**24) Как вывести строку в обратной последовательности символов? Какой оператор позволяет принудительно выбросить исключение? Есть ли дополнительные условия к методу, который потенциально может выбросить исключение?**

**1)Способы перевернуть строку в Java**

В примерах ниже мы попробуем вывести строку в реверсном режиме разными способами, а именно: рекурсивно и с использованием методов charAt(), toCharArray() и StringBuffer().

**Рекурсивный метод**

Первым делом давайте попробуем реализовать вывод строки в обратном порядке с помощью «велосипедной рекурсии» :).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public static String reverseStringWithRecursion(String inputString) {          String rightPart;          String leftPart;            int length = inputString.length();            // заканчиваем рекурсивный обход          if (length <= 1) {              return inputString;          }            leftPart = inputString.substring(0, length / 2);            rightPart = inputString.substring(length / 2, length);            // рекурсивно переворачиваем левую и правую часть входной строки          return reverseStringWithRecursion(rightPart) + reverseStringWithRecursion(leftPart);      } |

Суть этого метода сводится к тому, чтобы рекурсивно разделять входную строку на две части, уменьшая с каждым разом длину входной строки. Как только длина этой строки уменьшится до одного символа, это будет означать, что переворачивание фрагмента закончено и эти фрагменты будут собраны воедино. В итоге на выход мы получаем новую строку в обратном порядке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // пример работы рекурсивного способа  // вручную считаем для строки "java":  При первом проходе длина строки = 4, а левая и правая части:  leftString = "ja"  rightString = "va"  // проход закончен так:  return "va" + "ja"    Теперь "va" + "ja" уже по отдельности опять проходят через этот метод:  пример для "va":  leftString = "v"  rightString = "a"  // проход закончен так:  return "a" + "v"    пример для "ja":  leftString = "j"  rightString = "a"  // проход закончен так:  return "a" + "j"    в итоге на выход получаем "avaj" |

Рекурсивный метод на выход выдаст тот же результат.

**Переворачиваем строку с помощью charAt()**

Другим способом является проход по символам в строке с помощью метода charAt() и сохранение строки с обратным порядком символов. Смотрим на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public static String reverseStringWithCharAt(String inputString) {          int stringLength = inputString.length();          String result = "";          for (int i = 0; i < stringLength; i++) {              result = inputString.charAt(i) + result;          }          return result;      } |

В коде выше мы [в цикле](https://javadevblog.com/primer-ispol-zovaniya-operatora-break-v-for-while-do-while-tsiklah.html) обходим входную строку и по индекску забираем символ. Далее пересоздаем результирующую строку с новым символом вначале.

**Выводим строку в обратном порядке с помощью toCharArray()**

Этот способ очень похож на предыдущий, но вместо [charAt](https://javadevblog.com/kak-preobrazovat-stroku-string-v-simvol-char-na-java.html" \t "_blank)() мы представляем строку в виде массива символов используя метод toCharArray() и затем в обратном порядке присваиваем символы результирующей строке. Смотрим пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public static String reverseWithCharArray(String inputString) {          char[] charArray = inputString.toCharArray();          String resultString = "";          for (int i = charArray.length - 1; i >= 0; i--) {              resultString += charArray[i];          }          return resultString;      } |

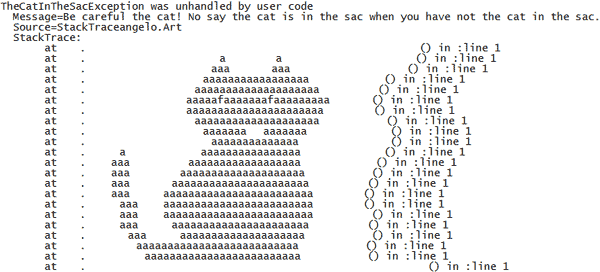
**Используем StringBuffer или StringBuilder для переворачивания строки в Java**

