МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Ордена трудового Красного Знамени федеральное государственное бюджетное
образовательное учреждение высшего образования
«Московский технический университет связи и информатики»

Кафедра Математическая кибернетика и информационные технологии

Отчет по лабораторной работе №4

“Обработка исключений”

Выполнил: студент группы БПИ2401

ФИО: Орлов Даниил Дмитриевич

Проверил: Харрасов Камиль Раисович

Москва, 2025

Цель работы:

Изучить механизм обработки исключений в языке Java и выполнить задания.

1 задание.

```
1   public class ArrayAverage {
        Run | Debug
2       public static void main(String[] args) {
```

Обьявляем публичный класс ArrayAverage.

```
3           String[] arr = {"1", "2", "three", "4", "5"};
4           double sum = 0;
5           int count = 0;
```

Обьявляем массив, в который будут храниться числа и элемент “three” для демонстрации NumberFormatException. Обьявляем переменную sum для подсчета суммы чисел и переменную count для подсчета чисел.

```
6           try {
7               for (int i = 0; i < arr.length; i++) {
8                   int value = Integer.parseInt(arr[i]);
9                   sum += value;
10                  count++;
11              }
```

Создаем блок try {, в котором будет располагаться код, который может вызвать исключения. Создаем цикл for, который проходит по всем числам в массиве с индексом от 0 до arr.lenght. с помощью Integer.parseInt() пытаемся преобразовать строку в int. если строка содержит нечисловой текст, то метод выбросит NumberFormatException. Далее добавляем число к сумме и увеличиваем счетчик.

```
12          if (count == 0) {
13              throw new ArithmeticException(s: "Нет числовых элементов для вычисления среднего");
14          }
```

Проверяем не равен ли счетчик чисел нулю. В противном случае это означает, что у нас нет ни одного числа и нужно вывести AretmeticException.

```
15          double average = sum / count;
16          System.out.println("Среднее арифметическое: " + average);
```

Создаем переменную average для обозначения среднего арифметического и выводим его на экран.

```
17          } catch (ArrayIndexOutOfBoundsException ex) {
18              System.err.println("Ошибка доступа к массиву: " + ex.getMessage());
```

Создаем первый catch, с помощью которого будем ловить ошибку выхода за границы массива (если она возникла в цикле for или при обращении arr[i]). Переменная ex хранит обьект исключения. Выводим сообщение об ишибке в поток ошибок System.err, с помощью ex.getMessage() возвращаем текст ошибки.

```
19          } catch (NumberFormatException ex) {
20              System.err.println("Неверный формат числа в массиве: " + ex.getMessage());
21          } catch (ArithmeticException ex) {
22              System.err.println("Невозможно вычислить среднее: " + ex.getMessage());
23          }
24      }
25  }
```

Создаем второй и третий catch, делаем для них вывод сообщения об ошибке.

2 задание.

```
1   import java.io.FileInputStream;
2   import java.io.FileOutputStream;
3   import java.io.IOException;
```

Подключаем классы.

```
5   public class FileCopy {
6       public static void copyWithManualClose(String sourcePath, String destPath) {
7           FileInputStream in = null;
8           FileOutputStream out = null;
```

Обьявляем публичный класс FileCopy. Далее создаем статический метод, принимающий пути к исходному и целевому файлам; демонстрирует ручное закрытие ресурсов. Обьявляем переменные для входного и выходного потоков.

```
 9          try {
10              in = new FileInputStream(sourcePath);
11              out = new FileOutputStream(destPath);
12              byte[] buffer = new byte[4096];
13              int bytesRead;
```

Создаем блок try. Создаем переменные для открытия файла, его чтения и записи. Создаем буфер для чтения блоками по 4096 байт. Далее создаем переменную для количества прочитанных байт в одном вызове read.

```
14              while ((bytesRead = in.read(buffer)) != -1) {
15                  out.write(buffer, off: 0, bytesRead);
16              }
```

Читаем в буфер до конца файла. in.read возвращает количество прочитанных байт или -1, когда файл заканчивается. Далее записываем фактическое число прочитанных байт.

```
17              System.out.println(x: "Копирование завершено (manual close).");
18          } catch (IOException ex) {
19              System.err.println("Ошибка при открытии/чтении/записи файла: " + ex.getMessage());
20          } finally {
```

