Занятие 10

Anastasiya Solodkaya, Denis Stepulenok

LevelUP

11 ноября 2016 г.

1. Классы File, Path, Paths.
2. Потоки ввода/вывода
3. Scanner; java.nio.file: Files, Paths
4. Чтение и запись файла
5. Исключительные ситуации и их обработка
6. try-catch и try-with-resources

Работа с файлами

- java.io
- java.nio

Класс java.io.File

- Работает непосредственно с файлами
- Позволяет получить информацию о файле, о правах доступа к нему
- Папки - это тоже суть файлы, поэтому у него есть метод **isDirectory**, а так же методы для листинга внутренних файлов
- Может работать и с абсолютными, и с относительными путями

Класс java.io.File

```
File file0 = new File("test.txt");
File file1 = new File("C:\\test.txt");
File file2 = new File("C:\\Users");
```

Классы java.nio.file.Path и java.nio.file.Paths

- Работают с путями
- Позволяют получить информацию о пути, модифицировать его (подняться выше, объединить с другим и так далее)
- Класс **Path** имеет метод **toFile**
- Класс **Paths** вспомогательный - у него есть только методы **get** для того, чтобы получить путь по текстовому представлению

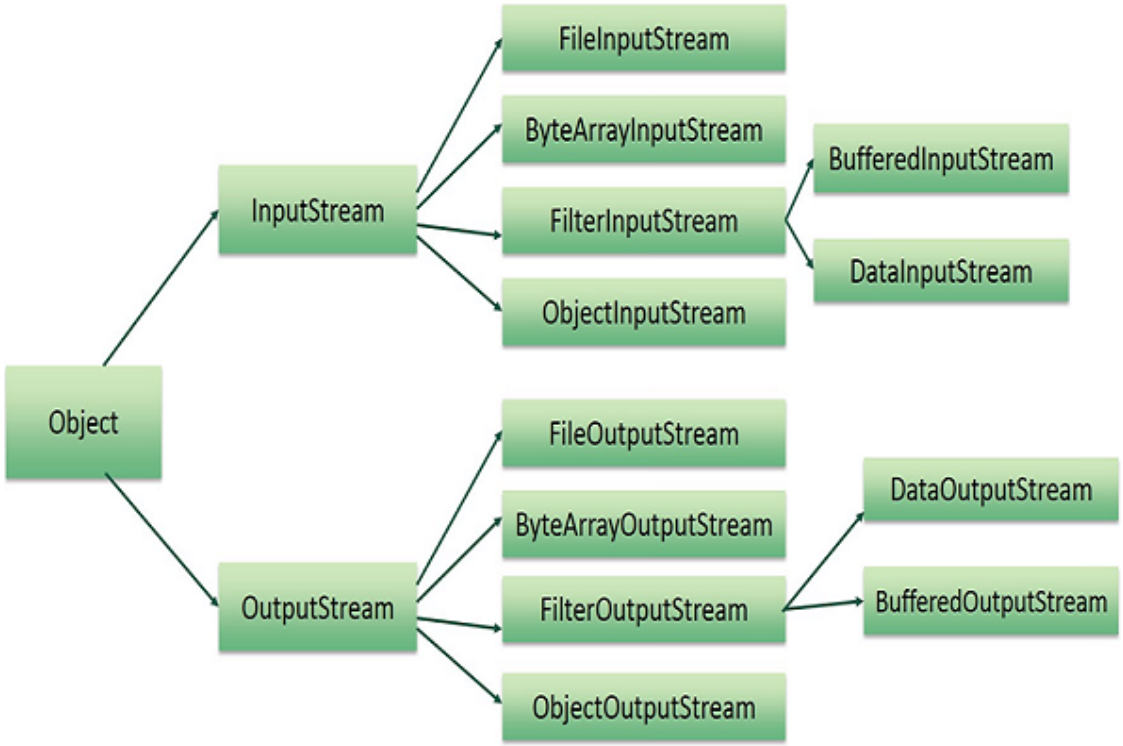
Класс java.nio.files.Paths

```
Path path0 = Paths.get("test.txt");
Path path1 = Paths.get("C:\\test.txt");
Path path2 = Paths.get(new URI("file:///C:/test.txt"));
```

Класс java.nio.file.Files

- Содержит множество методов, которые наиболее часто используются разработчиками
- Например, это **readAllLines** и **readAllBytes**
- А так же методы для проверки прав доступа, создания директорий (в том числе и временных)
- Открытие различных потоков для чтения-записи.

