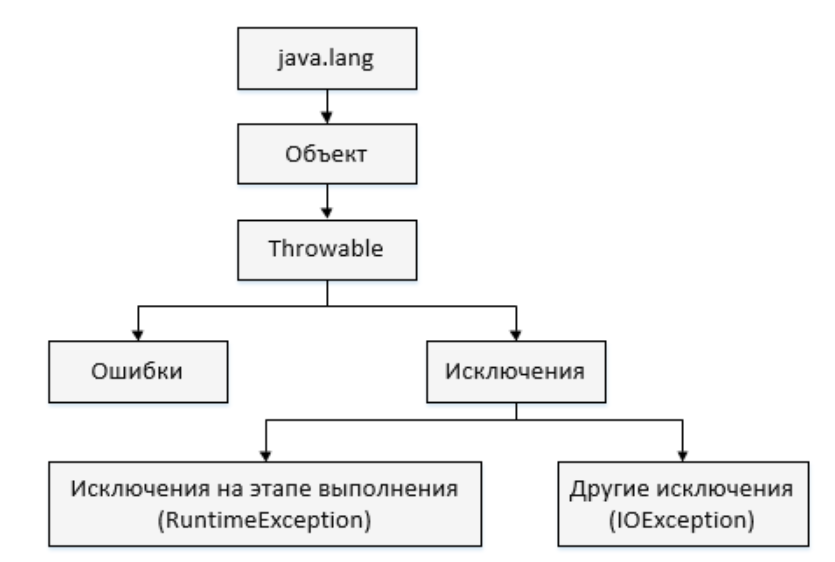
**Java — Исключения**

**Исключение в Java** — представляет проблему, которая возникает в ходе выполнения программы. В случае возникновения в Java исключения (exception), или исключительного события, имеет место прекращение нормального течения программы, и программа/приложение завершаются в аварийном режиме, что не является рекомендованным, и, как следствие, подобные случаи требуют в Java обработку исключений.

* **Контролируемые исключения** — контролируемое исключение представляет собой вид исключения, которое происходит на стадии компиляции, их также именуют исключениями периода компиляции. Обозначенные исключения не следует игнорировать в ходе компиляции, они требуют должного обращения (разрешения) со стороны программиста.
* **Неконтролируемые исключения** — неконтролируемое исключение представляет собой исключение, которое происходит во время выполнения. Они также носят название **исключения на этапе выполнения**. Данная категория может включать погрешности программирования, такие как логические ошибки либо неверный способ использования API. Исключения на этапе выполнения игнорируются в ходе компиляции.
* **Ошибки** — не являются исключениями, однако представляют проблемы, которые возникают независимо от пользователя либо программиста. Ошибки в вашем коде обычно игнорируются в виду того, что в редких случаях их обработка окажется результативной. К примеру, ошибка возникнет при переполнении стека. На этапе компиляции они также игнорируются.

Все классы исключений в Java представляют подтипы класса java.lang.Exception. Класс исключений является подклассом класса Throwable. Помимо класса исключений существует также подкласс ошибок, образовавшихся из класса Throwable.

Ошибки представляют аварийное состояние вследствие значительных сбоев, которые не обрабатываются программами Java.



|  |  |
| --- | --- |
| **№** | **Метод и описание** |
| 1 | **public String getMessage()** Возврат подробного сообщения о произошедшем исключении. Инициализация данного сообщения производится в конструкторе Throwable. |
| 2 | **public Throwable getCause()** Возврат причины исключения, представленной объектом Throwable. |
| 3 | **public String toString()** Возврат имени класса, соединенного с результатом getMessage(). |
| 4 | **public void printStackTrace()** Выведение результата toString() совместно с трассировкой стека в System.err, поток вывода ошибок. |
| 5 | **public StackTraceElement [] getStackTrace()** Возврат массива, содержащего каждый элемент в трассировке стека. Элемент с номером 0 представляет вершину стека вызовов, последний элемент массива отображает метод на дне стека вызовов. |
| 6 | **public Throwable fillInStackTrace()** Заполняет трассировку стека данного объекта Throwable текущей трассировкой стека, дополняя какую-либо предшествующую информацию в трассировке стека. |

## Обработка исключений — try и catch

Метод производит обработку исключения при использовании ключевых слов **try** и **catch**.

