

Занятие № 13, 14.

Исключения

Концепция обработки исключений

Сравнение традиционного механизма обработки ошибок с механизмом обработки исключений

Блок try-catch-finally

Типы исключений

Стандартные исключения в Java и их роль

Выброс исключения из метода

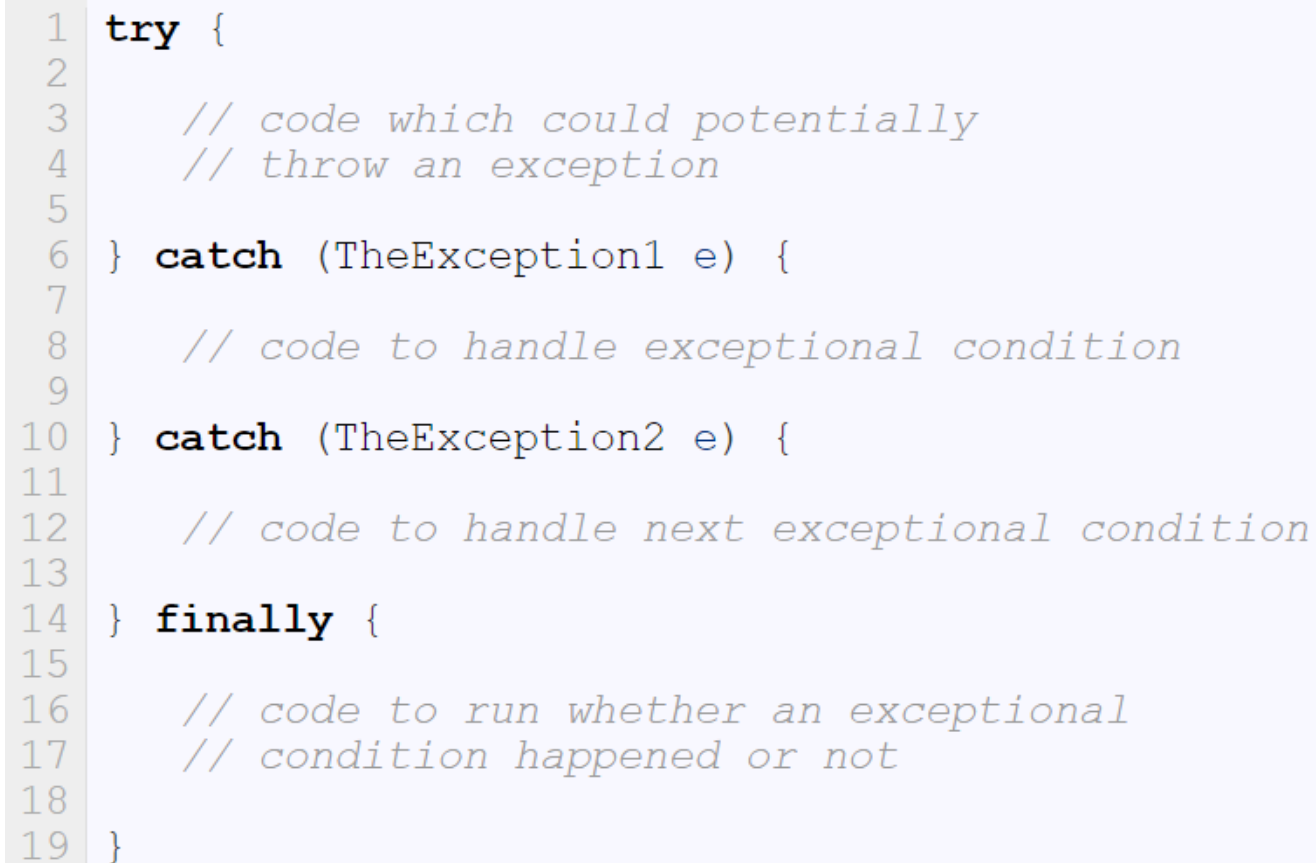
Концепция обработки исключений

Исключение - это нештатная ситуация, ошибка во время выполнения программы. Самый простой пример - деление на ноль. Можно вручную отслеживать возникновение подобных ошибок, а можно воспользоваться специальным механизмом исключений, который упрощает создание больших надёжных программ, уменьшает объём необходимого кода и повышает уверенность в том, что в приложении не будет необработанной ошибки.