Потоки ввода и вывода в java



Классы для работы с файлами через потоки

- **Reader** - позволяет считывать последовательности символов
- **Writer** - позволяет записывать последовательности символов
- **InputStream** - позволяет считывать последовательности байтов
- **OutputStream** - позволяет записывать последовательности байтов
- Префикс **File** означает, что работа идет с файлами

Классы для работы с файлами

- **FileInputStream** - позволяет считывать последовательности байтов из файла
- **BufferedReader** - для считывания пояледовательности символов с поддержкой буфферизации. Связка **FileInputStream** + **BufferedStream** - **весьма эффективна**
- **InputStreamReader** - тоже читает текст из файла, но при этом не происходит буфферизации.
- **FileOutputStream** - позволяет записывать последовательности байтов в файл
- В пару к **BufferedReader** идет **BufferedWriter** - записывает символы с поддержкой буфферизации
- **PrintWriter** - позволяет записывать последовательности символов с поддержкой форматирования

Пример

```
public static void main(String[] args) throws
    URISyntaxException, IOException {
     File file = new File("test.txt");
     BufferedWriter writer = new BufferedWriter(new
         OutputStreamWriter(new FileOutputStream(file)));
     writer.write("ABC");
     writer.flush();
     writer.close();
}
```

Пример

```
public static void main(String[] args) throws
    URISyntaxException, IOException {
     File file = new File("test.txt");
     PrintWriter writer = new PrintWriter(new
         OutputStreamWriter(new FileOutputStream(file)));
     writer.printf("%d%d", 10, 10);
     writer.flush();
     writer.close();
}
```

Пример

```
public static void main(String[] args) throws
    URISyntaxException, IOException {
    File file = new File("test.txt");
    BufferedReader reader = new BufferedReader(new
        InputStreamReader(new FileInputStream(file)));
    System.out.println(reader.readLine());
    reader.close();
}
```

Больше примеров, как считать файл

```
public static void main(String[] args) throws IOException {
    List<String> lines =
        Files.readAllLines(Paths.get("text.txt"));
    System.out.println(lines);
}
```

Больше примеров, как считать файл - Files

```
public static void main(String[] args) throws IOException {
    List<String> lines =
        Files.readAllLines(Paths.get("text.txt"));
    System.out.println(lines);
}
```

Больше примеров, как считать файл - Scanner

```
public static void main(String[] args) throws IOException {
    Scanner scanner = new Scanner(new File("test.txt"));
    while(scanner.hasNext()){
        System.out.println(scanner.nextLine());
    }
}
```

Exceptions

- Checked Exception - нуждаются в прямых указаниях программиста, как их обрабатывать. Их нельзя игнорировать, т.к. программа просто не скомпилируется.
- Runtime Exception - возникает в момент исполнения, обычно имеет отношение к ошибкам в логике работы программы, неправильном использовании методов и т.д.
- Error - проблемы, которые возникают в процессе исполнения, и они уже вне зоны контроля. Например, это StackOverflowError.

Exceptions

- Checked exception - **Exception**
- Runtime exception - **RuntimeException**
- Error - **Error**

Откуда берутся исключительные ситуации?

- Встроенный Java API
- Чужой код - библиотеки, коды коллеги и т.д.
- Ваш собственный код

Откуда берутся исключительные ситуации?

```
private static int divide(int a, int b){
    if(b == 0){
        throw new ArithmeticException();
    }
    return a / b;
}
```

Откуда берутся исключительные ситуации?

Что будет, если мы запустим такой код?

```
public static void main(String[] args) {
    int result = divide(5, 0);
}

private static int divide(int a, int b){
    if(b == 0){
        throw new ArithmeticException();
    }
    return a / b;
}
```

Откуда берутся исключительные ситуации?

Ответ: ArithmeticException - это runtime exception, оно будет выброшено, и метод завершит работу Exception in thread "main"java.lang.ArithmeticException
at com.levelp.app.mine.Test.divide(Test.java:14)
at com.levelp.app.mine.Test.main(Test.java:9)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:
at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI
at java.lang.reflect.Method.invoke(Method.java:497)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)

Откуда берутся исключительные ситуации?

Что будет, если мы запустим такой код?

```
public static void main(String[] args) {
    try {
        int result = divide(5, 0);
    } catch (ArithmeticException e) {
        System.out.println(e.getMessage());
    }
}

private static int divide(int a, int b){
    if(b == 0){
        throw new ArithmeticException("Zero division!");
    }
    return a / b;
}
```

Откуда берутся исключительные ситуации?

Ответ: напечатется строка

Zero division!

Откуда берутся исключительные ситуации?

А как насчет такого кода с checked exception?

```
private static int divide(int a, int b) {
    if (b == 0) {
        throw new Exception();
    }
    return a / b;
}
```