### Описание

Блок try/catch размещается в начале и конце кода, который может сгенерировать исключение. Код в составе блока try/catch является защищенным кодом, синтаксис использования try/catch выглядит следующим образом:

try {

// Защищенный код

}catch(НазваниеИсключения e1) {

// Блок catch

}

Код, предрасположенный к исключениям, размещается в блоке try. В случае возникновения исключения, обработка данного исключения будет производиться соответствующим блоком catch. За каждым блоком try должен немедленно следовать блок catch либо блок finally.

Оператор catch включает объявление типа исключения, которое предстоит обработать. При возникновении исключения в защищенном коде, блок catch (либо блоки), следующий за try, будет проверен. В случае, если тип произошедшего исключения представлен в блоке catch, исключение передается в блок catch аналогично тому, как аргумент передается в параметр метода.

## Многократные блоки catch

За блоком try могут следовать несколько блоков catch. Синтаксис многократных блоков catch выглядит следующим образом:

try {

// Защищенный код

}catch(ИсключениеТип1 e1) {

// Блок catch

}catch(ИсключениеТип2 e2) {

// Блок catch

}catch(ИсключениеТип3 e3) {

// Блок catch

}

Представленные выше операторы демонстрируют три блока catch, однако, после однократного try количество данных используемых блоков может быть произвольным. В случае возникновения исключения в защищенном коде, исключение выводится в первый блок catch в списке. Если тип данных генерируемого исключения совпадает с ИсключениеТип1, он перехватывается в указанной области. В обратном случае, исключение переходит ко второму оператору catch. Это продолжается до тех пор, пока не будет произведен перехват исключения, либо оно не пройдет через все операторы, в случае чего выполнение текущего метода будет прекращено, и исключение будет перенесено к предшествующему методу в стеке вызовов.

## Перехват многотипных исключений

В среде Java 7, Вы можете произвести обработку более чем одного исключения при использовании одного блока catch, данное свойство упрощает код. Ниже представлена модель реализации:

catch (IOException|FileNotFoundException ex) {

logger.log(ex);

throw ex;

## Ключевые слова throws/throw

В случае если метод не может осуществить обработку контролируемого исключения, производится соответствующее уведомление при использовании ключевого слова throws в Java. Ключевое слово throws появляется в конце сигнатуры метода.

При использовании ключевого слова throw вы можете произвести обработку вновь выявленного исключения либо исключения, которое было только что перехвачено.

Следует внимательно различать ключевые слова throw и throws в Java, так как throws используется для отложенной обработки контролируемого исключения, а throw, в свою очередь, используется для вызова заданного исключения.

Представленный ниже метод отображает, что им генерируется RemoteException:

import java.rmi.RemoteException;

public class Test {

public void deposit(double amount) throws RemoteException {

// Реализация метода

throw new RemoteException();

}

// Остаток определения класса

}

Метод также может объявить о том, что им генерируется более чем одно исключение, в случае чего исключения представляются в виде перечня, отделенные друг от друга запятыми. К примеру, следующий метод оповещает о том, что им генерируются RemoteException и InsufficientFundsException:

import java.rmi.RemoteException;

public class Test {

public void withdraw(double amount) throws RemoteException,

InsufficientFundsException {

// Реализация метода

}

// Остаток определения класса

}

## Блок finally

В Java finally следует за блоком try либо блоком catch. Блок finally в коде выполняется всегда независимо от наличия исключения.

Использование блока finally позволяет запустить какой-либо оператор, предназначенный для очистки, не зависимо от того, что происходит в защищенном коде.

Блок finally в Java появляется по окончании блоков catch, его синтаксис выглядит следующим образом:

### Синтаксис

try {

// Защищенный код

}catch(ИсключениеТип1 e1) {

// Блок catch

}catch(ИсключениеТип2 e2) {

// Блок catch

}catch(ИсключениеТип3 e3) {

// Блок catch

}finally {

// Блок finally всегда выполняется.

}

### Пример

public class Test {

public static void main(String args[]) {

int array[] = new int[2];

try {

System.out.println("Доступ к третьему элементу:" + array[3]);

}catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Исключение:" + e);

}finally {

array[0] = 6;

System.out.println("Значение первого элемента: " + array[0]);

System.out.println("Оператор finally выполнен.");

}

}

}

Вследствие этого будет получен следующий результат:

Исключение:java.lang.ArrayIndexOutOfBoundsException: 3

Значение первого элемента: 6

Оператор finally выполнен.

