Конспект лекции
Операторы и структура кода. Исключения

Цель и задачи лекции

Цель – рассмотреть управление ходом выполнения программы и возможные ошибки при выполнении программы.

Задачи:

1. Ознакомиться с ходом выполнения программы
2. Изучить операторы, влияющие на ход выполнения программы
3. Дать понятие локальных классов и переменных
4. Изучить инструкции управления программой
5. Понять иерархическую структуру классов ошибок

План занятия

1. Управление ходом выполнения программы
2. Блоки и инструкции
3. Исключения Java

Управление ходом программы

Управление потоком вычислений является фундаментальной основой всего языка программирования. Порядок выполнения программы определяется операторами. Операторы могут содержать другие операторы или выражения.

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).

Нормальное и преждевременное завершение инструкций

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.

Термины “завершена нормально” и “завершена преждевременно” применимы также к вычислению выражений. Единственная причина, по которой выражение может быть завершено преждевременно — генерация исключения, или из-за инструкции throw с заданным значением, или из-за исключения или ошибки времени выполнения.

Если инструкция вычисляет выражение, преждевременное завершение выражения всегда приводит к немедленному завершению инструкции по той же причине. Все последующие шаги нормального режима не выполняются.

Блоки и инструкции

Блок представляет собой последовательность инструкций, объявлений локальных классов и локальных переменных в фигурных скобках.

Выполнение блока осуществляется путем выполнения каждой инструкции объявления локальной переменной и других инструкций в порядке от первой до последней (слева направо). Если все эти инструкции блока завершаются нормально, то блок завершается нормально. Если любая из инструкций блока по любой причине завершается преждевременно, то весь блок завершается преждевременно по той же причине.

Локальные классы и переменные

Локальным классом является вложенный класс, не являющийся членом никакого класса и имеющий имя.

Все локальные классы являются внутренними классами.

Каждая инструкция объявления локального класса непосредственно содержит блок. Инструкции объявлений локальных классов могут свободно чередоваться с инструкциями другого вида в пределах блока.

Инструкция объявления **локальной переменной** объявляет одно или несколько имен локальных переменных.

Каждая инструкция объявления локальной переменной непосредственно содержится в блоке. Инструкции объявления локальных переменных могут свободно чередоваться с другими видами инструкций в блоке.

Логический оператор if

Инструкция if позволяет условное выполнение инструкции или условный выбор из двух инструкций, выполняя только одну из них.

В общем случае инструкция выглядит следующим образом:

```
if (логическое выражение)
    выражение или блок, если логическое выражение истинно
else
    выражение или блок, если логическое выражение ложно
```

Конструкция if-else-if

```
if ( условие )
    оператор 1;
else if ( условие )
    оператор 2;
else if ( условие )
    оператор 3;
...
else
    оператор N;
```

Условные операторы if выполняются последовательно, сверху вниз. Как только одно из условий, управляющих оператором if, оказывается равным true, выполняется оператор, связанный с данным условным оператором if, а остльная часть конструкции if-else-if пропускается. Если ни одно из условий не выполняется (т.е. не равно true), то выполняется заключительный оператор else. Этот последний оператор служит условием по умолчанию. Иными словами, если проверка всех остальных условий дает отрицательный результат, выполняется последний оператор else. Если же заключительный оператор else не указан, а результат проверки всех остальных условий равен false, то не выполняется никаких действий.

Оператор switch

В языке Java оператор switch является оператором ветвления. Он предоставляет простой способ направить поток исполнения команд по разным ветвям кода в зависимости от значения управляющего выражения. Зачастую оператор switch оказывается эффективнее длинных последовательностей операторов в конструкции if - else - if . Обща форма оператора switch имеет следующий вид:

```
switch (выражение) {
    case значение1:
```

```
        //последовательность операторов
        break;
    case значение2:
        //последовательность операторов
        break;
    ...

    default:
        //последовательность операторов
}
```

Во всех версиях Java до JDK 7 указанное выражение должно иметь тип byte, short, int, char или перечислимый тип. (Перечисления рассматриваются в главе 12.) Начиная JDK 7, выражение может также иметь тип String. Каждое значение, определенное в операторах ветвей case, должно быть однозначным константным выражением (например, литеральным значением). Дублирование значений в операторах ветвей case не допускается. Каждое значение должно быть совместимо по типу с указанным выражением.