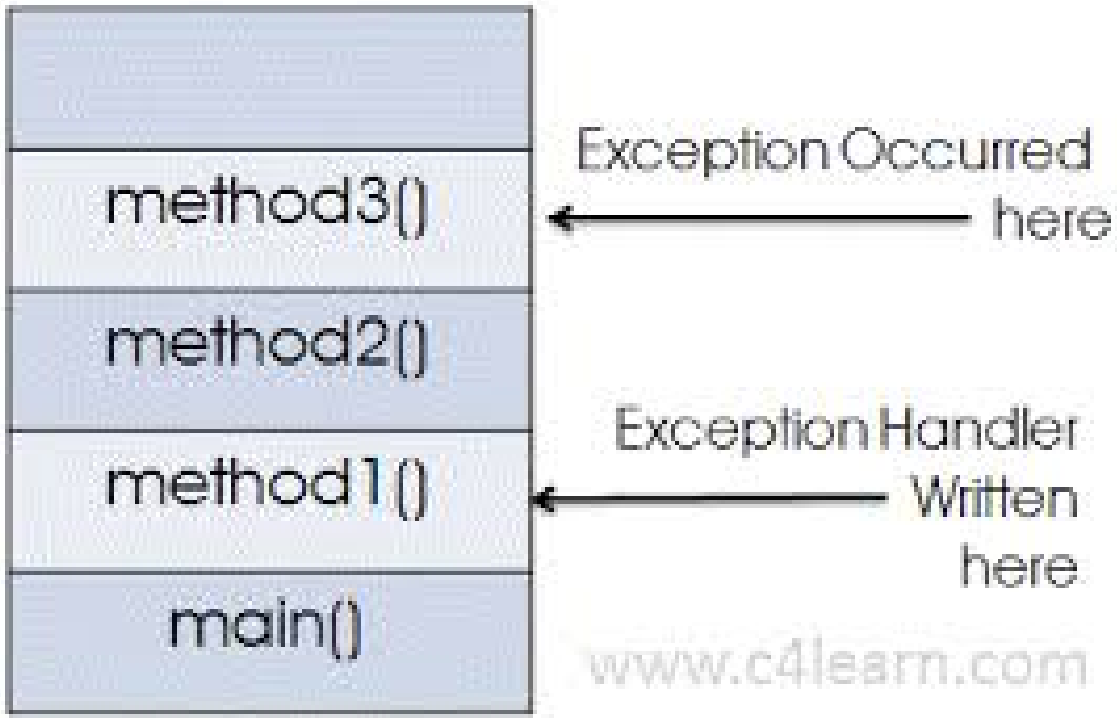
Такой код не скомпилируется. Компилятор вынуждает нас "сделать что-нибудь"с исключительной ситуацией.

Объявление исключительной ситуации в сигнатуре

```
private static int divide(int a, int b) throws Exception {
    if (b == 0) {
        throw new Exception();
    }
    return a / b;
}
```

Здесь мы передаем ответственность за обработку исключения методу, который вызывает наш метод. Возможно, он "передаст"ее так же наверх.

Что значит "передаст наверх"?



Ловля (обработка) исключительной ситуации

```
public static void main(String[] args) {
    try {
        int result = divide(5, 0);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

private static int divide(int a, int b) throws Exception {
    if (b == 0) {
        throw new Exception();
    }
    return a / b;
}
```

Резюме по обработке исключений

- Вы можете не обрабатывать исключительные ситуации типов **Error** или **RuntimeException**
- Вы обязаны сделать что-то с **checked exception**.
- Вы можете использовать конструкцию **try-catch** (+ finally) для обработки исключительной ситуации
- Вы можете объявить в описании метода какие угодно исключительные ситуации через слово **throws**, если их несколько - через запятую
- Вы можете выкинуть exception с помощью ключевого слова **throw**
- Точно так же вы можете выкинуть exception в блоке **catch** конструкции **try-catch** повторно с помощью все того же слова **throw**.

Как создать собственное исключение

```
public class MyException extends Exception {
}

public class MyException2 extends RuntimeException {
}

public class MyError extends Error {
}

public class MyThrowable extends Throwable {
}
```

try-catch

```
public static void main(String[] args) {
    try {
        int result = divide(5, 0);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

private static int divide(int a, int b) throws Exception {
    if (b == 0) {
        throw new Exception();
    }
    return a / b;
}
```

try-catch-finally

```
public static void main(String[] args) {
    try {
        int result = divide(5, 0);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    } finally {
        Sytstem.out.prntln("Complete!");
    }
}

private static int divide(int a, int b) throws Exception {
    if (b == 0) {
        throw new Exception();
    }
    return a / b;
}
```

try-catch-finally

- **catch** содержит декларацию того, что мы "ловим любой **Throwable**
- **catch** может содержать несколько типов в одном блоке
- **catch** блоки обрабатываются последовательно, а значит иерархия классов должна соблюдаться. Нет смысла ловить **FileNotFoundException** после **IOException**
- **finally** необходимо использовать для того, чтобы выполнить код по завершению **try-catch**
- **finally** выполнится в любом случае!
- Обычно это необходимо для того, чтобы закрыть ресурсы (например, потоки для работы с файлами)
- Самая плохая идея, которую можно придумать - возвращать что-то из **finally** блока.
- Вторая плохая идея - обрабатывать исключения пустым блоком

try-with-resources

```
public static void main(String[] args) {
    try (BufferedReader br =
                    new BufferedReader(new
                        FileReader("text.txt"))){
        System.out.println(br.readLine());
    }
}
```

try-with-resources

- Может содержать один или несколько ресурсов
- При выходе их блока закрывает все ресурсы
- Ресурсы должны быть **AutoCloseable**