



## Урок 2

# Исключения

Концепция обработки исключений, её сравнение с традиционным механизмом обработки ошибок, блок try-catch-finally, типы исключений, стандартные исключения в Java и их роль, выброс исключения из метода

### Исключения

[Блоки операторов try и catch](#)

[Вывод описания исключения](#)

[Применение нескольких операторов catch](#)

[Оператор throw](#)

[Оператор throws](#)

[Оператор finally](#)

[Встроенные в Java исключения](#)

[Создание собственных подклассов исключений](#)

[Многократный перехват исключений](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

# Исключения

Исключения в Java представляют собой объекты, генерируемые во время появления ошибочных ситуаций и содержащие информацию о них. Все исключения можно разделить на три группы:

- *Класс Exception и его подклассы*: исключения, которые обязательно должны быть перехвачены программой (Checked).
- *Класс RuntimeException и его подклассы*: исключения, охватывающие такие ситуации, как деление на ноль или ошибочная индексация массивов (Unchecked).
- *Класс Error и его подклассы*: исключения, появление которых не предполагается при нормальном выполнении программы, и используется для обозначения ошибок, происходящих в самой исполняющей среде. Примером такой ошибки может служить переполнение стека.

Иерархия исключений представлена на схеме ниже.

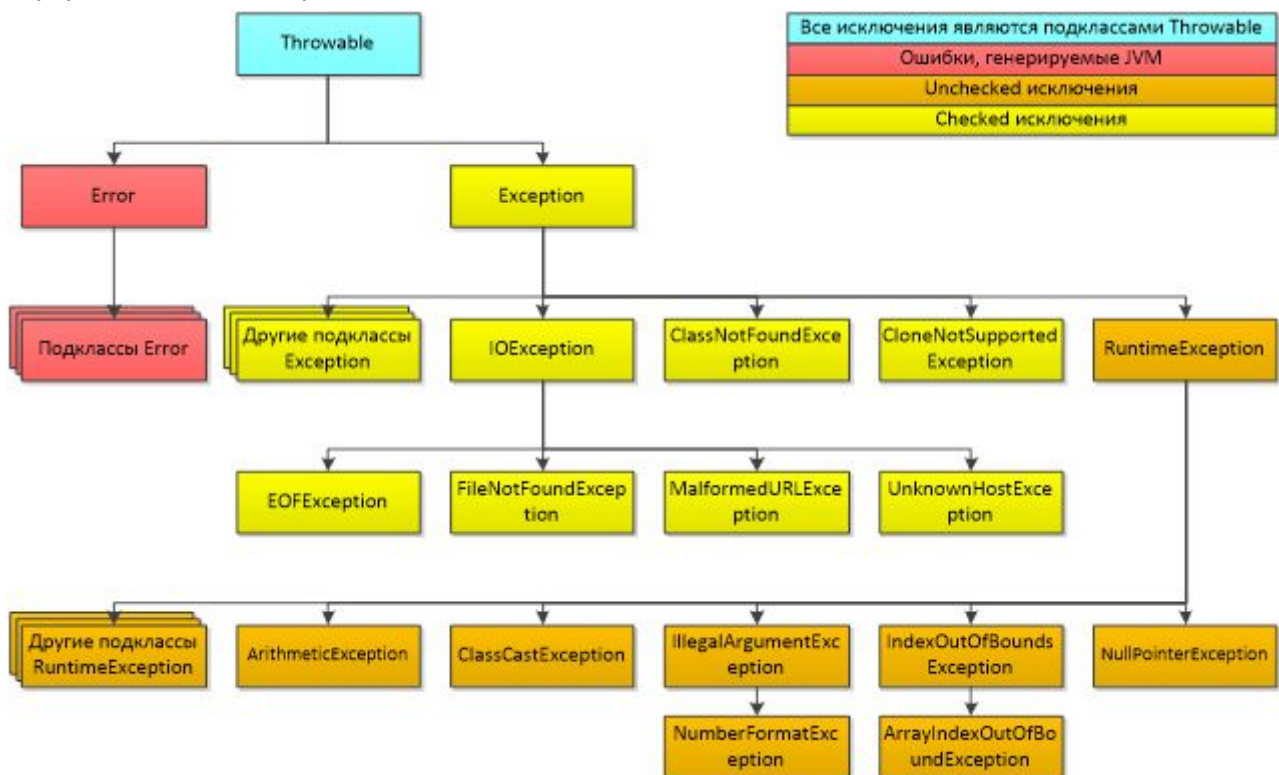


Рисунок 1 – Иерархия исключений Java

Рассмотрим пример кода, приводящего к ошибке при попытке деления на ноль.

```
public class MainClass {  
    public static void main(String[] args) {  
        int a = 0;  
        int b = 10 / a;  
    }  
}
```