В методе, в котором происходит ошибка, создаётся и передаётся специальный объект. Метод может либо обработать исключение самостоятельно, либо пропустить его. В любом случае исключение ловится и обрабатывается. Исключение может появиться благодаря самой системе, либо вы сами можете создать его вручную. Системные исключения возникают при неправильном использовании языка Java или запрещённых приёмов доступа к системе. Ваши собственные исключения обрабатывают специфические ошибки вашей программы.

Вернёмся к примеру с делением. Деление на ноль может предотвратить проверкой соответствующего условия. Но что делать, если знаменатель оказался нулём? Возможно, в контексте вашей задачи известно, как следует поступить в такой ситуации. Но если нулевой знаменатель возник неожиданно, деление в принципе невозможно, и тогда необходимо возбудить исключение, а не продолжать исполнение программы.

Обработка исключений – это встроенная возможность языка JAVA. Концепция обработки исключений позволяет сделать код более надежным и позволяет лучше читать и сопровождать его. Давайте посмотрим на конструкцию для обработки исключения:



```
1  try {  
2  
3      // code which could potentially  
4      // throw an exception  
5  
6  } catch (TheException1 e) {  
7  
8      // code to handle exceptional condition  
9  
10 } catch (TheException2 e) {  
11  
12     // code to handle next exceptional condition  
13  
14 } finally {  
15  
16     // code to run whether an exceptional  
17     // condition happened or not  
18  
19 }
```

В основном, код, который может вызвать исключение, помещается в такую конструкцию. Блок `catch`, описываемый для определенного исключения, будет вызываться – т.е. если случится исключение `TheException1`, тогда будет вызван блок внутри его `catch`. Однако блок `finally` будет вызываться всегда, даже если какой-либо блок `catch` содержит `return`.

Таким образом, блок `try` должен быть всегда. И всегда должен быть хотя бы один из блоков `catch` или `finally`.

Исходя из того, что блоков `catch` может быть много, система в случае исключения будет искать первый подходящий тип исключения. Причем с учетом наследования. Т.е. если вы ловите исключение `IOException`, то исключение `FileNotFoundException`, которое является подклассом `IOException`, будет обрабатываться в блоке `try`, который содержит `IOException`.

Первый совет по обработке исключений: старайтесь обрабатывать специфические исключения. Т.е. если вы создали метод, который будет генерировать исключение `FileNotFoundException` – не декларируйте, что метод вызывает исключение `IOException`. Нехорошо заставлять пользователя вашим классом обрабатывать исключение более высокого уровня, чем надо в действительности.

Второй совет: не делайте пустых catch блоков. Иными словами, не делайте такой код.

A screenshot of a code editor with a light blue background. The editor shows a Java try-catch block. Line 1: `try {`. Line 2: `...`. Line 3: `} catch (AnException e) {`. Line 4: `}`. The code is in a monospaced font. In the top right corner of the editor, there are icons for a menu, a code icon, a undo/redo icon, and a document icon.

```
1 try {  
2     ...  
3 } catch (AnException e) {  
4 }
```

Если даже логика вашего кода подразумевает полное отсутствие каких-либо действий при исключении – не поленитесь и напишите комментарий. Иначе пользователи вашего кода будут в затруднении.

Еще один добрый совет: Если метод класса вызывает исключение – пишите на него документацию. Используйте тэг `@throws`.

```
1  /**
2   * Loads the class
3   *
4   * @param name
5   *       Class name
6   *
7   * @return Resulting <tt>Class</tt> object
8   *
9   * @throws ClassNotFoundException
10  *        If class not found
11  */
12 public Class loadClass(String name) throws ClassNotFoundException
13 {
14     ...
15 }
```

RuntimeException исключения не требуют, чтобы их обрабатывали. И в общем не всегда их обработка будет выглядеть логичной и нужной.

Но в некоторых случаях это будет логично и понятно. Например, если вы ожидаете от пользователя ввода целого числа, которое вводится как строка. Для преобразования скорее всего будет использован метод `parseInt`, который вызывает runtime исключение `NumberFormatException`. Обычно runtime исключения обрабатываются тогда, когда необходимо восстановление при таких исключениях.

