GeekBrains

Исключения в программировании и их обработка

Урок 3

Продвинутые вопросы работы с исключениями в Java

Оглавление

На этом уроке

1. Разберём «продвинутые» возможности работы с исключениями.

Блок try-with-resources

1. При использовании внешних для JVM ресурсов, таких как файлы, сетевые соединения, соединения с базами данных и прочие, требуется **обязательно закрывать их в блоке finally**. Это связано с тем, что если приложение аварийно завершит свою работу, JVM сама почистит используемую память и освободит все свои служебные файлы. Но если ваше приложение получило доступ к сетевым соединениям, файлам или соединениям с базами данных, все эти ресурсы будут внешними для JVM, и она никак на них не сможет повлиять. Поэтому вы обязательно должны их освобождать, иначе всё заблокируете.

2. В примере ниже читаем содержимое файла для решения какой-то задачи, детали этой задачи не важны. В блоке finally освобождаем файл.

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class MainApp {
   public static void main(String args[]) {
       FileReader reader = null;
       try {
           reader = new FileReader(new
File("file.txt"));
           // Полезная работа, связанная с чтением
файла..
       } catch (IOException e) {
           e.printStackTrace();
       } finally {
           try {
               if (reader != null) {
                   reader.close();
               }
           } catch(IOException e) {
```

```
                e.printStackTrace();
            }
        }
    }
}
```

Выглядит код очень громоздко, блок finally занимает даже больше места, чем основная логика метода. Было бы неплохо упростить этот код. Для этого используем конструкцию **try-with-resources**. Код примет вид:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class MainApp {
   public static void main(String args[]) {
       try (FileReader reader = new FileReader(new
File("file.txt"))) {
           // Полезная работа, связанная с чтением
файла..
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

Рядом с try в круглых скобках указывается создаваемый ресурс, который при выходе из блока try должен быть освобождён, и не важно будет ли брошено какое-то исключение или нет. То есть вся работа по написанию блока finally выполняется автоматически без нашего участия.

Посмотрим, как Java определяет, что указанный объект в круглых скобках можно закрыть. Там можно указать только объекты, реализующие интерфейс **AutoClosable**. В таком случае для Java есть гарантия возможности вызова метода close() у объекта. Проверим это утверждение, создадим класс Box и скажем, что после работы коробка должна быть обязательно закрыта. Понятно, что это не очень важный ресурс ОС, но он используется для примера.

```
public class Box {

}
```

```
public class MainApp {
    public static void main(String[] args) {
        try (Box box = new Box()) {

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Получаем ошибку на этапе компиляции в строке, отмеченной маркером. Немного исправим код.

```
public class MainApp {
    public static void main(String[] args) {
        try (Box box = new Box()) {

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public class Box implements AutoCloseable {
    @Override
    public void close() throws Exception {
        // Закрываем коробку
    }
}
```

Теперь всё стало хорошо, ошибки на этапе компиляции нет, и при выходе из блока try у коробки будет вызван метод close(). При желании можете доработать этот пример и проверить, что close() действительно будет выполнен, добавив, например, вывод сообщения в консоль.

А если в блоке try-with-resources надо создать два объекта? Делать вложенную конструкцию?

Нет. Просто разделите создание объектов точкой с запятой.

```
public class MainApp {
    public static void main(String[] args) {
        try (Box box1 = new Box();
             Box box2 = new Box()) {
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

При работе с внешними ресурсами в коде старайтесь использовать try-with-resources. Это гарантирует, что вы не забудете закрыть ресурс в блоке finally, и значительно уменьшит количество кода.

Обработка исключений выше по стеку

На прошлом занятии мы уже говорили, что Checked-исключения обязательно обрабатываются либо посредством блока try-catch, либо через проброс на уровень выше. И если с try-catch всё понятно, то с throws — не совсем.

Когда надо использовать тот или иной подход? Почему бы всегда не брать первый?

Есть очень хороший простой и показательный пример. Представьте, что вы пишете приложение с графическим интерфейсом, которое формирует отчёты и складывает их в некое файловое хранилище. Код по сохранению отчёта по нажатию на кнопку Save выглядит так:

```
package com.geekbrains.app;

import com.xlsreport.maker.ReportExporter;

import javax.swing.*;

public class MainAppWindow extends JFrame {
    private ReportExporter reportExporter = new
ReportExporter();

    public void onSaveReportButtonClick() {
        String path = generateOutputReportPath();
        String outputData = "Очень важные данные для
отчета";
        reportExporter.saveReportToFile(path,
outputData);
    }

    public String generateOutputReportPath() {
        return "D:/reports/repository/1/" +
```

```
(int)(Math.random() * 10000000) + ".txt";
    }

    // ...
}
```

Сейчас придётся включить фантазию, чтобы не перегрузить код лишними деталями. Представим следующее:

1. Этот класс действительно целиком описывает некое окно графического приложения, написанного на библиотеке Swing, и в нём есть кнопка Save, при нажатии на которую вызывается метод onSaveReportButtonClick().
2. Берём действительно какие-то полезные данные и выбираем корректный путь к файлу, где эти полезные данные обязательно сохраняются, а не просто генерируем файл со случайным именем.
3. Для сохранения отчёта используем библиотеку, написанную другими разработчиками.

Если не сделать столько допущений, описание «приложения» займёт страниц 20–30, и вы просто потеряете мысль, за которой надо следить.

Теперь посмотрим на код «библиотеки» для формирования отчётов.

```
package com.xlsreport.maker;

import java.io.IOException;
import java.io.PrintWriter;

public class ReportExporter {
    public void saveReportToFile(String path, String
data) {
        String modifiedOutputData = data; // Представим,
что здесь форматируются входные данные
        try (PrintWriter writer = new PrintWriter(path))
{
            writer.println(modifiedOutputData);
        } catch (IOException e) {
            // Просто погасили исключение
        }
    }
}
```

Посмотрим, что происходит в коде выше: метод получает путь к файлу, куда надо сохранить отчёт, «как-то модифицирует данные, чтобы они красиво отображались в файле», и записывает результат по указанному пути. В блоке catch разработчик решил не мусорить сообщениями в консоль и просто ничего не сделал.

Вроде бы везде всё написано корректно.

Задание: посмотрите на код внимательно и подумайте, что может пойти не так.

А может пойти не так следующее: у разработчика есть диск D:/, он протестировал создание отчётов, всё хорошо работает, а значит, можно делать сборку приложения и передавать пользователям. Пользователи получили обновление, спокойно работают, жмут кнопку Save и даже не подозревают про подвох. Дело в том, что у некоторых пользователей в системе есть только диск C:/, и при попытке выполнить сохранение в библиотеке возникает исключение FileNotFoundException.

Но почему же они не узнали об ошибке и не обратились к разработчику, а впустую потратили время на множество несохранённых результатов?

Если взглянем на код библиотеки, то увидим, что исключение гасится, и даже в консоль никакое сообщение не выводится. Получается, что гасить исключения плохо, это может приводить к таким последствиям.

Поправим это.

```
package com.xlsreport.maker;

import java.io.IOException;
import java.io.PrintWriter;

public class ReportExporter {
   public void saveReportToFile(String path, String
data) {
       String modifiedOutputData = data; // Представим,
что здесь форматируются входные данные
       try (PrintWriter writer = new PrintWriter(path))
{
           writer.println(modifiedOutputData);
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
```

```
}
```

Стало ли лучше? В каком-то смысле, да, если у разработчика приложения возникнет исключение, он увидит его в консоли и попробует что-нибудь придумать. Но ведь может и не увидеть, помним, что у него диск D:/ существует. Пользователи точно не увидят, так как у них оконное приложение, а не консольное, да и даже если бы увидели распечатку исключений, вряд ли бы что-нибудь поняли. Получается, лучше стало, но требуется более корректное решение.

Может, попробуем в catch прописать показ окна с ошибкой?

Не можем, ведь мы не знаем, кто и в каком окружении будет использовать эту библиотеку: в консоли, окне, веб-приложении. Получается, как бы ни хотелось защитить пользователя нашей библиотеки от такой ошибки, мы не можем ничего сделать. Потому что просто невозможно на нашем уровне корректно обработать исключение. Есть ли решение? **Да**, воспользуется ключевым словом **throws**.