При обнаружении попытки деления на ноль, исполняющая среда Java приостанавливает выполнение программы и генерирует исключение. Как только исключение сгенерировано, оно должно быть перехвачено обработчиком исключений, который в данном случае отсутствует, поэтому исключение перехватывается стандартным обработчиком, который выводит описание исключения и результат трассировки стека, а затем прерывает выполнение программы. Результат выполнения программы:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at MainClass.main(MainClass.java:4)
```

Стоит обратить внимание на то, что в трассировку стека включены имена класса MainClass, метода main(), файла MainClass.java и номер четвертой строки кода. Следует также иметь в виду, что сгенерированное исключение относится к подклассу ArithmeticException, описывающему тип возникшей ошибки. В Java представлено несколько встроенных типов исключений, соответствующих разным видам ошибок.

Трассировка стека позволяет проследить последовательность вызовов методов, которые привели к ошибке. Далее представлен пример, позволяющий чуть более подробно рассмотреть этот вопрос:

```
public class MainClass {
    public static void justMethod() {
        int a = 0;
        int b = 10 / a;
    }
    public static void main(String[] args) {
        justMethod();
    }
}
```

**Результат:**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at MainClass.justMethod(MainClass.java:4)
at MainClass.main(MainClass.java:8)
```

Как видите, на дне стека находится восьмая строка кода из метода main(), в которой производится вызов метода justMethod(), вызвавший исключение при выполнении четвертой строки кода.

## Блоки операторов try и catch

Стандартный обработчик исключений Java удобен для отладки, но, как правило, обрабатывать исключения приходится вручную, что позволяет исправить возникшую ошибку и предотвратить прерывание выполнения программы. Для этого достаточно разместить контролируемый код в блоке оператора try, за которым должен следовать блок оператора catch, с указанием типа перехватываемого исключения.

Рассмотрим пример программы, использующей блоки операторов try и catch для обработки исключения типа ArithmeticException, генерируемого при попытке деления на ноль:

```
public static void main(String[] args) {
    int a, b;
    try {
        a = 0;
        b = 10 / a;
        System.out.println("Это сообщение не будет выведено в консоль");
    } catch (ArithmeticException e) {
        System.out.println("Деление на ноль");
    }
    System.out.println("Завершение работы");
}
```

**Результат:**

```
Деление на ноль
Завершение работы
```

Вызов метода `println()` в блоке оператора `try` не выполнится, поскольку при возникновении исключения, управление сразу же передаётся из блока `try` в блок `catch`. По завершении блока `catch` управление передается в строку кода, следующую после всего блока операторов `try/catch`. Целью большинства правильно построенных операторов `catch` является разрешение исключительных ситуаций и продолжение нормальной работы программы, как если бы ошибки вообще не было.

## Вывод описания исключения

В классе `Throwable` определен метод `printStackTrace()`, который выводит полную информацию об исключении в консоль, что бывает полезным на этапе отладки программы. Например:

```
public static void main(String args[]) {
    System.out.println("Начало");
    try {
        int a = 0;
        int b = 42 / a;
    } catch (ArithmeticException e) {
        e.printStackTrace();
    }
    System.out.println("Конец");
}
Результат:
Начало
java.lang.ArithmeticException: / by zero
    at MainClass.main(MainClass.java:7)
Конец
```

В приведённом выше примере при делении на ноль была выведена полная информация об исключении, и программа продолжила свою работу.

## Применение нескольких операторов `catch`

Иногда в одном фрагменте кода может возникнуть несколько разных исключений. Чтобы справиться с такой ситуацией, можно указать два или больше оператора `catch`, каждый из которых предназначен для перехвата отдельного типа исключения. Когда генерируется исключение, каждый оператор `catch` проверяется по порядку и выполняется тот из них, который совпадает по типу с возникшим исключением. По завершении одного из операторов `catch` все остальные пропускаются и выполнение программы продолжается с оператора, следующего сразу после блока операторов `try/catch`. В следующем примере программы перехватываются два разных типа исключений:

```
public static void main(String args[]) {
    try {
        int a = 10;
        a -= 10;
        int b = 42 / a;
        int[] c = {1, 2, 3};
        c[42] = 99;
    } catch (ArithmeticException e) {
        System.out.println("Деление на ноль: " + e);
    } catch (ArrayIndexOutOfBoundsException e){
        System.out.println("Ошибка индексации массива: " + e);
    }
    System.out.println("После блока операторов try/catch");
}
```