Оператор switch действует следующим образом. Значение выражения сравнивается с каждым значением в операторах ветвей case. При обнаружении совпадения выполняется последовательность кода, следующая после оператора данной ветви case. Если значения ни одной из констант в операторах ветвей case не совпадают со значением выражения, то выполняется оператор в ветви default. Но указывать этот оператор не обязательно. В отсутствие совпадений со значениями констант в операторах ветвей case, а таке оператора default никаких дальнейших действий не выполняется.

Оператор break служит для прерывания последовательности операторов в ветвях оператора switch. Как только очередь доходит до оператора break, выполнение продолжается с первой же строки кода, следующей после всего оператора switch. Оператор break служит для немедленного выхода из оператора switch.

Операторы цикла

Для управления конструкциями, которые обычно называются циклами, в Java предоставляются операторы for, while и do-while. Циклы многократно выполняют один и тот же набор инструкций до тех пор, пока не будет удовлетворено условие завершения цикла.

Цикл while

Оператор цикла while является самым основополагающим для организации циклов в Java. Он повторяет оператор или блок операторов до тех пор, пока значение его управляющего выражения истинно. Этот оператор цикла имеет следующую общую форму:

```
while (условие) {
    //тело цикла
}
```

где условие обозначает любое логическое выражение. Тело цикла будет выполняться до тех пор, пока условно выражение истинно. Когда условие становится ложным, управление передает строке кода, непосредственно следующей за циклом. Фигурные

скобки мог быть опущены только в том случае, если в цикле повторяется лишь один оператор.

Цикл do-while

Как было показано выше, если в начальный момент условное выражение, управляющее циклом while, ложно, то тело цикла вообще не будет выполняться. Но иногда тело цикла желательно выполнить хотя бы один раз, даже если в начальный момент условное выражение ложно. Иначе говоря, возможны случаи, когда проверку условия прерывания цикла желательно выполнять в конце цикла, а не в начале. Для этой цели в Java предоставляется цикл, который называется do-while. Тело этого цикла всегда выполняется хотя бы один раз, поскольку его условное выражение проверяется в конце цикла. Общая форма цикла do-while следующая:

```
do {
    //тело цикла
} while ( условие );
```

При каждом повторении цикла do-while сначала выполняется тело цикла, а затем вычисляется условное выражение. Если это выражение истинно, цикл повторяется. В противном случае выполнение цикла прерывается. Как и во всех циклах в Java, заданное условие должно быть логическим выражением.

Цикл for

Начиная с версии JDK 5, в Java имеются две формы оператора цикла for. Первая форма считается традиционной и появилась еще в исходной версии Java, а вторая – более нова форма цикла в стиле for each. Традиционная форма имеет вид:

```
for(выражение инициализации; условие; итерация) {
    //тело цикла
}
```

Если в цикле повторяется выполнение только одного оператора, то фигурные скобки можно опустить. Цикл for действует следующим образом. Когда цикл начинается, выполняется его инициализация. В общем случае это выражение, устанавливающее значение переменной управления циклом, которая действует в качестве счетчика, управляющего циклом. Важно понимать, что первая часть цикла for, содержащая инициализирующее выражение, выполняется только один раз. Затем вычисляется заданное условие, которое должно быть логическим выражением. Как правило, в этом выражении значение управляющей переменной сравнивается с целевым значением. Если результат этого сравнения истинный, то выполняется тело цикла. А если он ложный, то цикл завершается. И наконец, выполняется третья часть цикла for – итерация. Обычно эта часть цикла содержит выражение, в котором увеличивается или уменьшается значение переменной управления циклом. Затем цикл повторяется, и на каждом его шаге сначала вычисляется условное выражение, затем выполняется тело цикла, а после этого вычисляется итерационное выражение. Этот процесс повторяется до тех пор, пока результат вычисления итерационного выражения не станет ложным.

