Конспект лекций по Java. Занятие 14

В.Фесюнов

1. Генерация исключительных ситуаций

На прошлом занятии мы рассмотрели основные вопросы, связанные с исключениями. Мы разобрали, что такое исключения, для чего они нужны, как ведет себя программа при возникновении исключения, как можно сгенерировать исключение. Познакомились также с аппаратом перехвата и обработки исключений, в частности с тем, какая информация может быть получена из объекта-исключения.

Нам осталось разобрать еще несколько вопросов по данной теме.

Вернемся к генерации исключений. Во-первых, когда может потребоваться генерация исключений в разрабатываемой программе. Примитивный ответ на этот вопрос - где-то "в глубине". Те программы, которые мы разбирали и которые можем разобрать в данном изложении, к сожалению, не могут продемонстрировать необходимости генерации исключения просто потому, что они маленькие.

Можно предложить к рассмотрению такую умозрительную ситуацию. Мы создаем библиотеку, которая делает что-то полезное. Это что-то может использоваться в любых программах, в том числе и в "недиалоговых". И в каком-то из классов библиотеки есть метод

```
public double f(double x) {
   . . .
}
```

Причем, по смыслу метода параметр x обязан быть неотрицательным числом.

Если бы такая ситуация встретилась в небольшой диалоговой программе, а не в универсальной библиотеке, приемлемый вариант был бы такой: выдать на экран

сообщение об ошибке и вернуть какое-то значение.

В нашей же умозрительной ситуации этого делать нельзя - нашей библиотеке ничего не известно о программе, которая ее использует. Здесь нужно сгенерировать исключение, подобрав подходящий к данной ситуации класс.

Обратимся к документации по пакету java.lang. В списке классов и интерфейсов (нижнее из двух левых окон) классы исключений сгруппированы отдельно. Сначала идут интерфейсы (**Interfaces**), потом обычные классы (**Classes**), потом исключения (**Exceptions**). В списке классов исключений мы можем увидеть тот класс, который по смыслу подходит к данной ситуации. Это **IllegalArgumentException** .

Выбрав подходящее к ситуации исключение, мы можем реализовать метод f(...) так:

```
public double f(double x) {
   if ( x < 0 )
       throw new IllegalArgumentException(
                 "Параметр x метода f(...) должен быть неотрицательным");
   . . .
}
```

В данном случае перехват этого исключения необязателен. Класс IllegalArgumentException порожден от RuntimeException, так что его можно не перехватывать.

- Если бы мы выбрали вместо `IllegalArgumentException` какое-то другое исключение, например, `IllegalAccessException` , то в описании метода f(..) мы должны были бы включить фрагмент "throws IllegalAccessException" перед фигурной скобкой. А при вызове f(...) всегда бы требовалось заключать этот вызов в try-catch-блок.

2. Создание собственных классов исключительных ситуаций

Разберемся, как создавать собственные классы исключений.

Во-первых, нужно отметить, что не нужно спешить создавать свои собственные

классы исключений. Сначала следует попытаться подобрать подходящее по смыслу исключение из стандартной библиотеки Java. И лишь в случае, когда это не удалось - создавать свое собственное.

Для создания своего собственного исключения достаточно написать такой код:

```
public class MyException extends Exception {
     public MyException() {};
     public MyException(String msg) {
           super(msg);
     };
}
```

Фактически единственное, что добавляет этот код к стандартным возможностям - это новое имя исключения (MyException). Но, обычно, этого достаточно.

Хотя никто не запрещает добавить в этот класс свои поля, конструкторы и методы.

- Следует отметить, что в данном случае создано исключение обязательное к перехвату. Если нужно создать исключение, которое не обязательно перехватывать, то нужно унаследовать его от RuntimeException.

