case 3, и нем прерывается выполнение switch и метод вернет Result 3

n=4 Условие if в строке 11 выполнено не будет т.о. значение n останется без изменений и будет выполнено условие case 4 и будет возвращено значение “X”

n=5 Условие if в строке 11 выполнено не будет т.о. значение n останется без изменений и будет выполнено условие default и будет возвращено значение “X”