```
package com.xlsreport.maker;

import java.io.IOException;
import java.io.PrintWriter;

public void saveReportToFile(String path, String data)
throws IOException {
    String modifiedOutputData = data; // Представим, что
здесь форматируются входные данные
    try (PrintWriter writer = new PrintWriter(path)) {
        writer.println(modifiedOutputData);
    }
}
```

Мы указали, что нет никакой возможности корректно обработать в нашем методе исключение, поэтому мы его прокинем на уровень выше, туда, где его корректно обработает кто-нибудь другой. В таком случае ни в коем случае не пишем блок catch, чтобы не перехватить исключение.

Теперь надо вернуться к коду оконного приложения.

```
// ...
```

```
public class MainAppWindow extends JFrame {
    // ...

    public void onSaveReportButtonClick() {
        String path = generateOutputReportPath();
        String outputData = "Очень важные данные для
отчета";
        reportExporter.saveReportToFile(path,
outputData);
    }

    // ...
}
```

В выделенной маркером строке появится ошибка на этапе компиляции, указывающая, что есть необработанное checked-исключение. Это то, что нам надо! Мы как разработчики оконного приложения поймём, что файл может и не записаться на диск, а значит, было бы неплохо показать пользователю сообщение об ошибке. **Корректируем.**

```
// ...

public class MainAppWindow extends JFrame {
    // ...

    public void onSaveReportButtonClick() {
        String path = generateOutputReportPath();
        String outputData = "Очень важные данные для
отчета";
        ReportExporter reportExporter = new
ReportExporter();
        try {
            reportExporter.saveReportToFile(path,
outputData);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null, "Ошибка!
Невозможно сохранить отчет", "Ошибка!",
JOptionPane.ERROR_MESSAGE);
        }
    }

    // ...
}
```

JOptionPane.showMessageDialog() показывает окно с сообщением об ошибке. Теперь при успешном сохранении отчёта пользователи смогут работать

дальше, а в случаях отсутствия диска D:/ они получат окно с сообщением об ошибке и смогут сразу же обратиться к разработчику.

Описанная ситуация, конечно, **немного преувеличена**: разработчик аккуратнее будет писать код и не привязываться к конкретному диску, он не сможет там быстро передать пользователям новую версию приложения, не озаботившись тестированием, да и вряд ли разработчик будет писать десктопное приложение на Java, хотя почему бы и нет.

Но цель примера — показать, что бывают ситуации, когда как бы ни старались, вы не сможете корректно выполнить обработку исключения, при этом у вызывающего кода такая возможность есть, и исключения потребуется до него добросить. И, конечно же, вы должны увидеть, что без причины расставлять везде try-catch не получится!

Пока мы далеко не ушли от разбираемого примера. **Редкие, но всё же возможные случаи.**

А если нам понадобилось обработать исключение в двух местах: и в библиотеке, и в клиентском коде (опять немного фантазии)?

```
package com.xlsreport.maker;

import java.io.IOException;
import java.io.PrintWriter;

public void saveReportToFile(String path, String data)
throws IOException {
   String modifiedOutputData = data; // Представим, что
здесь форматируются входные данные
   try (PrintWriter writer = new PrintWriter(path)) {
       writer.println(modifiedOutputData);
   } catch (IOException e) {
       // Что-то полезное делаем
       throw e;
   }
}
```

Как видите, вы можете обработать исключение и повторно его бросить, чтобы клиентский код тоже мог его обработать. Вопрос: «Зачем библиотеке генератору отчётов в этом случае что-то обрабатывать», — оставим без ответа.

Создание собственных типов исключений

В языке Java помимо стандартных исключений, есть возможность использования собственных типов исключений. Чтобы создать новое исключение, надо всего лишь создать новый класс и унаследовать его от одного из существующих типов исключений.