3. Исключительные ситуации и наследование

Исключения влияют на аппарат наследования. Разберем это на примере.

```
public class A {
    . . .
    public int f(String str) throws SomeException {
     . . .
    }

    public int g(int n) {
     . . .
    }
    . . .
}
```

Пусть у нас есть вышеописанный класс A, и мы хотим использовать его в качестве базового для построения класса B. Пусть в классе B мы хотим переопределить (override) методы f(...) и g(...) класса A. В таком случае мы обязаны записать их заголовки точно так же, как они описаны в классе A. Т.е. мы обязаны указать "throws SomeException" в описании метода f(...) и не имеем право указывать предложение "throws ..." в описании метода g(...).

```
public class B extends A {
    . . .
    public int f(String str) throws SomeException {
     . . .
    }
    public int g(int n) {
     . . .
    }
    . . .
}
```

Те же правила относятся и к абстрактным классам и интерфейсам. В любом случае заголовки переопределяемых (override) методов должны быть одинаковыми.

4. Ввод/вывод (I/O) в Java

Стандартная библиотека Java имеет весьма развитые средства ввода/вывода. Однако, они оставляют впечатление какой-то незавершенности, несогласованности. Это, отчасти, так и есть. Дело в том, что в ранних версиях Java возможности ввода/вывода были реализованы с определенными ошибками, что заставило разработчиков Java создать новые возможности ввода/вывода, лишенные недостатков старых. Но, в то же время, для совместимости версий старые возможности тоже остались.

Это составляет определенную трудность в изучении возможностей ввода/вывода.

Хотя, если во всем четко разобраться и сделать практические выводы по использованию тех или иных стандартных приемов, то окажется, что при всем разнообразии возможностей, существует один-два варианта, из которых нужно сделать выбор.

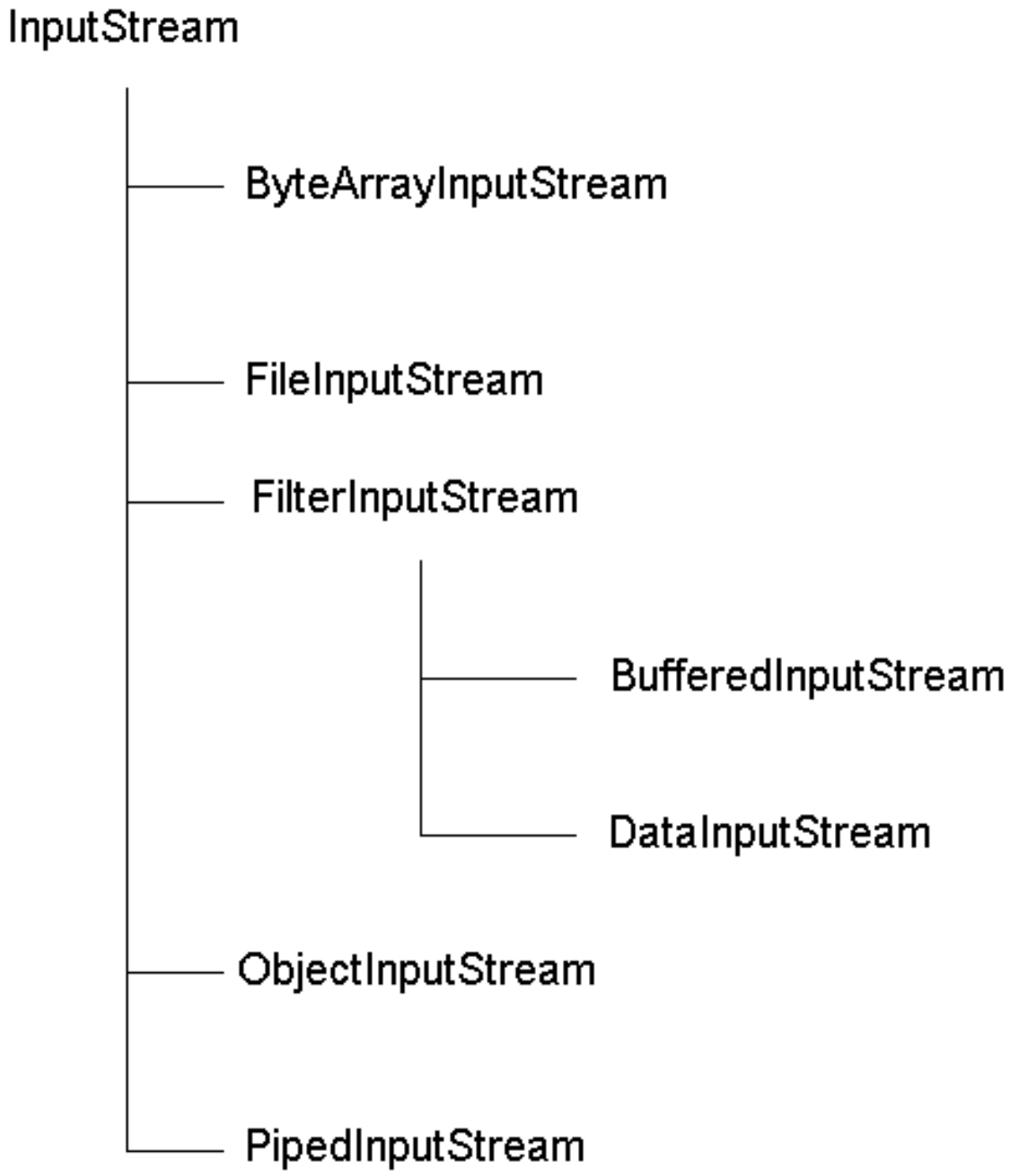
Все возможности ввода-вывода сосредоточены в пакете java.io.

