



# Java course

- [Начало Java](#)
- [Проект «Отдел кадров»](#)
- [Курсы](#)
- [Статьи](#)
- [Контакты/Вопросы](#)

- [Введение](#)
- [Установка JDK](#)
- [Основные шаги](#)
- [Данные](#)
- [Порядок операций](#)
- [IDE NetBeans](#)
- [ООП](#)
- [Инкапсуляция](#)
- [Наследование](#)
- [Пакеты](#)
- [Переопределение и перегрузка](#)
- [Полиморфизм](#)
- [Статические свойства и методы](#)
- [Отношения между классами](#)
- [Визуализация работа](#)
- [Пример — очередь объектов](#)
- [Массивы — знакомство](#)
- [Многомерные массивы](#)
- [Абстрактные классы](#)
- [Интерфейсы](#)

## Исключения

Причиной появления исключений, на мой взгляд, стало очень простое предположение — во время исполнения программы вряд ли можно быть уверенным, что все функции будут делать свою работу без сбоев. Например, если ваш код должен прочитать файл с каким-то именем, а файла такого просто нет ? Или вы хотите по сети обратиться к другой программе, но добрая старушка-уборщица выдернула шнур из компьютера ? В конце концов ваш калькулятор при вычислении дроби получил в знаменателе ноль. Да много ли каких еще препятствий может встретиться на пути.

Конечно, не все методы/функции обязательно будут оканчиваться крахом, но это обычное дело. И что в этом случае необходимо предпринять ? В старое доброе время в языке Си было принято возвращать какое-то значение. Причем в разных случаях тот же ноль мог означать что все хорошо, но мог означать что все плохо. В общем стандарта как не было, так до сих пор и нет — что должна возвращать функция в случае возникновения ошибки. Определенно можно сказать только одно — функция должна возвращать:

- а) признак того, что произошла ошибка
- б) информацию о том, а что собственно плохого произошло

С пунктом б) как раз достаточно просто — у нас же ООП. Значит удобно всю необходимую информацию поместить в какой-либо объект какого-то класса и вернуть. В этом объекте мы можем развернуться по полной — и сообщение написать и код ошибки выдать и много чего еще.

Что касается пункта а), то несложно догадаться, что возвращать ошибку как результат выполнения

- [Расширенное описание классов](#)
- [Исключения](#)
- [Решения на основе классов](#)
- [Список контактов — начало](#)
- [Коллекции — базовые принципы](#)
- [Коллекции — продолжение](#)
- [Список контактов — GUI приложение](#)
- [Что такое JAR-файлы](#)
- [Многопоточность — первые шаги](#)
- [Многопоточность и синхронизация](#)
- [Работаем с XML](#)
- [Reflection — основы](#)
- [Установка СУБД PostgreSQL](#)
- [Базы данных на Java — первые шаги](#)
- [Возможности JDBC — второй этап](#)
- [JDBC — групповые операции](#)
- [Список контактов — работаем с БД](#)
- [Переезжаем на Maven](#)
- [Потоки ввода-вывода](#)
- [Сетевое взаимодействие](#)
- [С чего начинается Web](#)

функции — плохая идея. Если мы ожидаем целое число, а ошибка будет описываться пусть даже просто строкой, то вряд ли кто-то сможет предложить что-то более-менее приемлемое. Но идея возвращать объект-описание ошибки — красивая и благодарная идея.

Отсюда родилась следующее решение — метод может генерировать **исключение** (объект-ошибку) и завершаться не вызовом **return** в случае ошибки, а иначе — метод будет «бросать» исключение.

И тогда код, вызывающий такой метод может получить ответ двумя способами:

1. Обычным возвратом значения из метода, если все хорошо
2. Будет «ловить» исключение

Соответственно метод, который хочет возвращать ошибку, тоже должен уметь возвращать какое-то значение (или ничего не возвращать в случае **void**) когда все хорошо и создавать и «бросать» исключения, если что-то пошло не так. Я уже дважды использовал термин «бросать», причем не без умысла — в языке Java используется глагол **throw** (бросать, кидать), посему больше не буду брать его в кавычки.

Мы получаем три момента для рассмотрения:

1. Описание объекта-исключения — его же надо уметь создавать
2. Описание метода, который умеет бросать исключения
3. Как правильно ловить исключение

Итак, еще раз — надо создать исключение, надо правильно бросить исключение, надо правильно поймать исключение.