Существует пять ключевых слов, используемых в исключениях: **try**, **catch**, **throw**, **throws**, **finally**. Порядок обработки исключений следующий.

Операторы программы, которые вы хотите отслеживать, помещаются в блок **try**. Если исключение произошло, то оно создаётся и передаётся дальше. Ваш код может перехватить исключение при помощи блока **catch** и обработать его. Системные исключения автоматически передаются самой системой. Чтобы передать исключение вручную, используется **throw**. Любое исключение, созданное и передаваемое внутри метода, должно быть указано в его интерфейсе ключевым словом **throws**. Любой код, который следует выполнить обязательно после завершения блока **try**, помещается в блок **finally**.

Схематически код выглядит так:

```
try {  
    // блок кода, где отслеживаются ошибки  
}  
catch (тип_исключения_1 exceptionObject) {  
    // обрабатываем ошибку  
}  
catch (тип_исключения_2 exceptionObject) {  
    // обрабатываем ошибку  
}  
finally {  
    // код, который нужно выполнить после завершения блока try  
}
```

Существует специальный класс для исключений **Throwable**. В него входят два класса **Exception** и **Error**.

Класс `Exception` используется для обработки исключений вашей программой. Вы можете наследоваться от него для создания собственных типов исключений. Для распространённых ошибок уже существует класс `RuntimeException`, который может обрабатывать деление на ноль или определять ошибочную индексацию массива.

Класс `Error` служит для обработки ошибок в самом языке Java и на практике вам не придётся иметь с ним дело.

Прежде чем научиться обрабатывать исключения, нам хочется посмотреть, а что происходит, если ошибку не обработать. Давайте разделим число котов в вашей квартире на ноль, хотя мы и знаем, что котов на ноль делить нельзя!

```
int catNumber;  
int zero;  
catNumber = 1; // у меня один кот  
zero = 0; // ноль, он и в Африке ноль  
int result = catNumber / zero;
```

Я поместил код в обработчик щелчка кнопки. Когда система времени выполнения Java обнаруживает попытку деления на ноль, она создаёт объект исключения и передаёт его. Да вот беда, никто не перехватывает его, хотя это должны были сделать вы. Видя вашу бездеятельность, объект перехватывает стандартный системный обработчик Java, который отличается вредным характером. Он останавливает вашу программу и выводит сообщение об ошибке, которое можно увидеть в журнале LogCat:

```
Caused by: java.lang.ArithmeticException: divide by zero at ru.alexanderklimov.test.MainActivity.onClick(MainActivity.java:79)
```

Как видно, созданный объект исключения принадлежит к классу **ArithmeticException**, далее системный обработчик любезно вывел краткое описание ошибки и место возникновения.

Вряд ли пользователи вашей программы будут довольны, если вы так и оставите обработку ошибки системе. Если программа будет завершаться с такой ошибкой, то скорее всего вашу программу просто удалят. Посмотрим, как мы можем исправить ситуацию.

Поместим проблемный код в блок **try**, а в блоке **catch** обработаем исключение.

```
int catNumber;  
int zero;  
  
try { // мониторим код  
    catNumber = 1; // у меня один кот  
    zero = 0; // ноль, он и в Африке ноль  
    int result = catNumber / zero;  
    Toast.makeText(this, "Не увидите это сообщение!", Toast.LENGTH_LONG).show();  
} catch (ArithmeticException e) {  
    Toast.makeText(this, "Нельзя котов делить на ноль!", Toast.LENGTH_LONG).show();  
}  
Toast.makeText(this, "Жизнь продолжается", Toast.LENGTH_LONG).show();
```

Теперь программа аварийно не закрывается, так как мы обрабатываем ситуацию с делением на ноль.

В данном случае мы уже знали, к какому классу принадлежит получаемая ошибка, поэтому в блоке **catch** сразу указали конкретный тип. Обратите внимание, что последний оператор в блоке **try** не срабатывает, так как ошибка происходит раньше строчкой выше. Далее выполнение передаётся в блок **catch**, далее выполняются следующие операторы в обычном порядке.

Операторы **try** и **catch** работают совместно в паре. Хотя возможны ситуации, когда **catch** может обрабатывать несколько вложенных операторов **try**.