Сначала рассмотрим всю иерархию классов ввода/вывода, а потом сделаем практические выводы.

Существуют две параллельные иерархии классов ввода (InputStream и Reader) и две параллельные иерархии классов вывода (OutputStream и Writer). Иерархии Reader и Writer введены в новых версиях Java. Кроме этих четырех иерархий есть еще класс RandomAccessFile, который объединяет как возможности ввода, так и возможности вывода, и класс File, предназначенный для работы с файловой системой на уровне файлов, каталогов и т.п.

Рассмотрим каждую из этих иерархий классов.

Иерархия InputStream



Здесь приведена только часть классов этой иерархии. Некоторые из этих классов мы рассмотрим бегло, а на некоторых остановимся более подробно.

Класс **InputStream** .

Является абстрактным классом. Но, несмотря на это, играет очень важную роль. Он является *родоначальником иерархии* и содержит реализацию ряда методов ввода.

Как родоначальник иерархии он используется следующим образом. Очень часто в качестве параметров конструкторов или методов различных классов выступает InputStream. Согласно правилам аппарата наследования это означает, что в качестве параметра может быть передан объект любого класса из приведенной иерархии. Как мы увидим ниже, это позволяет комбинировать классы для достижения нужных нам целей.

Кроме того, InputStream как родоначальник иерархии определяет, какие методы должны быть у всех классов данной иерархии.

Рассмотрим методы класса InputStream.

public abstract int read() throws IOException

Читает один байт из входного потока. Результат, как ни странно, int (занимает 4 байта). Прочитанный байт заносится в младший байт результата.

public int read(byte[] b) throws IOException

Читает последовательность байт из файла. Длина последовательности равна длине массива, заданного в качестве параметра. Может прочесть меньше байт, если чтение достигло конца файла. Возвращает количество прочитанных байтов.

public int read(byte[] b, int off, int len) throws IOException

То же, что и предыдущий метод, но заполняет массив с указанного байта и читает не больше, чем указано параметром len.

public long skip(long n) throws IOException

Пропускает n байт файла.

Это основные методы. Есть еще методы available, mark, markSupported, reset,

применяемые в специальных случаях.

Как мы видим, в InputStream реализованы все методы, кроме первого - он объявлен абстрактным. Соответственно, в порожденных классах все эти методы можно не реализовывать, а унаследовать от InputStream, кроме указанного абстрактного метода read(...), который должен быть реализован в любом порожденном неабстрактном классе.

- Все методы класса InputStream могут порождать IOException, поэтому вся работа с файлами должна быть заключена в try-catch-блоки.

Класс **FileInputStream** .

Это основной класс из данной иерархии для работы с файлами. Имеет два основных конструктора.

FileInputStream(File file) throws FileNotFoundException

и

FileInputStream(String name) throws FileNotFoundException

В остальном, этот класс ничего не добавляет к функциональности класса InputStream.

Класс **FilterInputStream** .

Это просто базовый класс для двух других классов. Он не является абстрактным, но сам по себе ничего не добавляет к возможностям InputStream. Его единственный конструктор требует передачи в качестве параметра объекта класса InputStream, т.е. фактически объекта некоторого неабстрактного класса, порожденного от InputStream.