## Класс для исключения

Научимся создавать исключения. Для таких классов существует специальная иерархия классов, которая начинается с класса [java.lang.Throwable](#). (Надеюсь вы помните, что java.lang означает пакет, в котором находится описание класса. По сути директория). Если перевести это название получится что-то вроде «готовый к бросанию» — с литературным переводом возможно у меня не очень красиво получилось, но идея именно такая: объекты этого класса (и всех потомков) могут быть брошены. Этот класс редко используется для прямого наследования, чаще используется его потомок, класс java.lang.Exception. А уж у этого класса «детишек» очень много. Как вы наверно уже догадались, количество готовых классов для Java огромно. Вряд ли конечно на каждого человека на земле приходится один класс, хотя возможно я не так уж и далек от истины. В общем, уже готовых классов-исключений среди других много. Так что будет что изучать. Но о них мы поговорим несколько позже, а пока все-таки создадим наш класс-исключение.

```

1 package edu.javacourse.exception;
2
3 public class SimpleException extends Exception
4 {
5     // Это наше поле для хранения информации, присущей данному

```

```

6      // классу-исключению. Поле немножко надуманное, но здесь может быть
7      // и достаточно важная информация
8      private int errorCode;
9
10     // переопределяем конструктор
11     public SimpleException(String message)
12     {
13         this(0, message);
14     }
15
16     // Создаем свой конструктор
17     public SimpleException(int errorCode, String message)
18     {
19         // Вызываем конструктор предка
20         super(message);
21         // Добавляем инициализацию своего поля
22         this.errorCode = errorCode;
23     }
24
25     // Метод для получения кода ошибки
26     public int getErrorCode()
27     {
28         return errorCode;
29     }
30 }

```

Как вы можете видеть ничего особенного в описании нет — создали класс, унаследовали его от класса **Exception**, определили свой конструктор, переопределили уже существующий — в общем ничего неординарного и загадочного. Единственное, что хотелось бы отметить — это наличие у класса **Throwable** (предка **Exception**) нескольких достаточно востребованных методов.

**getMessage()** — получить сообщение об ошибке, которое обычно имеет смысл читать

**printStackTrace()** — распечатать полный стек вызовов. Стек вызовов — это полный список всех методов внутри которых случилась ошибка. Т.е. если вызывался `method1`, внутри него `method2`, потом `method3` и в нем случилось исключение, то вы увидите все три метода. Чуть позже мы с вами посмотрим пример использования этого метода. Не пренебрегайте им — он очень удобный и информативный

## Метод для генерации исключения

Генерировать исключение можно в методе и этот метод может объявить, что он кидает исключения определенного класса. Делается это достаточно просто — после списка аргументов (в скобках) пишется ключевое слово **throws** и через запятую перечисляются классы исключений, который может порождать данный метод. Потом открывается фигурная скобка и мы пишем тело метода. Давайте посмотрим пример такого описания — наш класс **Generator** включает метод **helloMessage** который принимает в качестве строки имя, чтобы отдать строку «Hello, <имя>». Но если имя не указано (указатель на строку равен null), то метод не возвращает например пустую строку — он кидает исключение, которое можно использовать в дальнейшей логике.

```

1 package edu.javacourse.exception;
2
3 public class Generator
4 {
5     // Данный метод описан с указанием того, что он способен кинуть
6     // исключение типа SimpleException
7     public String helloMessage(String name) throws SimpleException
8     {
9         if (name == null) {
10             // Мы должны сначала создать объект-исключение
11             SimpleException se = new SimpleException(10, "Message is null");
12             // Теперь мы можем "кинуть" это исключение - это другой способ выйти
13             // из метода - отличный от варианта с return
14             throw se;
15
16             // Можно совместить создание и кидание - можете закомментировать
17             // предыдущие строки и использовать нижеприведенную
18             // throw new SimpleException(10, "Message is null");
19         }
20         return "Hello, " + name;
21     }
22 }

```

Опять смотрим код и видим, что после проверки на null мы создаем объект-исключение (обратите внимание — мы просто создаем нужный нам объект, заполняем его нужными значениями (мы в нем описываем реальную проблему, которая произошла) и кидаем. Опять же обращаю ваше внимание на то, что мы не делаем вызов **return** — этот вызов делается в случае если все хорошо. Мы кидаем исключение — пишем специальную конструкцию **throw** (не перепутайте — в описании метода пишем глагол в третьем лице единственного числа по английской грамматике с окончанием **s** — **throws** — «бросает», а в коде используем повелительное наклонение — **throw** — «бросай»).