Следует помнить, что:

* Выражение catch не может существовать без оператора try.
* При наличии блока try/catch, выражение finally не является обязательным.
* Блок try не может существовать при отсутствии выражения catch либо выражения finally.
* Существование какого-либо кода в промежутке между блоками try, catch, finally является невозможным.

## Конструкция try-with-resources

В норме, при использовании различных видов ресурсов, таких как потоки, соединения и др., нам предстоит закрыть их непосредственно при использовании блока finally. В программе, представленной ниже, нами производится считывание данных из файла при использовании FileReader, после чего он закрывается блоком finally.

### Пример 1

import java.io.FileReader;

import java.io.File;

import java.io.IOException;

public class Test {

public static void main(String args[]) {

FileReader fr = null;

try {

File f = new File("file.txt");

fr = new FileReader(f);

char [] array = new char[10];

fr.read(array); // чтение содержимого массива

for(char c : array)

System.out.print(c); // вывод символов на экран, один за одним

}catch(IOException e1) {

e1.printStackTrace();

}finally {

try {

fr.close();

}catch(IOException e2) {

e2.printStackTrace();

}

}

}

}

Конструкция try-with-resources, также именуемая как **автоматическое управление ресурсами**, представляет новый механизм обработки исключений, который был представлен в 7-ой версии Java, осуществляя автоматическое закрытие всех ресурсов, используемых в рамках блока try catch.

Чтобы воспользоваться данным оператором, вам всего лишь нужно разместить заданные ресурсы в круглых скобках, после чего созданный ресурс будет автоматически закрыт по окончании блока. Ниже представлен синтаксис конструкции try-with-resources.

### Синтаксис

try(FileReader fr = new FileReader("Путь к файлу")) {

// использование ресурса

}catch() {

// тело catch

}

}

Программа ниже производит считывание данных в файле, используя конструкцию try-with-resources.

### Пример 2

import java.io.FileReader;

import java.io.IOException;

public class Test {

public static void main(String args[]) {

try(FileReader fr = new FileReader("E://Soft/NetBeans 8.2/Projects/test/test/file.txt")) {

char [] array = new char[10];

fr.read(array); // чтение содержимого массива

for(char c : array)

System.out.print(c); // вывод символов на экран, один за одним

}catch(IOException e) {

e.printStackTrace();

}

}

}

При работе с конструкцией try-with-resources следует принимать во внимание следующие нюансы:

* С целью использования конструкции try-with-resources следует реализовать интерфейс AutoCloseable, после чего соответствующий метод close() будет вызван автоматически во время выполнения.
* В конструкции try-with-resources возможно указание одного и более классов.
* При указании нескольких классов в блоке try конструкции try-with-resources, закрытие данных классов будет производиться в обратном порядке.
* За исключением внесения ресурсов в скобки, все элементы являются равными аналогично нормальному блоку try/catch в составе блока try.
* Ресурсы, внесенные в try, конкретизируются до запуска блока try.
* Ресурсы непосредственно в составе блока try указываются как окончательные.

## Создание своих собственных исключений

Вы можете создать свои собственные исключения в среде Java. При записи собственных классов исключений следует принимать во внимание следующие аспекты:

* Все исключения должны быть дочерними элементами Throwable.
* Если вы планируете произвести запись контролируемого исключения с автоматическим использованием за счет правила обработки или объявления, вам следует расширить класс Exception.
* Если вы хотите произвести запись исключения на этапе выполнения, вам следует расширить класс RuntimeException.

Вы можете определить собственный класс исключений, как показано ниже:

class MyException extends Exception {

}

Вам лишь необходимо расширить предопределенный класс Exception с целью создания собственного исключения. Данная категория относится к контролируемым исключениям. Следующий класс InsufficientFundsException исключительных ситуаций, определяемых пользователем, расширяет класс Exception, делая его контролируемым исключением. Класс исключений, подобно всем остальным классам, содержит используемые области и методы.

### Пример

// Название файла InsufficientFundsException.java

import java.io.\*;

public class InsufficientFundsException extends Exception {

private double amount;

public InsufficientFundsException(double amount) {

this.amount = amount;

}

public double getAmount() {

return amount;

}

}

С целью демонстрации наших исключений, определяемых пользователем, следующий класс Checking содержит метод withdraw(), генерирующий InsufficientFundsException.