Прямое использование этого класса в программе бессмысленно.

Класс **BufferedInputStream** .

Является одним из двух наследников FilterInputStream. Функционально он тоже ничего не добавляет к возможностям InputStream. Он служит для организации более эффективного "буферизованного" ввода данных.

Использовать его не обязательно, но для большей эффективности работы программы лучше все же использовать.

Класс **DataInputStream** .

Второй наследник FilterInputStream.

Имеет много полезных методов, позволяющих вводить из входного потока строки, числа любых типов. Но, тем не менее, в обычных программах он совершенно бесполезен, т.к. работает с кодировкой Unicode.

Класс **ObjectInputStream** .

Очень важный и нужный класс. Позволяет реализовывать ввод объектов. Мы будем рассматривать его отдельно, когда перейдем к интерфейсу Serializable.

Класс **PipedInputStream** .

Это специальный класс, используемый для связи отдельных программ друг с другом. Является важным инструментом организации синхронизации потоков. Мы не будем его рассматривать, но на него стоит обратить внимание.

4.1. Практическое использование классов иерархии InputStream

К сожалению, иерархия InputStream дает лишь базовые возможности ввода. Максимальный уровень, который она обеспечивает - это чтение массива байт. Это хорошо при построении на ее основе каких-то средств работы с файлами, но мало для практического использования в прикладных программах.

Можно было бы использовать функциональность DataInputStream, но, как отмечалось, он работает только с Unicode, что ограничивает его применимость.

Мы не будем рассматривать практическое использование иерархии InputStream, а обратимся сразу к иерархии Reader, которая дает более развитые средства работы с файлами.

4.2. Иерархия Reader

Является тоже довольно развитой иерархией.

Класс **Reader** .

Служит базовым классом в иерархии. Имеет точно такие же методы, как и класс InputStream.

Класс **FileReader** .

Имеет два основных конструктора, позволяющих открыть файл:

public FileReader(File file) throws FileNotFoundException

public FileReader(String name) throws FileNotFoundException

Больше никаких "своих" возможностей не имеет, все остальное он наследует от своих предков.

Класс **BufferedReader** .

Это самый интересный и важный для нас класс в этой иерархии. Он не только обеспечивает эффективный буферизованный ввод данных, но и имеет очень важный метод:

public String readLine() throws IOException

Этот метод позволяет читать строку из входного потока. В комбинации с другими классами Java этот метод позволяет организовать ввод с разбиением на слова, вводить числа и т.д.

Обратимся к документации по классу BufferedReader и обратим внимание на его конструкторы. Основной из них это:

public BufferedReader(Reader in)

Т.е. сам по себе этот класс использовать нельзя - только в комбинации с другим классом, например с FileReader.

Как мы увидим в примере далее, основной способ открытия файла для чтения выглядит так:

BufferedReader in = new BufferedReader(new FileReader("myfile.txt"));

Типичный пример чтения строки из файла:

```
String line;
. . .
    line = in.readLine();
```

Это наиболее важные классы иерархии Reader. Но для полноты картины рассмотрим и другие классы этой иерархии.

Класс **PushbackReader** .

Позволяет возвращать обратно в поток прочитанную информацию (или любую другую) так, что при следующем чтении будет читаться эта возвращенная в поток информация. Это не означает, что мы, скажем, модифицируем читаемый файл. Все модификации производятся во внутреннем буфере данного класса. Кстати, размер этого буфера ограничен и задается в конструкторе класса.

Класс **InputStreamReader** .

Это вспомогательный класс-переходник от иерархии InputStream к иерархии Reader.

Класс **PipedReader** .

По аналогии с PipedInputStream используется для связи программ друг с другом при помощи каналов (pipes).

Класс **StringReader** .

Обеспечивает работу по чтению информации из строки при помощи средств, предусмотренных иерархией Reader.

4.3. Развернутый пример использования иерархии Reader

Данный пример может служить в качестве отправной точки при построении программ, в которых необходимо организовывать разбиение на слова или осуществлять ввод числовой информации, представленной в текстовом виде.