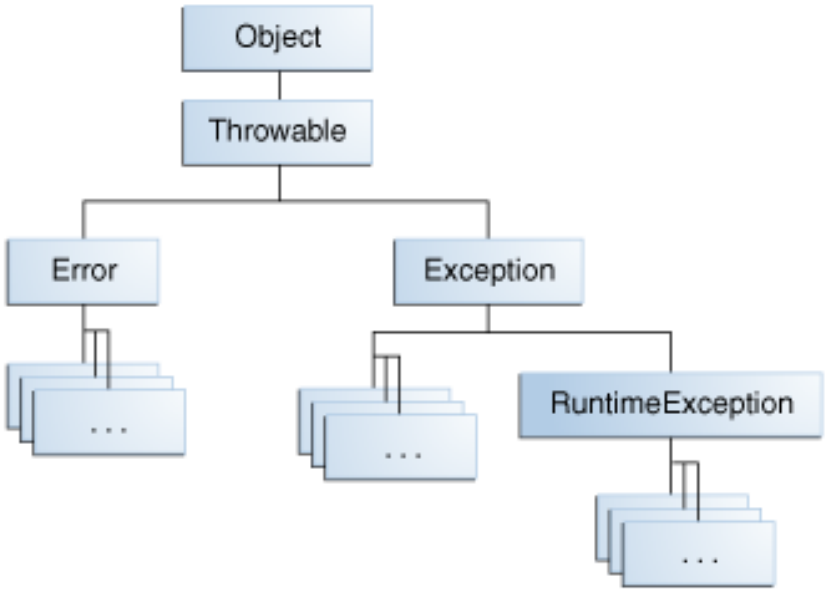
В Java можно использовать вторую форму цикла for, реализующую цикл в стиле foreach. Цикл в стиле foreach предназначен для строго последовательного выполнения повторяющихся действий над коллекцией объектов вроде массива. Общая форма разновидности цикла for в стиле foreach имеет следующий вид:

```
for( тип итерационная_переменная: коллекция ) {
    //тело цикла
}
```

где тип обозначает конкретный тип данных; итерационная_переменная - имя итерационной переменной, которая последовательно принимает значения из коллекции: от первого и до последнего: а коллекция - перебираемую в цикле коллекцию.

Исключения Java

Исключительные ситуации (исключения) и ошибки возникают во время выполнения программы, когда появившаяся проблема не может быть решена в текущем контексте и невозможно продолжение работы программы. Примерами «популярных» ошибок являются: попытка индексации вне границ массива, вызов метода на нулевой ссылке или деление на нуль. При возникновении исключения в приложении создается объект, описывающий это исключение. Затем текущий ход выполнения приложения останавливается, и включается механизм обработки исключений.



Каждой исключительной ситуации поставлен в соответствие некоторый класс, экземпляр которого инициируется при исключительной ситуации. Если подходящего класса не существует, то он может быть создан разработчиком.

Все исключения являются наследниками суперкласса Throwable и его подклассов Error и Exception из пакета java.lang.

Исключительные ситуации типа Error возникают только во время выполнения программы. Такие исключения связаны с серьезными ошибками, к примеру, с переполнением стека, не подлежат исправлению и не могут обрабатываться приложением.

Ошибки (Errors)

Error – критическая ошибка, приведшая к остановке программы. Не обрабатывается программистом НИКОГДА. Приводит к остановке программы ВСЕГДА. Пример - OutOfMemoryError (закончилась выделенная под программу оперативная память).