В этом способе мы воспользуемся методами для вывода строки в обратном порядке, которые строенны в Java и не требуеют с нашей стороны никаких обработок. Все, что нам нужно использовать — это класс [StringBuffer](https://javadevblog.com/rekomendatsii-po-ispol-zovaniyu-string-stringbuffer-i-stringbuilder-v-java.html" \t "_blank) и его метод reverse():

|  |  |
| --- | --- |
| 1  2 | String inputString = "java";  String reversedString = new StringBuffer(inputString).reverse().toString(); |

Вот так в две строчки кода мы избавились от необходимости писать отдельные методы для переворота строки — Java за нас делает всю работу. Интересно знать, что внутри метода reverse() происходит сложный процес создания новой перевернутой строки со множеством проверок и оптимизаций быстродействия, о которых мы не задумывались в наших тестовых программках.

# 2) **Исключения**



[try](http://developer.alexanderklimov.ru/android/java/exception.php#try)  
[Оператор throw](http://developer.alexanderklimov.ru/android/java/exception.php#throw)  
[Оператор throws](http://developer.alexanderklimov.ru/android/java/exception.php#throws)  
[Оператор finally](http://developer.alexanderklimov.ru/android/java/exception.php#finally)  
[Встроенные исключения Java](http://developer.alexanderklimov.ru/android/java/exception.php#system)  
[Создание собственных классов исключений](http://developer.alexanderklimov.ru/android/java/exception.php#custom)

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 нуль 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.

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 **Trowable**. 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 (как и нормальному любопытному коту) 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 незадача, никто не перехватывает его, хотя это должны были сделать вы. Видя вашу бездеятельность, объект перехватывает стандартный системный обработчик Java, который отличается вредных 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 ёго.

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**.

## **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 должные 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**.

Мы могли бы написать такой код для кнопки:

Cat cat;

public void onClick(View view) {

if(cat == null){

throw new NullPointerException("Котик не инициализирован");

}

}

Мы объявили объект класса **Cat**, но забыли его проинициализировать, например, в **onCreate()**. Теперь нажатие кнопки вызовет исключение, которое обработает система, а в логах мы можем прочитать сообщение об ошибке. Возможно, вы захотите использовать другое исключение, например, **throw new UnsupportedOperationException("Котик не инициализирован");**.

В любом случае мы передали обработку ошибки системе. В реальном приложении вам нужно обработать ошибку самостоятельно.

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

Перепишем пример с обработкой ошибки.

public void onClick(View view) {

if (cat == null) {

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 с 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**.

Теперь программа не закроется аварийно, а будет просто выводить сообщения в всплывающих **Toast**.

## **Оператор throws**

Если метод может породить исключение, которое он сам не обрабатывает, он должен задать это поведение так, чтобы вызывающий его код мог позаботиться об этом исключении. Для этого к объявлению метода добавляется конструкция **throws**, которая перечисляет типы исключений (кроме исключений Error и RuntimeException и их подклассов).

Общая форма объявления метода с оператором **throws**:

тип имя\_метода(список\_параметров) throws список\_исключений {

// код внутри метода

}

В фрагменте *список\_исключений* можно указать список исключений через запятую.

Создадим метод, который может породить исключение, но не обрабатывает его. А в щелчке кнопки вызовем его.

// Метод без обработки исключения

public void createCat(){

Toast.makeText(this, "Вы создали котёнка", Toast.LENGTH\_LONG).show();

throw new NullPointerException("Кота не существует");

}

// Щелчок кнопки

public void onClick(View v) {

createCat();

}

Если вы запустите пример, то получите ошибку. Исправим код.

// Без изменений

public void createCat() throws NullPointerException {

Toast.makeText(this, "Вы создали котёнка", Toast.LENGTH\_LONG).show();

throw new NullPointerException("Кота не существует");

}

// Щелчок кнопки

public void onClick(View v) {

try {

createCat();

} catch (NullPointerException e) {

// TODO: handle exception

Toast.makeText(this, e.getMessage(), Toast.LENGTH\_LONG).show();

}

}

Мы поместили вызов метода в блок **try** и вызвали блок **catch** с нужным типом исключения. Теперь ошибки не будет.