Несколько позже мы рассмотрим еще несколько моментов, которые касаются описания методов, которые бросают исключения. Пока же перейдем к третьему шагу — как поймать исключение.

## Ловим исключение

Для того, чтобы поймать исключение используется конструкция **try ... catch**. Перед блоком кода, который порождает исключение пишем слово **try** и открываем фигурные скобки. После окончания блока закрываем скобки, пишем слово **catch**, в скобках указываем переменную класса-исключения, открываем скобки и там пишем код, который будет вызываться ТОЛЬКО в случае если метод/методы внутри **try** породили исключение. Смотрим код:

```

1 package edu.javacourse.exception;
2
3 public class Starter

```

```

4 {
5     public static void main(String[] args)
6     {
7         // создаем наш класс для генерации исключений
8         Generator generator = new Generator();
9
10        // Данный блок будет обрабатывать исключение
11        // и оно там действительно возникнет - мы же передали null
12        try {
13            String answer = generator.sendMessage(null);
14            System.out.println("Answer 1:" + answer);
15        } catch (SimpleException ex) {
16            // Здесь мы можем обработать объект-исключение,
17            // получить некоторую информаию
18            System.out.println("Error code:" + ex.getErrorCode());
19            System.out.println("Error message:" + ex.getMessage());
20        }
21
22        // Данный блок будет обрабатывать исключение
23        // но его не будет - мы передали корректный параметр
24        try {
25            String answer = generator.sendMessage("Yoda");
26            System.out.println("Answer 2:" + answer);
27        } catch (SimpleException ex) {
28            // Здесь мы можем обработать объект-исключение,
29            // получить некоторую информаию
30            System.out.println("Error:" + ex.getMessage());
31        }
32    }
33 }

```

Полный текст проекта можно скачать тут — [SimpleException](#). При запуске можно увидеть, что первый кусочек **try ... catch** выкинет исключение и мы увидим текст об ошибках

```

Error code:10
Error message:Message is null

```

Второй же пройдет гладко и мы увидим приветствие для самого известного джедая

```

Answer 2:Hello, Yoda

```

Обратите внимание, что в первом блоке **try ... catch** строка «Answer 1:» не выводится. Т.е. поведение при генерации исключения следующее: сразу после создания исключения все остальные строки внутри блока уже не выполняются и вы сразу перемещаетесь в блок **catch**. Думаю, что это

достаточно очевидно.

Итак, мы рассмотрели простой вариант использования исключений, но это конечно же не все. Так что продолжим.

## Блок **finally**

Как мы видели чуть выше, если во время исполнения внутри блока **try ... catch** случается исключение, то все оставшиеся строки НЕ ВЫПОЛНЯЮТСЯ. Но это не всегда полностью соответствует нашим желаниям. Существует немалое количество ситуаций, когда, чтобы не случилось, какие-то строки кода должны выполняться вне зависимости от того, каков результат. Именно для этого существует секция **finally**. Например, вы открыли файл на запись и долгое время что-то туда записывали. Но наступил момент, когда какие-то данные по определенным причинам туда не попали и запись надо прекращать. Но даже в этой ситуации файл нужно сохранить (закрыть). Или вы открыли сетевое соединение, которое в любом случае надо закрыть. Наконец вам просто надо всегда обнулить какой-то счетчик в конце. Давайте посмотрим структуру с **finally**

```
1 package edu.javacourse.exception;
2
3 public class Starter
4 {
5     public static void main(String[] args)
6     {
7         // создаем наш класс для генерации исключений
8         Generator generator = new Generator();
9
10        // Данный блок будет обрабатывать исключение
11        // и оно там действительно возникнет - мы же передали null
12        try {
13            String answer = generator.sendMessage(null);
14            System.out.println("Answer 1:" + answer);
15        } catch (SimpleException ex) {
16            // Здесь мы можем обработать объект-исключение,
17            // получить некоторую информация
18            System.out.println("Error code:" + ex.getErrorCode());
19            System.out.println("Error message:" + ex.getMessage());
20        } finally {
21            // Этот блок будет вызываться всегда, независимо от результата
22            System.out.println("Этот блок вызывается всегда");
23        }
24
25        // Данный блок будет обрабатывать исключение
26        // но его не будет - мы передали корректный параметр
27        try {
28            String answer = generator.sendMessage("Yoda");
29            System.out.println("Answer 2:" + answer);
30        } catch (SimpleException ex) {
31            // Здесь мы можем обработать объект-исключение,
```

```

32         // получить некоторую информаию
33         System.out.println("Error:" + ex.getMessage());
34     } finally {
35         // Этот блок будет вызываться всегда, независимо от результата
36         System.out.println("Этот блок вызывается всегда");
37     }
38 }
39 }

```

Если вы запустите наш пример, то сможете увидеть, что вывод на печать в секции **finally** вызывается в обоих случаях.