Применяя несколько операторов catch, важно помнить, что перехват исключений из подклассов должен следовать до перехвата исключений из суперклассов. Дело в том, что оператор catch, в котором перехватывается исключение из суперкласса, будет перехватывать все исключения из этого суперкласса, а также все исключения из его подклассов. Это означает, что исключения из подкласса вообще не будут обработаны, если попытаться перехватить их после исключений из его суперкласса. Кроме того, недостижимый код считается в Java ошибкой. Рассмотрим в качестве примера следующую программу:

В последовательности операторов catch подкласс исключений должен быть указан перед его суперклассом, иначе это приведёт к недостижимому коду и ошибке во время компиляции.

```
public static void main(String args[]) {
    try {
        int a = 0;
        int b = 42 / a;
    } catch (Exception e) {
        System.out.println("Exception");
    } catch (ArithmeticException e) { // ОШИБКА : недостижимый код !
        System.out.println("Этот код недостижим");
    }
}
```

Если попытаться скомпилировать эту программу, то появится сообщение об ошибке, уведомляющее, что второй оператор catch недостижим, потому что исключение уже перехвачено. Класс исключения типа ArithmeticException является производным от класса Exception, и поэтому первый оператор catch обработает все ошибки, относящиеся к классу Exception, включая и класс ArithmeticException. Это означает, что второй оператор catch так и не будет выполнен. Чтобы исправить это положение, придётся изменить порядок следования операторов catch.

## Оператор throw

Пока что речь шла только об исключениях, генерируемых самой исполняющей системой Java. Но исключения можно генерировать и непосредственно в прикладной программе, с помощью оператора throw. Его общая форма выглядит следующим образом:

```
throw генерируемый_экземпляр;
```

где генерируемый\_экземпляр должен быть объектом класса Throwable или производного от него подкласса.

Поток исполнения программы останавливается сразу же после оператора throw, и все последующие операторы не выполняются. В этом случае ближайший объемлющий блок оператора try проверяется на наличие оператора catch с совпадающим типом исключения. Если совпадение обнаружено, управление передаётся этому оператору. В противном случае проверяется следующий внешний блок оператора try и т.д. Если же не удастся найти оператор catch, совпадающий с типом исключения, то стандартный обработчик исключений прерывает выполнение программы и выводит результат трассировки стека. Пример:

```
public static void main(String[] args) {
    try {
        throw new NullPointerException("NPE Test");
    } catch (NullPointerException e) {
        System.out.println("Catch block");
    }
}
```

## Оператор throws

Если метод способен вызвать исключение, которое он сам не обрабатывает, то он должен задать своё поведение таким образом, чтобы вызывающий его код мог обезопасить себя от такого исключения. С этой целью в объявление метода вводится оператор `throws`, где перечисляются типы исключений, которые метод может генерировать. Это обязательно для всех *checked* исключений (о них речь пойдет в пункте «Встроенные в Java исключения»). Если этого не сделать, то во время компиляции возникнет ошибка.

Ниже приведена общая форма объявления метода, которая включает оператор `throws`.

```
Тип название_метода(список_параметров) throws список_исключений {  
...  
}
```

Здесь `список_исключений` обозначает разделяемый запятыми список исключений, которые метод может сгенерировать. В примере ниже в методе `createReport()` может возникнуть исключение `IOException`, которое сам метод `createReport` не обрабатывает, следовательно, вызов этого метода необходимо взять в блок `try/catch`.

```
public static void main(String args[]) {  
    try {  
        createReport();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
public static void createReport() throws IOException {  
    PrintWriter pw = new PrintWriter("report.txt");  
    pw.close();  
}
```

## Оператор finally

Когда генерируется исключение, выполнение метода направляется по нелинейному пути, резко изменяющему нормальную последовательность выполнения операторов в теле метода. В зависимости от того, как написан метод, исключение может даже стать причиной преждевременного возврата из метода. В некоторых методах это может вызвать серьезные осложнения. Так, если файл открывается в начале метода и закрывается в конце, то вряд ли кого-нибудь устроит, что код, закрывающий файл, будет обойдён механизмом обработки исключений. Для таких непредвиденных обстоятельств и служит оператор `finally`.