## **Оператор finally**

Когда исключение передано, выполнение метода направляется по нелинейному пути. Это может стать источником проблем. Например, при входе метод открывает файл и закрывает при выходе. Чтобы закрытие файла не было пропущено из-за обработки исключения, был предложен механизм **finally**.

Ключевое слово **finally** создаёт блок кода, который будет выполнен после завершения блока **try/catch**, но перед кодом, следующим за ним. Блок будет выполнен, независимо от того, передано исключение или нет. Оператор **finally** не обязателен, однако каждый оператор **try** требует наличия либо **catch**, либо **finally**.

## **Встроенные исключения Java**

Существуют несколько готовых системных исключений. Большинство из них являются подклассами типа **RuntimeException** и их не нужно включать в список **throws**. Вот небольшой список непроверяемых исключений.

* ArithmeticException - арифметическая ошибка, например, деление на нуль
* ArrayIndexOutOfBoundsException - выход индекса за границу массива
* ArrayStoreException - присваивание элементу массива объекта несовместимого типа
* ClassCastException - неверное приведение
* EnumConstantNotPresentException - попытка использования неопределённого значения перечисления
* IllegalArgumentException - неверный аргумент при вызове метода
* IllegalMonitorStateException - неверная операция мониторинга
* IllegalStateException - некорректное состояние приложения
* IllegalThreadStateException - запрашиваемая операция несовместима с текущим потоком
* IndexOutofBoundsException - тип индекса вышел за допустимые пределы
* NegativeArraySizeException - создан массив отрицательного размера
* NullPointerException - неверное использование пустой ссылки
* NumberFormatException - неверное преобразование строки в числовой формат
* SecurityException - попытка нарушения безопасности
* StringIndexOutOfBounds - попытка использования индекса за пределами строки
* TypeNotPresentException - тип не найден
* UnsupportedOperationException - обнаружена неподдерживаемая операция

Список проверяемых системных исключений, которые можно включать в список **throws**.

* ClassNotFoundException - класс не найден
* CloneNotSupportedException - попытка клонировать объект, который не реализует интерфейс **Cloneable**
* IllegalAccessException - запрещен доступ к классу
* InstantiationException - попытка создать объект абстрактного класса или интерфейса
* InterruptedException - поток прерван другим потоком
* NoSuchFieldException - запрашиваемое поле не существует
* NoSuchMethodException - запрашиваемый метод не существует
* ReflectiveOperationException - исключение, связанное с рефлексией

## **Создание собственных классов исключений**

Система не может предусмотреть все исключения, иногда вам придётся создать собственный тип исключения для вашего приложения. Вам нужно наследоваться от **Exception** (напомню, что этот класс наследуется от **Trowable**) и переопределить нужные методы класса **Throwable**. Либо вы можете наследоваться от уже существующего типа, который наиболее близок по логике с вашим исключением.

* final void addSuppressed(Throwable exception) - добавляет исключение в список подавляемых исключений (JDK 7)
* Throwable fillInStackTrace() - возвращает объект класса **Throwable**, содержащий полную трассировку стека.
* Throwable getCause() - возвращает исключение, лежащее под текущим исключение или null
* String getLocalizedMessage() - возвращает локализованное описание исключения
* String getMessage() - возвращает описание исключения
* StackTraceElement[] getStackTrace() - возвращает массив, содержащий трассировку стека и состояний из элементов класса **StackTraceElement**
* final Throwable[] getSuppressed() - получает подавленные исключения (JDK 7)
* Throwable initCause(Throwable exception) - ассоциирует исключение с вызывающим исключением. Возвращает ссылку на исключение.
* void printStackTrace() - отображает трассировку стека
* void printStackTrace(PrintStream stream) - посылает трассировку стека в заданный поток
* void printStackTrace(PrintWriter stream) - посылает трассировку стека в заданный поток
* void setStackTrace(StackTraceElement elements[]) - устанавливает трассировку стека для элементов (для специализированных приложений)
* String toString() - возвращает объект класса **String**, содержащий описание исключения.