## Множество исключений

Как вы наверно уже догадываетесь, метод может порождать не одно исключение, а много. Тем более, если у вас внутри блока **try ... catch** вызывается несколько методов, каждый из которых порождает свой вид исключений. Соответственно и «ловить» приходится несколько исключений.

```

1 try {
2     ....
3 } catch (Exception1 ex) {
4     // Обработка исключения класса Exception1
5 } catch (Exception2 ex) {
6     // Обработка исключения класса Exception2
7 } catch (Exception3 ex) {
8     // Обработка исключения класса Exception3
9 }

```

Рассмотрим несложный пример — класс **Generator** в методе **helloMessage** порождает два исключения, а класс **Starter** будет ловить эти два исключения.

```

1 package edu.javacourse.exception;
2
3 public class Generator
4 {
5     public String helloMessage(String name) throws FirstException, SecondException
6     {
7         if ("FIRST".equals(name)) {
8             throw new FirstException("FirstException occurred");
9         }
10        if ("SECOND".equals(name)) {
11            throw new SecondException("SecondException occurred");

```

```

12     }
13     return "Hello, " + name;
14 }
15 }

```

Как видим в описании метода мы через запятую перечисляемся все исключения, которые порождаются этим методом. В примере ниже можно видеть нашу «многоярусную» конструкцию для обработки разных исключений в разных ветках.

```

1 package edu.javacourse.exception;
2
3 public class Starter
4 {
5     public static void main(String[] args)
6     {
7         Generator generator = new Generator();
8
9         try {
10             String answer = generator.sendMessage("FIRST");
11             //String answer = generator.sendMessage("SECOND");
12             //String answer = generator.sendMessage("OTHER");
13             System.out.println("Answer 1:" + answer);
14         } catch (FirstException ex) {
15             System.out.println("Error message:" + ex.getMessage());
16         } catch (SecondException ex) {
17             System.out.println("Error message:" + ex.getMessage());
18         }
19     }
20 }

```

Ниже код для двух классов исключений.

```

1 package edu.javacourse.exception;
2
3 public class FirstException extends Exception
4 {
5     public FirstException(String message)
6     {
7         super(message);
8     }
9 }

```



```

1 package edu.javacourse.exception;
2
3 public class SecondException extends Exception
4 {
5     public SecondException(String message) {
6         super(message);
7     }
8 }

```

Как видим ничего особенно сложного нет. В Java 1.7 появилась более компактная конструкция, в которой классы исключений перечисляются в одном catch через знак «|». Вот так:

```

catch (FirstException | SecondException ex) {
    System.out.println(«Error message:» + ex.getMessage());
}

```

```

1 package edu.javacourse.exception;
2
3 public class Starter
4 {
5     public static void main(String[] args)
6     {
7         Generator generator = new Generator();
8
9         try {
10             String answer = generator.helloMessage("FIRST");
11             //String answer = generator.helloMessage("SECOND");
12             //String answer = generator.helloMessage("OTHER");
13             System.out.println("Answer 1:" + answer);
14         } catch (FirstException | SecondException ex) {
15             System.out.println("Error message:" + ex.getMessage());
16         }
17     }
18 }

```

Но нередко все исключения можно обработать в одной ветке catch. В этом случае можно использовать полиморфизм и наследование исключений. Как и любой класс, исключение тоже наследуется. При построении секции catch это можно использовать. Принцип следующий — поиск подходящего исключения начинается с первого catch. Вы понимаете, что все наследники класса **Exception** подходят под этот класс. Значит если у нас самый первый catch будет ловить класс **Exception**, то туда будут попадать практически все исключения.