Информируем об успешном завершении копирования. Ловим любое IOException и выводим сообщение об ошибке. Создаем блок finally, который выполняется в любом случае, он нужен для закрытия ресурсов.

```
21              if (in != null) {
22                  try {
23                      in.close();
24                  } catch (IOException ex) {
25                      System.err.println("Ошибка при закрытии входного потока: " + ex.getMessage());
26                  }
27              }
```

Проверяем был ли поток закрыт. Создаем блок try, с помощью которого будет производиться попытка закрыть поток. Ловим ошибку при закрытии и выводим ее.

```
28              if (out != null) {
29                  try {
30                      out.close();
31                  } catch (IOException ex) {
32                      System.err.println("Ошибка при закрытии выходного потока: " + ex.getMessage());
33                  }
34              }
35          }
36      }
```

Проверяем был ли выходной поток закрыт. Создаем блок try, с помощью которого будет производиться попытка закрыть выходной поток. Ловим ошибку при закрытии и выводим ее.

```
38      public static void copyWithTryWithResources(String sourcePath, String destPath) {
39          try (FileInputStream in = new FileInputStream(sourcePath);
40               FileOutputStream out = new FileOutputStream(destPath)) {
```

Создаем второй метод, который использует try-with-resources. Далее открываем входной и выходной потоки.

```
41              byte[] buffer = new byte[8192];
42              int bytesRead;
```

Создаем буфер и переменную, для количества прочитанных байт.

```
43              while ((bytesRead = in.read(buffer)) != -1) {
44                  out.write(buffer, off: 0, bytesRead);
45              }
```

Создаем цикл для чтения пока есть данные. Далее создаем запись, которая может вызывать IOException при ошибке записи.

```
45              }
46              System.out.println(x: "Копирование завершено (try-with-resources).");
47          } catch (IOException ex) {
```

Делаем вывод об успешном копировании. Далее создаем catch, который будет ловить любые IOException – это обработает ошибки открытия, чтения, записи и ошибки при закрытии.

```
48              System.err.println("Ошибка при копировании файла: " + ex.getMessage());
49          }
50      }
```

Выводим сообщение об ошибке при копировании файла.

```
52      public static void main(String[] args) {
53          String source = "source.txt";
54          String dest1 = "dest_manual.txt";
55          String dest2 = "dest_twr.txt";
```

Создаем метод main для тестирования. Далее указываем путь, имя файла для первого варианта и имя файла для второго.

```
56          copyWithManualClose(source, dest1);
57          copyWithTryWithResources(source, dest2);
58      }
59  }
```

Вызываем первый и второй метод.

3 задание

```
1   import java.io.FileWriter;
2   import java.io.IOException;
3   import java.io.PrintWriter;
4   import java.time.LocalDateTime;
5   import java.time.format.DateTimeFormatter;
```

Делаем все необходимые импорты

```
7   class CustomDivisionException extends Exception {
8       public CustomDivisionException(String message) {
9           super(message);
10      }
11  }
```

Обьявляем класс CustomDivisionException наследующий Exception. Благодаря extends Exception компилятор требует обработать или обьявить throws. Далее создаем конструктор класса, который принимает сообщение. super(message) вызывает конструктор суперкласса и передает сообщение – это стандартный способ задать текст исключение.

```
13  public class DivisionWithCustomException {
14      public static double divide(int a, int b) throws CustomDivisionException {
15          if (b == 0) {
16              throw new CustomDivisionException("Деление на ноль запрещено: делитель = 0");
17          }
18          return (double) a / b;
19      }
```

Создаем основной публичный класс программы. Создаем статический метод, который бросает CustomDivisionException при делении на ноль. Проверяем не равен ли делитель нулю. Если равен – бросаем наше пользовательское исключение с описанием. Если делитель не ноль – выполняем деление и возвращаем результат в дробной форме.

```
21      public static void logException(Exception ex) {
22          DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
23          String time = LocalDateTime.now().format(formatter);
```

Создаем статический метод logException, который принимает Exception и записывает ее в файл excrptions.log. Далее создаем форматтер для вывода даты и времени. Потом получаем текущую дату и время и форматируем в строку time.

```
24              try (FileWriter fw = new FileWriter("exceptions.log", true);
25                   PrintWriter pw = new PrintWriter(fw)) {
```

Открываем FileWriter в режиме append для добавления записей в конец файла. Используем try-with-resources для автоматического закрытия потока. Далее оборачиваем FileWriter в PrintWriter для удобства записи строк.

```
27              pw.println("[" + time + "] Exception: " + ex.getClass().getName());
28              pw.println("Message: " + ex.getMessage());
```