В этом примере информация вводится построчно, каждая строка печатается. После этого выполняется разбор строки при помощи класса StringTokenizer. Строка разбивается на слова (слово - это подстрока, отделенная от другой подстроки пробелами). Каждое слово анализируется на наличие цифры в начале слова и в этом

случае преобразуется в число.

Файл SSTest.java

```
import java.util.*;
import java.io.*;

public class SSTest {

  public static void main(String args[]) {

    if( args.length == 0 ) {
      System.out.println("Нужен параметр вызова: имя файла");
      return;
    }
    String thisLine;

    try {
       BufferedReader fin = new BufferedReader(new FileReader(args[0]));

       while ((thisLine = fin.readLine()) != null) {
           System.out.println("==Введена строка:"+thisLine);
           StringTokenizer st = new StringTokenizer(thisLine);
           System.out.println("  Строка состоит из:");
           while (st.hasMoreTokens()) {
               String token = st.nextToken();
               char c = token.charAt(0);
               if( c < '0' || c > '9' )
                   System.out.println(token);
               else {
                   double d = Double.parseDouble(token);
                   System.out.println(token+" - это число = "+d);
               }
           }
       }
    } catch (FileNotFoundException e) {
        System.out.println("Файл " + args[0] + " не найден");
    } catch (Exception e) {
        e.printStackTrace();
    }
```

```
  }
}
```

Например, при обработке файла

```
1234567 98765 0987
9    23 45.4 56.7
asdfgt12
asd,орывпрпывр
```

мы получим такой результат

```
==Введена строка:1234567 98765 0987
  Строка состоит из:
1234567 - это число = 1234567.0
98765 - это число = 98765.0
0987 - это число = 987.0
==Введена строка:9    23 45.4 56.7
  Строка состоит из:
9 - это число = 9.0
23 - это число = 23.0
45.4 - это число = 45.4
56.7 - это число = 56.7
==Введена строка:asdfgt12
  Строка состоит из:
asdfgt12
==Введена строка:asd,орывпрпывр
  Строка состоит из:
asd,орывпрпывр
```

- Программа SSTest не претендует на завершенность и имеет свои недостатки. Так, в частности, появление слова "123abcd" во входном файле вызовет генерацию Exception.

Данный пример интересен тем, что в нем производится не только построчное чтение из файла, но и разбиение строк на слова. Кроме того, здесь продемонстрировано, как можно организовать ввод числовой информации: если слово является строковым представлением числа, то оно в данном примере преобразуется к типу double.

Познакомимся подробнее с классом StringTokenizer, который использован для разбиения строк на слова.

5. Класс StringTokenizer. (Пакет java.util)

Это полезный вспомогательный класс, позволяющий разбивать строку на слова с последующей обработкой слов в цикле. Разбиение строки производится конструктором класса.

public StringTokenizer(String str)

Кроме данного конструктора имеются еще два

public StringTokenizer(String str, String delim)

public StringTokenizer(String str, String delim, boolean returnDelims)

Первый позволяет определить список символов-разделителей слов. Так, например, мы могли бы улучшить наш пример, добавив символы точка, запятая, точка с запятой, двоеточие и др. в качестве разделителей при помощи конструктора:

StringTokenizer st = new StringTokenizer(thisLine, " \t\n\r\f.,;:");

Второй конструктор имеет параметр returnDelims. Если задать в качестве его значения true, то сами символы-разделители будут образовывать отдельные слова.

Класс StringTokenizer имеет ряд методов, но обычно используются следующие два метода:

public boolean hasMoreTokens()

Проверяет, есть ли еще слова в списке.

public String nextToken()

Выдает очередное слово.

6. Задания

1. Входной файл содержит целое число в первой строке, обозначающее размерность

матрицы и числа, задающие значения элементов матрицы в следующих строках. Написать программу для ввода такой матрицы. В качестве результата программа должна печатать на консоль матрицу по строкам.

2. Входной файл содержит только числа - значения элементов матрицы. Написать программу для ввода матрицы. Размерность матрицы определяется исходя из количества введенных элементов.

Подсказка: сначала нужно ввести элементы в список (например, LinkedList), потом проверить, что их количество равно квадрату некоторого n, потом разместить двумерный массив n на n и переписать в него элементы из списка.