Давайте рассмотрим вот такой пример обработки:

```

1 package edu.javacourse.exception;
2
3 public class Starter
4 {
5     public static void main(String[] args)
6     {
7         Generator generator = new Generator();
8
9         try {
10             String answer = generator.sendMessage("FIRST");
11             //String answer = generator.sendMessage("SECOND");
12             //String answer = generator.sendMessage("OTHER");
13             System.out.println("Answer 1:" + answer);
14         } catch (Exception ex) {
15             System.out.println("Error message:" + ex.getMessage());
16         } catch (FirstException ex) {
17             System.out.println("Error message:" + ex.getMessage());
18         } catch (SecondException ex) {
19             System.out.println("Error message:" + ex.getMessage());
20         }
21     }
22 }

```

Этот пример не будет компилироваться, т.к. второй и третий catch не будет достигнут никогда — мы все время попадаем в первую ветку catch. Но если мы переместим обработку **Exception** в конец, то этот код будет корректным, т.к. сначала мы проверяем более точные классы исключений. Код ниже на первый взгляд, не сильно отличается. но его можно скомпилировать.

```

1 package edu.javacourse.exception;
2
3 public class Starter
4 {
5     public static void main(String[] args)
6     {
7         Generator generator = new Generator();
8
9         try {
10             String answer = generator.sendMessage("FIRST");
11             //String answer = generator.sendMessage("SECOND");
12             //String answer = generator.sendMessage("OTHER");
13             System.out.println("Answer 1:" + answer);
14         } catch (FirstException ex) {
15             System.out.println("Error message:" + ex.getMessage());

```

```
16         } catch (SecondException ex) {  
17             System.out.println("Error message:" + ex.getMessage());  
18         } catch (Exception ex) {  
19             System.out.println("Error message:" + ex.getMessage());  
20         }  
21     }  
22 }
```

Этим приемом нередко пользуется — учитесь строить, читать и распознавать такие конструкции.

## Класс RuntimeException

В Java не все исключения необходимо обрабатывать. Существует целый класс, от которого можно порождать исключения и которые не требуют обязательной обработки. Это класс **RuntimeException**.

Это достаточно обширная иерархия, в которой достаточно много распространенных классов — `NullPointerException`, `ClassCastException`.

## Пожелания

Самое главное пожелание — не оставляйте блоки `catch` без информации. Очень неплохим подспорьем будет вызов метода **`printStackTrace`**. Очень важно понимать, что случилось, почему. И если блок `catch` не будет информативным, то выяснение причин будет крайне сложным занятием.

Ужасным кодом будет нечто вроде этого:

```
1 try {  
2     ...  
3 } catch (Exception ex) {  
4     System.out.println("Произошло исключение");  
5 }
```

Что здесь можно понять в случае возникновения исключения ? НИЧЕГО ? Уважайте свой труд — выводите подробную информацию.

## Что дальше ?

Рассмотрев исключения, мы завершили знакомство с конструкциями языка Java. Я говорю именно о конструкциях — у нас еще впереди долгий путь изучения. Но если выражаться образно, то алфавит освоили. Мы еще будем встречать некоторые конструкции и слова, но подавляющая часть уже пройдена. Впереди нас ждет изучение уже готовых классов, которые предназначены для решения различных задач.

Все конструкции служат главному — позволить вам создавать описания классов, создавать объекты и управлять последовательностью вызовов для решения своей задачи. Можно по-разному оценивать удобство, но раз это так сделано разработчиками Java — надо уметь этим пользоваться.

И еще раз выскажу пожелание — учитесь читать код, учитесь понимать программу, просто просматривая что она делает, шаг за шагом.

И теперь нас ждет следующая статья: [Решения на основе классов](#).

## 16 comments to *Исключения*



• Январь 29, 2015 at 02:13

[SeveneduS](#) says:

Для начала хочу поблагодарить за статью.  
Пожалуйста переправьте:  
приер такого -> приМер такого описания.

[Reply](#)



• Сентябрь 19, 2015 at 15:53

*Grif* says:

```
«finally {  
// Этот блок будет вызываться всегда, независиОМ от результата  
System.out.println(«Этот блок вызывается всегда»);  
}»
```

Прошу прощения за досаду, маленькая опечатка, «О» и «М» надо местами поменять.

[Reply](#)



• Декабрь 10, 2015 at 18:57

*Я* says:

Да что вы все заладили с поправками!  
Будет у автора возможность, прогонит он статьи через спеллчекер или поставит ogphtus и, если надо, стилистику текста поправит!

Комментарии к статье не для того предназначены!

Давайте еще спеллчекингом комментариев друг-друга займемся?! Это же интереснее, чем Java!

[Reply](#)



o

Декабрь 11, 2015 at 17:03

*admin* says:

На самом деле это удобно и я очень благодарен таким замечаниям — исправил достаточно много ошибок. Заниматься поиском и исправлением все никак не получается. Я со временем такие комментарии удаляю.