Проверяемые и непроверяемые исключения

Exception - общий класс исключений.

Исключения бывают «проверяемыми» и «непроверяемыми».

К «проверяемым» исключениям относятся исключения:

- Унаследованные непосредственно от класса Exception (кроме RuntimeException)
- Унаследованные далее по иерархии наследования от Exception (кроме RuntimeException)

Если используемый в коде метод выбрасывает «проверяемое исключение», то необходимо:

- Либо поместить его в блок try-catch-finally (или в блок «try with resources»)
- Либо поместить в блок throws сигнатуры этого метода класс данного исключения (или исключения, являющегося родительским для выбрасываемого)

Конструкция try-catch

В общем случае конструкция выглядит так:

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

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 оператора catch, ассоциированного с данным оператором try.

Если в пределах try возникает исключительная ситуация, то далее выполнение кода производится по одному из перечисленных ниже сценариев. Возникла исключительная ситуация, класс которой указан в качестве параметра одного из блоков catch. В этом случае производится выполнение блока кода, ассоциированного с данным catch (заключенного в фигурные скобки). Далее, если код в этом блоке завершается нормально, то и весь оператор try завершается нормально и управление передается на оператор (выражение), следующий за закрывающей фигурной скобкой последнего catch. Если код в catch завершается не штатно, то и весь try завершается нештатно по той же причине.

Если возникла исключительная ситуация, класс которой не указан в качестве аргумента ни в одном catch, то выполнение всего try завершается нештатно.

Конструкция try-catch-finally

Оператор finally предназначен для того, чтобы обеспечить гарантированное выполнение какого-либо фрагмента кода. Вне зависимости от того, возникла ли исключительная ситуация в блоке try, задан ли подходящий блок catch, не возникла ли ошибка в самом блоке catch, все равно блок finally будет в конце концов исполнен.

Последовательность выполнения конструкции следующая: если оператор try выполнен нормально, то будет выполнен блок finally. В свою очередь, если оператор finally выполняется нормально, то и весь оператор try выполняется нормально.

Если во время выполнения блока try возникает исключение и существует оператор catch, который перехватывает данный тип исключения, происходит выполнение связанного с catch блока. Если блок catch выполняется нормально, либо ненормально, все равно затем выполняется блок finally. Если блок finally завершается нормально, то оператор try завершается так же, как завершился блок catch.

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

Во всех случаях, если блок finally завершается ненормально, то весь try завершится ненормально по той же причине.

Работа с Closeable и AutoCloseable

Оператор try с ресурсами может объявлять один или более ресурсов. Ресурс как объект, который должен быть закрыт после того, как программа заканчивается с ним. Оператор try с ресурсами гарантирует, что каждый ресурс закрывается в конце оператора. Любой объект, который реализует java.lang.AutoCloseable и включает все объекты, которые реализуют java.io.Closeable, может использоваться в качестве ресурса.

С AutoClosable снято одно ограничение. Метод close() может бросать любое исключение, а не только IOException. С другой стороны, в отличие от Closable, вызывать close() на AutoClosable можно только единожды.

Следующий пример читает первую строку из файла. Это использует экземпляр BufferedReader считать данные из файла. BufferedReader ресурс, который должен быть закрыт после того, как программа заканчивается с ним:

```
static String readFirstLineFromFile(String path) throws IOException {
    try ( BufferedReader br = new BufferedReader(new FileReader(path)) ) {
        return br.readLine();
    }
}
```

Оператор throw

Исключения можно генерировать и непосредственно в прикладной программе, используя оператор throw.

Его общая форма выглядит следующим образом:

```
throw генерируемый_экземпляр;
```

где генерируемый_экземпляр должен быть объектом класса Throwable или производного от него подкласса. Примитивные типы вроде int или char , а также классы, кроме Throwable, например String или Object, нельзя использовать для генерирования исключений. Получить объект класса Throwable можно двумя способами , указав соответствующий параметр в операторе catch или создав этот объект с помощью оператора new.