```
public class MyException extends RuntimeException {
}
```

Готово. Теперь можно пробовать бросать исключение в коде.

```
public class MainApp {
    public static void main(String[] args) {
        throw new MyException();
    }
}
```

От чего зависит, будет ли созданное исключение Checked или Unchecked?

Если вы унаследовали класс от Checked-исключения, то и ваше исключение будет Checked, в противном случае — Unchecked.

Зачем создавать новые типы исключений, ведь в стандартной Java-библиотеке достаточно встроенных типов?

В проекте вам может потребоваться бросать исключения, связанные со спецификой решения какой-то задачи. Например, вы решили создать библиотеку для обработки изображений, и в этой библиотеке есть метод, позволяющий загрузить изображение в память. Но ваш код поддерживает не все возможные форматы.

Если пользователь попытается загрузить изображение неподдерживаемого формата, то как сообщить ему об этой ошибке?

Можно создать соответствующий тип исключений.

```
public class IllegalImageFormatException extends
RuntimeException {
}
```

Как уже было сказано ранее, даже название типа исключения помогает разработчику понять, что пошло не так. Допустим, в вашей библиотеке был

вызван метод, накладывающий на изображение фильтр, при этом в фильтр передались некорректные параметры. Подумаем, можно ли в таком случае бросать IllegalImageFormatException. Нет. Логично было бы создать новый тип исключений.

```
public class IllegalFilterParametersException extends
RuntimeException {
}
```

Продолжая эту идею, можно составить список исключений, которые будут описывать все возможные ошибки по работе с библиотекой, предназначенной для обработки изображений. Все исключения, характерные для вашей библиотеки или проекта, было бы логично сгруппировать, чтобы у них был общий корень. Посмотрим, что это значит.

```
public class JavaCVException extends RuntimeException {
}

public class IllegalImageFormatException extends
JavaCVException {
}

public class IllegalFilterParametersException extends
JavaCVException {
}
```

Создали корневое исключение JavaCVException (Java Computer Vision) и унаследовали остальные специфические исключения от него. Это даёт очень интересный эффект.

```
public class MainApp {
    public static void main(String[] args) {
        // ...
        try {
            cvLib.loadImage("C:/image.png");
        } catch (JavaCVException e) {
            e.printStackTrace();
        }
    }
}
```

Пока пользователь начинает работу с вашей библиотекой, он может везде перехватывать корневое исключение, не волнуясь, что приложение упадёт. Как только он получит достаточно опыта, то сможет начать менять код и указывать реакцию на уже конкретные исключения.

```
public class MainApp {
    public static void main(String[] args) {
        // ...
        try {
            CvImage img =
cvLib.loadImage("C:/image.png");
            cvLib.filters().blur(img, 2);
        } catch (IllegalImageFormatException e) {
            System.out.println("Выбрано изображение с
неподдерживаемым форматом, выберите другой файл");
        } catch (IllegalFilterParametersException e) {
            System.out.println("Выбраны некорректные
параметры фильтра");
        } catch (JavaCVException e) {
            System.out.println("Ой");
        }
    }
}
```

Ещё один интересный подход при работе с собственными типами исключений — «оборачивание» одного исключения в другое. Например, мы реализуем метод размытия изображения, и где-то в логике очень редко вылетает исключение, которое мы не учли.

Как сделать так, чтобы вместо него вылетело исключение *IllegalFilterParametersException*?