[Reply](#)



•

Март 6, 2017 at 17:53

*Aiven* says:

Опять же поправка — «Я уже дважды использовать термин «бросать»»

Спасибо за Ваш сайт, здесь есть всё что необходимо, кроме заданий для закрепления материала.

[Reply](#)



o

Март 6, 2017 at 18:07

*admin* says:

Спасибо, исправил. По поводу заданий — это достаточно важный элемент, но просто задания давать не получится — их же еще надо проверить. Для этого надо что-то придумать — просто опубликовать ответы нельзя — будут подсматривать и эффект будет нулевой. Я в процессе поиска решения.

[Reply](#)



• Май 1, 2017 at 12:22

*Serga* says:

Здравствуй! два последних кода совпадают, скорее всего вместо одного из них должен быть генератор для исключений.

[Reply](#)



○

Май 2, 2017 at 12:45

*admin* says:

Не очень понял данный комментарий. Что значит «два последних кода совпадают» ?

Если имеются в виду два куска в самом конце, которые включают обработку нескольких классов исключений, то они РАЗНЫЕ. Надо внимательно посмотреть.

[Reply](#)



■

Май 4, 2017 at 16:20

*Serga* says:

извиняюсь, не заметил. отличные курсы 😊

[Reply](#)



• Август 10, 2017 at 12:06

*Дмитрий* says:

Извиняюсь, возник вопрос касательно переопределения методов и необходимости указывать в заголовке метода перечень потенциально выбрасываемых исключений. В общем, уже не один раз убеждался на практике, что сигнатурой метода правильно считать сочетание имени

метода с количеством и типами его формальных параметров. Сочетание сигнатуры метода с типом возвращаемого значения принято считать контрактом. В контракт также входят все типы исключений потенциально возбуждаемых методом.

Собственно, должны ли мы при переопределении в обязательном порядке указывать все исключения родительского метода? Тип возвращаемого значения должен соответствовать полностью, а вот с исключениями что-то не очень понятно. Если исключения вместе с типом возвращаемого значения и сигнатурой метода составляют контракт, тогда почему же при переопределении учитывается только сигнатура и тип возвращаемого значения? Ориентируюсь на [.](#)

[Reply.](#)



o

Август 10, 2017 at 16:29

*admin* says:

Переопределяя метод, можно либо использовать те же исключения, либо можно сузить класс исключения — например заменить `IOException` на `FileNotFoundException`.

Либо можно совсем убрать исключения у наследника. Вот и комбинируйте 😊

[Reply.](#)



•

Февраль 2, 2018 at 10:09

*Zharaskan* says:

Добрый день! В чем смысл кода // переопределяем конструктор

```
public SimpleException(String message)
{
    this(0, message);
}
```

[Reply.](#)



o

Февраль 2, 2018 at 15:25

*admin* says:

Просто для наглядности, что можно такое сделать — два конструктора. В общем особой надобности в таком конструкторе может и не быть.

[Reply](#)



Февраль 2, 2018 at 12:49

*Zharaskan* says:

При создании объекта SimpleException se = new SimpleException(10, «Message is null»);вызывается первый конструктор public

SimpleException(String message)

```
{  
this(0, message);
```

} через this(0, message)передаются параметры (10, «Message is null»)второму конструктору public SimpleException(int errorCode, String message)

```
{  
// Вызываем конструктор предка
```

```
super(message);
```

```
// Добавляем инициализацию своего поля
```

```
this.errorCode = errorCode;
```

```
}, а ноль в параметрах this(0, message)необходима для начальной инициализации параметра int errorCode. Правильно ли? Спасибо за внимание!
```

[Reply](#)



Февраль 3, 2018 at 19:28

*admin* says:

Когда вызывается конструктор с двумя параметрами, то в нем вызывается конструктор предка — Exception. После этого присваивается код.

Когда вызывается конструктор с одним параметром. то внутри него вызывается конструктор с двумя параметрами и код ставится в 0.

[Reply](#)



Февраль 5, 2018 at 06:40



*Zharaskan* says:

Спасибо!Я тоже так думал. Однако код: / переопределяем конструктор

```
public SimpleException(String message)
```

```
{
```

```
this(0, message);
```

```
} просто ввел меня в ступор!
```

[Reply](#)

## Leave a reply


Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя \*

E-mail \*

Сайт

× 3 = 27 

Copyright © 2018 [Java Course](#)

Designed by [Blog templates](#), thanks to: [Free WordPress themes for photographers](#), [LizardThemes.com](#) and [Free WordPress real estate themes](#)