Если вы хотите увидеть описание ошибки, то параметр **e** и поможет увидеть его.

```
catch (ArithmeticException e) {  
    Toast.makeText(this, e + ": Нельзя котов делить на ноль!", Toast.LENGTH_LONG).  
    show();  
}
```

По умолчанию класс **Trowable**, к которому относится **ArithmeticException**, возвращает строку, содержащую описание исключения. Но вы можете и явно указать метод **e.toString**.

Блок **try-catch-finally**

Несколько исключений

Фрагмент кода может содержать несколько проблемных мест. Например, кроме деления на ноль, возможна ошибка индексации массива. В таком случае вам нужно создать два или более операторов **catch** для каждого типа исключения. Причём они проверяются по порядку. Если исключение будет обнаружено у первого блока обработки, то он будет выполнен, а остальные проверки пропускаются и выполнение программы продолжается с места, который следует за блоком **try/catch**.

```
int catNumber;
int zero;

try { // мониторим код
    catNumber = 1; // у меня один кот
    zero = 1; // ноль, он и в Африке ноль
    int result = catNumber / zero;
    // Создадим массив из трёх котов
    String[] catNames = {"Васька", "Барсик", "Мурзик"};
    catNames[3] = "Рыжик";
    Toast.makeText(this, "Не увидите это сообщение!", Toast.LENGTH_LONG).show();
} catch (ArithmeticException e) {
    Toast.makeText(this, e.toString() + ": Нельзя котов делить на ноль!", Toast.LENGTH_LONG).show();
}
catch (ArrayIndexOutOfBoundsException e) {
    Toast.makeText(this, "Ошибка: " + e.toString(), Toast.LENGTH_LONG).show();
}
Toast.makeText(this, "Жизнь продолжается", Toast.LENGTH_LONG).show();
```

В примере мы добавили массив с тремя элементами, но обращаемся к четвёртому элементу, так как забыли, что отсчёт у массива начинается с нуля. Если оставить значение переменной **zero** равным нулю, то сработает обработка первого исключения деления на ноль, и мы даже не узнаем о существовании второй ошибки. Но допустим, что в результате каких-то вычислений значение переменной стало равно единице. Тогда наше исключение **ArithmeticException** не сработает. Но сработает новое добавленное исключение **ArrayIndexOutOfBoundsException**. А дальше всё пойдёт как раньше.

Тут всегда нужно помнить одну особенность. При использовании множественных операторов **catch** обработчики подклассов исключений должны находиться выше, чем обработчики их суперклассов. Иначе, суперкласс будет перехватывать все исключения, имея большую область перехвата. Иными словами, **Exception** не должен находиться выше **ArithmeticException** и **ArrayIndexOutOfBoundsException**. К счастью, среда разработки сама замечает непорядок и предупреждает вас, что такой порядок не годится. Увидев такую ошибку, попробуйте перенести блок обработки исключений ниже.

Вложенные операторы try

Операторы try могут быть вложенными. Если вложенный оператор try не имеет своего обработчика catch для определения исключения, то идёт поиск обработчика catch у внешнего блока try и т.д. Если подходящий catch не будет найден, то исключение обработает сама система (что никуда не годится).

Оператор throw

Часть исключений может обрабатывать сама система. Но можно создать собственные исключения при помощи оператора throw. Код выглядит так:

```
throw экземпляр_Throwable
```

Вам нужно создать экземпляр класса **Throwable** или его наследников. Получить объект класса **Throwable** можно в операторе **catch** или стандартным способом через оператор **new**.

Поток выполнения останавливается непосредственно после оператора **throw** и другие операторы не выполняются. При этом ищется ближайший блок **try/catch** соответствующего исключению типа.

```
try {  
    throw new NullPointerException("Кота не существует");  
} catch (NullPointerException e) {  
    Toast.makeText(this, e.getMessage(), Toast.LENGTH_LONG).show();  
}
```

Мы создали новый объект класса **NullPointerException**. Многие классы исключений, кроме стандартного конструктора по умолчанию с пустыми скобками, имеют второй конструктор со строковым параметром, в котором можно разместить подходящую информацию об исключении. Получить текст из него можно через метод **getMessage()**, что мы и сделали в блоке **catch**.