Самый простой способ - создать класс с конструктором по умолчанию.

// Если этот код работает, его написал Александр Климов,

// а если нет, то не знаю, кто его писал.

package ru.alexanderklimov.exception;

import android.os.Bundle;

import android.support.v7.app.AppCompatActivity;

import android.view.View;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

public void testMethod() throws HungryCatException{

System.out.println("Возбуждаем HungryCatException из метода testMethod()");

throw new HungryCatException(); // конструктор по умолчанию

}

public void onClick(View view) {

try {

testMethod();

} catch (HungryCatException e) {

e.printStackTrace();

System.out.println("Наше исключение перехвачено");

}

}

class HungryCatException extends Exception{

}

}

Мы создали собственный класс **HungryCatException**, в методе **testMethod()** его возбуждаем, а по нажатию кнопки вызываем этот метод. В результате наше исключение сработает.

Создать класс исключения с конструктором, который получает аргумент-строку, также просто.

// Если этот код работает, его написал Александр Климов,

// а если нет, то не знаю, кто его писал.

package ru.alexanderklimov.exception;

import android.os.Bundle;

import android.support.v7.app.AppCompatActivity;

import android.view.View;

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

public void testMethod() throws HungryCatException {

System.out.println("Возбуждаем HungryCatException из метода testMethod()");

throw new HungryCatException(); // конструктор по умолчанию

}

public void testMethod2() throws HungryCatException {

System.out.println("Возбуждаем HungryCatException из метода testMethod2()");

throw new HungryCatException("Создано во втором методе");

}

public void onClick(View view) {

try {

testMethod();

} catch (HungryCatException e) {

e.printStackTrace();

System.out.println("Наше исключение перехвачено");

}

try {

testMethod2();

} catch (HungryCatException e) {

e.printStackTrace();

}

}

class HungryCatException extends Exception {

HungryCatException() {

}

HungryCatException(String msg) {

super(msg);

}

}

}

Ещё вариант. Добавим также метод **toString()**.

class CustomException extends Exception {

String message;

CustomException(String str) {

message = str;

}

public String toString() {

return ("Custom Exception Occurred: " + message);

}

}

// где-то вызываем

try {

throw new CustomException("This is a custom message");

} catch (CustomException e) {

System.out.println(e);

}

Теперь класс содержит два конструктора. Во втором конструкторе используется конструктор родительского класса с аргументом **String**, вызываемый ключевым словом **super**.

## **Перехват произвольных исключений**

Можно создать универсальный обработчик, перехватывающий любые типы исключения. Осуществляется это перехватом базового класса всех исключений **Exception**:

cacth(Exception e) {

Log.w("Log", "Перехвачено исключение");

}

Подобная конструкция не упустит ни одного исключения, поэтому её следует размещать в самом конце списка обработчиков, во избежание блокировки следующих за ней обработчиков исключений.

## **Основные правила обработки исключений**

Используйте исключения для того, чтобы:

* обработать ошибку на текущем уровне (избегайте перехватывать исключения, если не знаете, как с ними поступить)
* исправить проблему и снова вызвать метод, возбудивший исключение
* предпринять все необходимые действия и продолжить выполнение без повторного вызова действия
* попытаться найти альтернативный результат вместо того, который должен был бы произвести вызванный метод
* сделать все возможное в текущем контексте и заново возбудить это же исключение, перенаправив его на более высокий уровень
* сделать все, что можно в текущем контексте, и возбудить новое исключение, перенаправив его на более высокий уровень
* завершить работу программы
* упростить программу (если используемая схема обработки исключений делает все только сложнее, значит, она никуда не годится)
* добавить вашей библиотеке и программе безопасности