Записываем первую строку лога – временная метка и имя класса исключения. Далее записываем сообщение исключения.

```
28              for (StackTraceElement st : ex.getStackTrace()) {
29                  pw.println("\tat " + st.toString());
30              }
```

Проходим по стеку вызовов исключения, чтобы записать трассировку. Каждая строка стека вызывается с отступом \t.

```
31          } catch (IOException ioEx) {
32              System.err.println("Не удалось записать лог: " + ioEx.getMessage());
33          }
34      }
```

Если возникла ошибка при открытии или записи лога – обрабатываем ее. Далее информируем в стандартный поток ошибок.

```
36      public static void main(String[] args) {
37          int x = 10;
38          int y = 0;
```

Обьявляем метод main для теста. Внутри него обьявляем делимое и делитель.

```
40          try {
41              double result = divide(x, y);
42              System.out.println("Результат: " + result);
```

Создаем блок try для вызова divide, так как метод может бросить CustomDivisionException. осуществляем попытку деления. Если деление прошло успешно – выводим результат.

```
43          } catch (CustomDivisionException ex) {
44              System.err.println("Поймано CustomDivisionException: " + ex.getMessage());
45              logException(ex);
```

Ловим наше пользовательское исключение. Выводим сообщение об ошибке и вызываем наш метод logException, чтобы записать информацию об исключении в файл exceptions.log.

```
46          } catch (Exception ex) {
47              System.err.println("Поймано другое исключение: " + ex.getMessage());
48              logException(ex);
49          }
50      }
51  }
```

Создаем дополнительный catch для любых других исключений. Выводим сообщение об ошибке и вызываем наш метод для логирования.

Ответы на контрольные вопросы:

1. **Что такое исключение в Java?**

Ошибка, возникающая во время выполнения программы (runtime), которую можно обработать.

2. **Какие ключевые классы исключений вы знаете?**

- Exception
- RuntimeException
- IOException
- NullPointerException
- ArithmeticException
- IllegalArgumentException
- Error

3. **Что такое проверяемые и непроверяемые исключения?**

Checked - требуют try/catch или throws.

Unchecked - наследуются от RuntimeException, можно не ловить.

4. **Какие исключения и как необходимо обрабатывать?**

Только Checked (например, IOException).

Один из вариантов обработки – через try/catch.

5. **Какие исключения относятся к классу Error и как их обрабатывать?**

- StackOverflowError
- OutOfMemoryError
- LinkageError

Не обрабатываются - критические ошибки JVM.

6. **Какие исключения относятся к классу RuntimeException и как их обрабатывать?**

- NullPointerException
- ArithmeticException
- IndexOutOfBoundsException
- ClassCastException

Можно обрабатывать, но не обязательно.

7. **Как создать собственный класс исключения?**

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

8. **Как обрабатываютсяисключенияв Java? Какиеконструк ции для этого используются?**

Через:

- try
- catch
- finally
- throw
- throws

9. **Можно ли использовать try без catch или finally?**

Нет, кроме случая try-with-resources.

10. **Что произойдет, если исключение возникнет в блоке finally?**

Оно перекроет любое исключение из try/catch.

11. **Как можно пробросить исключение выше по стеку вы зовов?**

Использовать throws:

void method() throws IOException

12. В чем разница между finally и try-with-resources?

- finally - закрывать ресурсы вручную
- try-with-resources - закрываются автоматически

13. Какие классы можно использовать в try-with-resources? Что такое AutoCloseable?

Любой класс, реализующий AutoCloseable.

AutoCloseable — это интерфейс, который определяет один метод:

```
public interface AutoCloseable {
   void close() throws Exception;
}
```

14. Можно ли в одном try использовать несколько catch блоков? В каком порядке их нужно располагать?

Да, можно. Порядок - от более конкретных к более общим.

15. В чем разница между throw и throws?

throw - бросает исключение

throws - предупреждает, что метод может его бросить

16. Что такоеStackOverflowErroги OutOfMemoryError? Мож но ли их обработать?

StackOverflowError - переполнение стека (бесконечная рекурсия)

OutOfMemoryError - закончилась память

Обычно не обрабатываются, программа завершается.

Вывод:

В результате выполнения данной лабораторной работы были изучены и освоены основные механизмы обработки исключений.

Ссылка на репозиторий: https://github.com/qpdle/ITiP.git