Сравнение традиционного механизма обработки ошибок с механизмом обработки исключений

Традиционный подход к обработке динамических ошибок унаследован Delphi от Turbo Pascal и заключается в вызове функций, возвращающих коды завершения подпрограмм. Обычно подпрограмма, завершенная с нулевым кодом, считается успешно выполненной, а ненулевые коды говорят о каких-либо ошибках. Однако в Windows имеют место и функции, возвращающие нулевое значение в случае ошибки.

Возврат кода завершения подпрограммы характерен, например, для системных API-функций Windows. Определение ошибки при работе с такими функциями вызывает существенное затруднение по двум причинам. Во-первых, кодов ошибок для каждой функции множество, а потому немалое время уходит на изучение справочной информации, тем более что каждая функция возвращает свой набор ошибок. Во-вторых, при проверке возвращаемых значений к каждой строке исходного текста программы добавляется одна или несколько строк с проверками кода завершения, что отрицательно сказывается на читабельности исходных текстов программ.

Преимущество подхода использования исключений состоит в том, что программист не должен анализировать значение, возвращаемое подпрограммами, а может сосредоточиться на логике собственной программы, считая, что каждый фрагмент программы работает корректно, а в случае ошибки управление будет передано в соответствующее место автоматически.

Управление программой с помощью исключений является наиболее прогрессивным в современном программировании и рекомендуется для использования при написании собственных программ.

Типы исключений

Все исключительные ситуации можно разделить на две категории: проверяемые(`checked`) и непроверяемые(`unchecked`).

Все исключения, порождаемые от `Throwable`, можно разбить на три группы. Они определяются тремя базовыми типами: наследниками `Throwable`- классами `Error` и `Exception`, а также наследником `Exception` — `RuntimeException`.

Ошибки, порожденные от `Exception` (и не являющиеся наследниками `RuntimeException`), являются проверяемыми. Т.е. во время компиляции проверяется, предусмотрена ли обработка возможных исключительных ситуаций. Как правило, это ошибки, связанные с окружением программы (сетевым, файловым вводом-выводом и др.), которые могут возникнуть вне зависимости от того, корректно написан код или нет. Например, открытие сетевого соединения или файла может привести к возникновению ошибки и компилятор требует от программиста предусмотреть некие действия для обработки возможных проблем. Таким образом повышается надежность программы, ее устойчивость при возможных сбоях.

Исключения, порожденные от `RuntimeException`, являются непроверяемыми и компилятор не требует обязательной их обработки. Как правило, это ошибки программы, которые при правильном кодировании возникать не должны (например, `IndexOutOfBoundsException` - выход за границы массива, `java.lang.ArithmeticException` - деление на ноль). Поэтому, чтобы не загромождать программу, компилятор оставляет на усмотрение программиста обработку таких исключений с помощью блоков `try-catch`.

Исключения, порожденные от `Error`, также не являются проверяемыми. Они предназначены для того, чтобы уведомить приложение о возникновении фатальной ситуации, которую программным способом устранить практически невозможно (хотя формально обработчик допускается). Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые проблемы на уровне JVM. В качестве примера можно привести `StackOverflowError` (переполнение стека), `OutOfMemoryError` (нехватка памяти).

Методы, код которых может порождать проверяемые исключения, должны либо сами их обрабатывать, либо в заголовке метода должно быть указано ключевое слово `throws` с перечислением необрабатываемых проверяемых исключений. На непроверяемые ошибки это правило не распространяется. Переопределенный (`overridden`) метод не может расширять список возможных исключений исходного метода.

Стандартные исключения в Java

и их роль

Класс `Exception` обычно применяется в качестве универсального средства, позволяющего обрабатывать ошибки любого типа. Для более тонкой классификации ошибок лучше использовать стандартные классы, порожденные от класса `Exception`, или разработанные вами самостоятельно.

Стандартные классы обработки ошибок предусмотрены практически для каждой библиотеки классов Java. В описании методов указано, какие исключения могут создаваться при их вызове в случае возникновения тех или иных ошибочных ситуаций.