```
public class JavaCVLibFilters {
    public void blur(CvImage img, int kernelSize) {
        try {
            // Вычисления
            // Вычисления
            // Вычисления
        } catch (Exception e) {
            throw new IllegalFilterParametersException();
        }
    }
}
```

Если в процессе работы фильтра срабатывает ошибка, мы считаем, что пользователь некорректно настроил фильтр, но видит он вполне понятное исключение. А если бы из метода было выброшено ArrayIndexOutOfBoundsException или ArithmeticException, пользователю библиотеки было бы неясно, откуда взялись эти стандартные исключения. Получается, мы обернули стандартное исключение немного в другую форму.

Добавим немного особенностей нашим исключениям. Допустим, мы решили написать метод, преобразующий массив строк в массив целых чисел.

```
import java.util.Arrays;

public class MainApp {
    public static void main(String[] args) {
        System.out.println(Arrays.toString(transform(new
String[]{"1", "2", "3", "4"})));
    }

    public static int[] transform(String[] input) {
        int[] output = new int[input.length];
        for (int i = 0; i < input.length; i++) {
            output[i] = Integer.parseInt(input[i]);
        }
        return output;
    }
}
```

Если во входном массиве попадётся строка, которую невозможно преобразовать к целому числу, получим исключение NumberFormatException.

А если потребуется вместо этого бросить своё исключение, и чтобы исключение хранило информацию об индексе некорректного элемента?

```
public class ArrayTransformationException extends
RuntimeException {
}
```

С этим мы знакомы. А теперь немного модифицируем код. Позволим исключению хранить в себе информацию о некорректном элементе массива.

```
public class ArrayTransformationException extends
RuntimeException {
    private int illegalElementIndex;
    private String illegalElementValue;

    public int getIllegalElementIndex() {
        return illegalElementIndex;
    }

    public String getIllegalElementValue() {
        return illegalElementValue;
    }

    public ArrayTransformationException(int
illegalElementIndex, String illegalElementValue) {
        super(String.format("Во входном массиве на
позиции: %d находится некорректный элемент: %s",
illegalElementIndex, illegalElementValue));
        this.illegalElementIndex = illegalElementIndex;
        this.illegalElementValue = illegalElementValue;
    }
}
```

В первой строке конструктора заполняем сообщение, которое будет выведено в консоль, через конструктор родителя. Посмотрим, как этим теперь пользоваться.

```
public class MainApp {
    public static void main(String[] args) {
        System.out.println(Arrays.toString(transform(new
String[]{"1", "2", "3", "4"})));
    }

    public static int[] transform(String[] input) {
        int[] output = new int[input.length];
        for (int i = 0; i < input.length; i++) {
            try {
                output[i] = Integer.parseInt(input[i]);
            } catch (NumberFormatException e) {
                throw new ArrayTransformationException(i,
input[i]);
            }
        }
        return output;
    }
}
```

Если в процессе преобразования вылетит стандартный NumberFormatException, мы его перехватим и обернём в собственный тип исключения, передав в его конструктор детализацию об ошибке. Теперь специально укажем некорректный массив и посмотрим, что выведется в консоль.

```
public class MainApp {
   public static void main(String[] args) {
       System.out.println(Arrays.toString(transform(new
String[]{"1", "2a", "3", "4"})));
   }
}

Результат в консоли:
Exception in thread "main" ArrayTransformationException:
Во входном массиве на позиции: 1 находится некорректный
элемент: 2a
    at MainApp.transform(MainApp.java:14)
    at MainApp.main(MainApp.java:5)
```

Если потребуется, можете перехватить исключение, и через геттеры запросить у него либо индекс «битого» элемента, либо его значение.

Заключение

На этом курс по работе с исключениями завершён. Теперь вы понимаете причины возникновения ошибок в коде, знаете способы, которыми программа может о них сигнализировать, понимаете, почему важно корректно обрабатывать их, и знаете все инструменты, предоставляемые Java для удобной работы с исключениями.

Контрольные вопросы

1. В каких случаях надо использовать throws?
2. Зачем создавать собственные типы исключений?
3. Возможно ли обработать одно и то же исключение в двух разных местах кода?
4. В чём заключается преимущество try-with-resources?