Поток исполнения программы останавливается сразу же после оператора throw, и все последующие операторы не выполняются . В этом случае ближайший объемлющий блок оператора try проверяется на наличие оператора catch с совпадающим типом исключения. Если совпадение обнаружено, управление передается этому оператору. В противном случае проверяется следующий внешний блок оператора try и т.д. Если же не удастся найти оператор catch, совпадающий с типом исключения, то стандартный обработчик исключений прерывает выполнение программы и выводит результат трассировки стека.

Ниже приведен пример программы, в которой генерируется исключение. Обработчик, перехватывающий это исключение, повторно генерирует его для внешнего обработчика.

```
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        }
        catch (NullPointerException e) {
            System.out.println("caught inside demoproc");
            throw e;
        }
    }
    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("recaught: " + e);
        }
    }
}
```

В этом примере обработка исключения проводится в два приема. Метод main создает контекст для исключения и вызывает demoproc. Метод demoproc также устанавливает контекст для обработки исключения, создает новый объект класса NullPointerException и с помощью оператора throw возбуждает это исключение. Исключение перехватывается в следующей строке внутри метода demoproc, причем объект-исключение доступен коду обработчика через параметр e. Код обработчика выводит сообщение о том, что возбуждено исключение, а затем снова возбуждает его с помощью оператора throw, в результате чего оно передается обработчику исключений в методе main. Ниже приведен результат, полученный при запуске этого примера.

При переопределении методов следует помнить, что если переопределяемый метод объявляет список возможных исключений, то переопределяющий метод не может расширять этот список, но может его сужать. 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 LegalThree 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, так как переопределение метода method() верное (список ошибок не изменился);
- определение класса LegalTwo будет корректным, так как переопределение метода method() верное (новый метод не может выбрасывать ошибок, а значит, не расширяет список возможных ошибок старого метода);
- определение класса LegalThree будет корректным, так как переопределение метода method() будет верным (новый метод может создавать исключения, которые являются подклассами исключения, возбуждаемого в старом методе, то есть список сузился);

- определение класса IllegalOne будет некорректным, так как переопределение метода method() неверно (IllegalAccessExceptionне является подклассом IOException, список расширился);
- определение класса IllegalTwo будет некорректным: хотя заголовок method() объявлен верно (список не расширился), в теле метода бросается исключение, не указанное в throws.

Литература и ссылки

1. Спецификация языка Java https://docs.oracle.com/javase/specs/jls/se8/html/index.html
2. Механизм подавления исключений http://www.theserverside.com/tutorial/OCPJP-OCAJP-Java-7-Suppressed-Exceptions-Try-With-Resources
3. Шилдт Г. Java 8. Полное руководство. 9-е издание. — М.: Вильямс, 2012. — 1377 с.
4. Эккель Б. Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.

Вопросы для самоконтроля

1. Проанализируйте, скомпилируется ли приведенный ниже код и в какой строке произойдет ошибка

```
class A{
    public A method() throws Throwable { // 1
        return new Single();
    }
}

class Single extends A{
    public Single method(String str) throws RuntimeException { // 2
        return new Single();
    }
    public Single method() throws IOException {  //3
        return new Double();
    }
}

class Double extends Single{
    public void method(Integer digit) throws ClassCastException {      // 4
    }
    public Double method() throws Exception {  // 5
        return new Double();
    }
}
```

2. Запустите и проанализируйте результат выполнения программы

```
public class Main {
    public static void main(String[] args) {
        try{
            String str = null;
            if(str.equals("message")){
                System.out.println(str);
            }
        } catch (NullPointerException npe){
```

```
            System.out.println("NPE");
            return;
        } catch (ArithmeticException are){
            System.out.println("ARE");
        } catch (Exception ex){
            System.out.println("EX");
        } finally {
            System.out.println("Finally");
        }
    }
}
```