Для одного блока `try` можно определить несколько блоков `catch`, которые будут обрабатываться последовательно. Если возникнет исключение, то будет выполнен тот блок `catch`, в параметре которого это исключение объявлено. В том случае, когда ни один блок не подходит, выполняется блок с объявлением класса `Exception`. А если такой блок не предусмотрен, исключение будет обработано на уровне интерпретатора Java.

Описание примера

В этом примере мы разместили в одном блоке try три строки, вызывающих возникновение трех различных исключений:

```
int i = 0;
String szShortString = "123";
char chr;
Object ch = new Character('*');

try
{
    i = 5/i;
    chr = szShortString.charAt(10);
    System.out.println((Byte)ch);
}
```

В первой строке выполняется деление на ноль, в результате которого происходит исключение `ArithmeticException`.

Вторая строка вызывает возникновение исключения `StringIndexOutOfBoundsException`, так как в ней выполняется попытка адресации за пределы текстовой строки.

И, наконец, в третьей строке мы выполняем недопустимое преобразование классов, которое "наказывается" исключением `ClassCastException`.

Для обработки исключений в нашей программе предусмотрено три блока `catch`:

```
catch(StringIndexOutOfBoundsException ex)
{
    System.out.println(ex.toString());
}
catch(ArithmeticException ex)
{
    System.out.println(ex.toString());
}
catch(Exception ex)
{
    System.out.println(ex.toString());
}
```

Первые два из них перехватывают, соответственно, исключения `ArithmeticException` и `StringIndexOutOfBoundsException`, а третье - любые другие исключения, которые могут возникнуть. В частности, третий блок `catch` способен обработать исключение `ClassCastException`. Если запустить программу на выполнение, она закончит свою работу с сообщением о возникновении исключения `ArithmeticException`:

```
java.lang.ArithmeticException: / by zero
```

Закрыв символом комментария строку, выполняющую деление на нуль, мы включим в работу блок `catch` с параметром `StringIndexOutOfBoundsException`:

```
java.lang.StringIndexOutOfBoundsException:  
String index out of range: 10
```

Блок catch с объявлением класса Exception получит управление после того, как мы закроем внутри блока try все строки кроме последней, выполняющей неправильное преобразование классов. Этот блок выведет на консоль следующее сообщение:

```
java.lang.ClassCastException:  
    java.lang.Character
```

Если же в только что описанной ситуации мы дополнительно удалим блок catch с классом Exception, программа завершит свою работу с выдачей сообщения о возникновении исключения:

```
java.lang.ClassCastException:  
    java.lang.Character  
        at StdExeption.main(Compiled Code)
```

Из этого сообщения видно, что в методе `main` класса `StdExeption` возникло исключение `ClassCastException`. Оно возникло при работе с объектом класса `java.lang.Character`.

Выброс исключения из метода

Наследование методов, бросающих исключения

Рассмотрим следующий код:

```
public class SuperClass {  
    public void start() throws IOException{  
        throw new IOException("Not able to open file");  
    }  
}  
public class SubClass extends SuperClass{  
    public void start() throws Exception{  
        throw new Exception("Not able to start");  
    }  
}
```

Он не скомпилируется.

Дело в том, что метод `start()` был переопределен в сабклассе и в сигнатуре был указан более общий класс исключения. Согласно *правилам переопределения методов*, это действие не допустимо.

Можно лишь сужать класс исключения:

```
import java.io.FileNotFoundException;
import java.io.IOException;
public class SuperClass {
    public void start() throws IOException{
        throw new IOException("Not able to open file");
    }
}
class SubClass extends SuperClass{
    public void start() throws FileNotFoundException{
// FileNotFoundException - наследник IOException
        throw new FileNotFoundException("Not able to start");
    }
}
```

Рассмотрим пример:

```
public class ThrowsDemo{
    static void throwOne() throws IllegalAccessException {
        System.out.println(" В теле метода throwOne(al).");
        throw new IllegalAccessException("демонстрация");
    }

    public static void main(String args[]){
        try{
            throwOne();
        } catch(IllegalAccessException e) {
            System.out.println("Перехвачено исключение: " + e);
        }
    }
}
```

Здесь есть throws, который сообщает о том, что метод может выбросить это исключение и throw new IllegalAccessException("демонстрация"), который его выбрасывает.