// Название файла Checking.java

import java.io.\*;

public class Checking {

private int number;

private double balance;

public Checking(int number) {

this.number = number;

}

public void deposit(double amount) {

balance += amount;

}

public void withdraw(double amount) throws InsufficientFundsException {

if(amount <= balance) {

balance -= amount;

}else {

double needs = amount - balance;

throw new InsufficientFundsException(needs);

}

}

public double getBalance() {

return balance;

}

public int getNumber() {

return number;

}

}

Следующая программа Bank демонстрирует вызов методов deposit() и withdraw() класса Checking.

// Название файла Bank.java

public class Bank {

public static void main(String [] args) {

Checking c = new Checking(101);

System.out.println("Депозит $300...");

c.deposit(300.00);

try {

System.out.println("\nСнятие $100...");

c.withdraw(100.00);

System.out.println("\nСнятие $400...");

c.withdraw(400.00);

}catch(InsufficientFundsException e) {

System.out.println("Извините, но у Вас $" + e.getAmount());

e.printStackTrace();

}

}

}

## Общие исключения

В Java можно выделить две категории исключений и ошибок.

* **Исключения JVM** — данная группа представлена исключениями/ошибками, которые вызываются непосредственно и логически со стороны JVM. Примеры: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.
* **Программные исключения** — данные исключения вызываются непосредственно приложением либо программистами API. Примеры: IllegalArgumentException, IllegalStateException.

# Java — Встроенные исключения

Java определяет несколько классов исключений внутри стандартного пакета **java.lang**.

Наиболее общие из этих исключений являются подклассами стандартного типа RuntimeException. Поскольку java.lang неявно импортируется во все java-программы, то большинство исключений, полученных из RuntimeException, автоматические.

Java определяет несколько других типов исключений, которые относятся к его различным библиотекам класса. Ниже приведен список неконтролируемых исключений на этапе выполнения (Unchecked RuntimeException).

|  |  |
| --- | --- |
| **№** | **Исключение и описание** |
| 1 | **java.lang.ArithmeticException** Арифметическая ошибка, например, деление на ноль. |
| 2 | **java.lang.ArrayIndexOutOfBoundsException** Индекс массива выходит за пределы. |
| 3 | **java.lang.ArrayStoreException** Присвоение элементу массива несовместимого типа. |
| 4 | **java.lang.ClassCastException** Недопустимое приведение типов. |
| 5 | **java.lang.IllegalArgumentException** Недопустимый аргумент, используемый для вызова метода. |
| 6 | **java.lang.IllegalMonitorStateException** Недопустимая работа монитора, например, ожидание разблокированного потока. |
| 7 | **java.lang.IllegalStateException** Окружающая обстановка или приложение находится в неправильном состоянии. |
| 8 | **java.lang.IllegalThreadStateException** Запрошенная операция несовместима с текущим состоянием потока. |
| 9 | **java.lang.IndexOutOfBoundsException** Некоторый тип индекса находится за пределом. |
| 10 | **java.lang.NegativeArraySizeException** Массив создан с отрицательным размером. |
| 11 | **java.lang.NullPointerException** Недопустимое использование нулевой ссылки. |
| 12 | **java.lang.NumberFormatException** Неверное преобразование строки в числовой формат. |
| 13 | **java.lang.SecurityException** Попытка нарушить безопасность. |
| 14 | **java.lang.StringIndexOutOfBounds** Попытка индексирования за пределами строки. |
| 15 | **java.lang.UnsupportedOperationException** Была обнаружена неподдерживаемая операция. |

Ниже приведен список контролируемых исключений (Checked Exceptions) в Java, определенных в java.lang.

|  |  |
| --- | --- |
| **№** | **Исключение и описание** |
| 1 | **java.lang.ClassNotFoundException** Класс не найден. |
| 2 | **java.lang.CloneNotSupportedException** Попытка клонировать объект, который не реализует Cloneable интерфейс. |
| 3 | **java.lang.IllegalAccessException** Запрещен доступ к классу. |
| 4 | **java.lang.InstantiationException** Попытка создать объект абстрактного класса или интерфейса. |
| 5 | **java.lang.InterruptedException** Один поток был прерван другим потоком. |
| 6 | **java.lang.NoSuchFieldException** Запрошенное поле не существует. |
| 7 | **java.lang.NoSuchMethodException** Запрошенный метод не существует. |