Оператор `finally` образует блок кода, который будет выполнен по завершении блока операторов `try/catch`, но перед следующим за ним кодом. Блок оператора `finally` выполняется независимо от того, сгенерировано ли исключение или нет. Если исключение сгенерировано, блок оператора `finally` выполняется, даже при условии, что ни один из операторов `catch` не совпадает с этим исключением.

В любой момент, когда метод собирается вернуть управление вызывающему коду из блока оператора `try/catch` (через необработанное исключение или явным образом через оператор `return`), блок оператора `finally` выполняется перед возвратом управления из метода. Это может быть удобно для закрытия файловых дескрипторов либо освобождения других ресурсов, которые были выделены в начале метода и должны быть освобождены перед возвратом из него. Указывать оператор `finally` не обязательно, но каждому оператору `try` требуется хотя бы один оператор `catch` или `finally`. Ниже приведена общая форма блока обработки исключений.

```
try {
    // блок кода, в котором отслеживаются исключения
} catch (ТипИсключения1 e1) {
    // обработчик исключения тип_исключения_1
} catch (ТипИсключения2 e2) {
    // обработчик исключения тип_исключения_2
} finally {
    // блок кода, который обязательно выполнится по завершении блока try
}
```

## Встроенные в Java исключения

В стандартном пакете `java.lang` определен ряд классов исключений, большинство из которых относятся к подклассам стандартного типа `RuntimeException`, их не обязательно включать в список оператора `throws` в объявлении метода, такие исключения называются непроверяемыми(`unchecked`), поскольку компилятор не проверяет, обрабатываются или генерируются они в каком-нибудь методе. Кроме этого существуют исключения, которые должны быть включены в список оператора `throws` в объявлении методов, способных генерировать их, но не обрабатывать самостоятельно. Такие исключения называются проверяемыми (`checked`). Ниже приведены некоторые часто встречающиеся подклассы непроверяемых исключений, производные от класса `RuntimeException`.

Тип исключения	Описание
<code>ArithmeticException</code>	Арифметическая ошибка
<code>ArrayIndexOutOfBoundsException</code>	Выход индекса за пределы массива
<code>ArrayStoreException</code>	Присваивание элементу массива объекта несовместимого типа
<code>ClassCastException</code>	Неверное приведение типов
<code>IllegalArgumentException</code>	Употребление недопустимого аргумента при вызове метода
<code>IndexOutOfBoundsException</code>	Выход индекса некоторого типа за допустимые пределы
<code>NegativeArraySizeException</code>	Создание массива отрицательного размера
<code>NullPointerException</code>	Неверное использование пустой ссылки
<code>NumberFormatException</code>	Неверное преобразование символьной строки в числовой формат

## Создание собственных подклассов исключений

Для создания собственного класса исключений, достаточно определить его как производный от класса `Exception`. В подклассах собственных исключений совсем не обязательно реализовать что-нибудь. Их присутствия в системе типов уже достаточно, чтобы пользоваться ими как исключениями.

## Многократный перехват исключений

Многократный перехват позволяет обрабатывать несколько исключений в одном и том же операторе `catch`, при условии, что для этого используется одинаковый код. Для организации такого перехвата достаточно объединить типы исключений в операторе `catch` с помощью логической операции ИЛИ.

```
try {
    ...
} catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
    ...
}
```

## Домашнее задание

- 1 Напишите метод, на вход которого подаётся двумерный строковый массив размером 4x4, при подаче массива другого размера необходимо бросить исключение `MyArraySizeException`.
- 2 Далее метод должен пройтись по всем элементам массива, преобразовать в `int`, и просуммировать. Если в каком-то элементе массива преобразование не удалось (например, в ячейке лежит символ или текст вместо числа), должно быть брошено исключение `MyArrayDataException`, с детализацией в какой именно ячейке лежат неверные данные.
- 3 В методе `main()` вызвать полученный метод, обработать возможные исключения `MySizeArrayException` и `MyArrayDataException`, и вывести результат расчета.

## Дополнительные материалы

- 1 Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. - М.: Вильямс, 2014. - 864 с.
- 2 Брюс Эккель Философия Java // 4-е изд.: Пер. с англ. – СПб.: Питер, 2016. – 1168 с.
- 3 Г. Шилдт Java 8. Полное руководство // 9-е изд.: Пер. с англ. - М.: Вильямс, 2015. - 1376 с.
- 4 Г. Шилдт Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. - М.: Вильямс, 2015. - 720 с.

## Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт Java 8. Полное руководство // 9-е изд.: Пер. с англ. - М.: Вильямс, 2015. - 1376 с.