Throwable

Описание
Иерархия исключений
Проверяемые и непроверяемые исключения
Ошибки (Error)
Исключения времени выполнения (RuntimeException)
Прочие исключения (Exception)
Использование проверяемых и непроверяемых исключений
Создание исключения
Получение информации об исключении

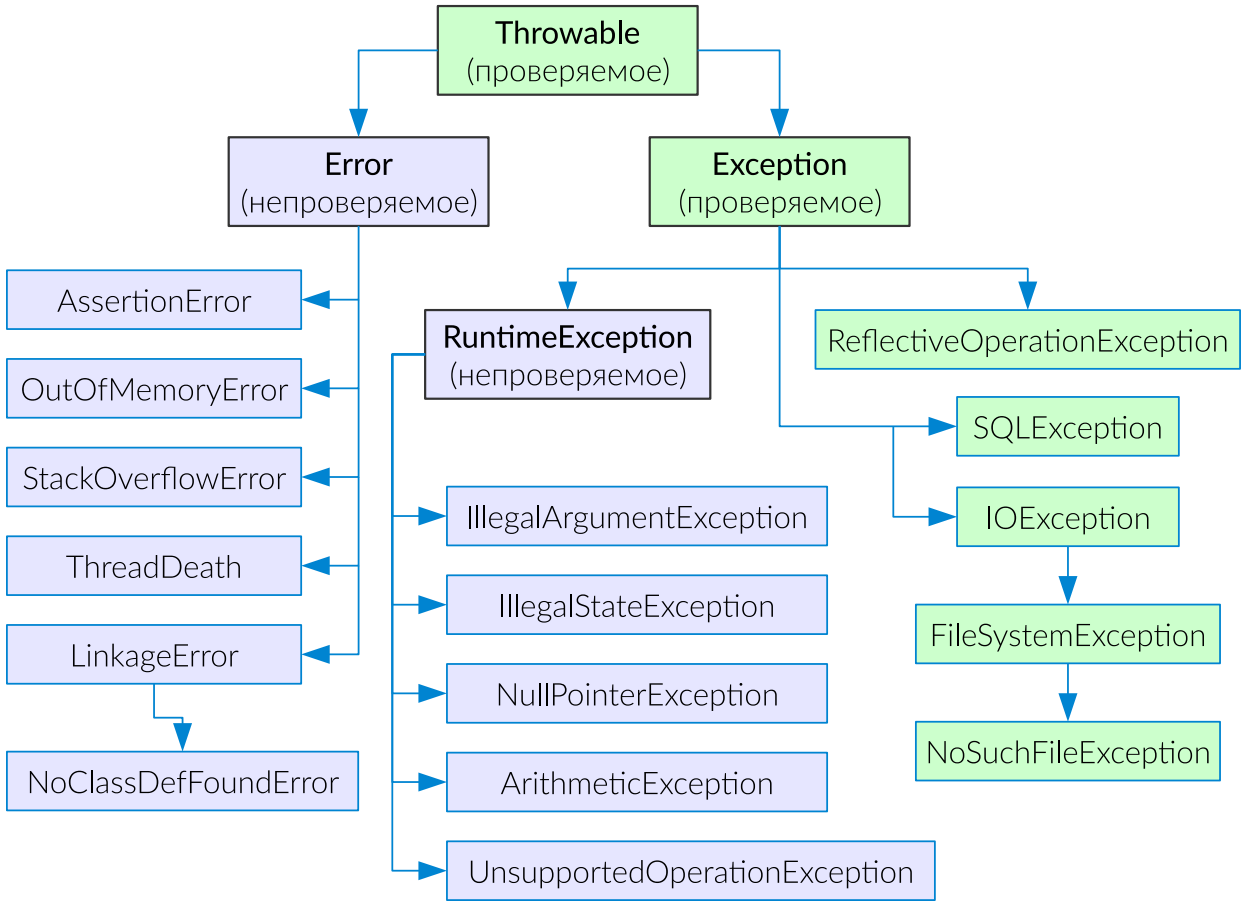
Класс: `java.lang.Throwable`

Описание

`Throwable` — базовый класс всех исключений Java. В инструкциях `throw` и `catch` можно использовать только объекты класса `Throwable` и его подклассов.

Объект-исключение содержит в себе текстовое сообщение, говорящее о причине ошибки, трассировку стека вызовов на момент создания, а также, возможно, *причину* (cause) — исключение более низкого уровня, завёрнутое в данное исключение.

Иерархия исключений



Все исключения в Java делятся на две большие и неравные группы. Меньшая группа наследует от класса `Error` и обозначает серьёзные низкоуровневые ошибки, после которых продолжение выполнения программы обычно бессмысленно. Большая группа наследует от класса `Exception` и отвечает за обычные нештатные ситуации, которые могут возникнуть в ходе выполнения программы.

Теоретически язык Java не запрещает определить свой подкласс класса `Throwable`, не производный ни от `Error`, ни от `Exception`. Компилятор будет рассматривать такую неведому зверушку как обычное проверяемое исключение (см. ниже), но на практике так никто не поступает, да и необходимости в этом нет.

Проверяемые и непроверяемые исключения

Классы `Error` и `RuntimeException` занимают особое положение в иерархии исключений. Эти классы, а также все производные от них, являются *непроверяемыми исключениями* (unchecked exceptions). Все остальные исключения являются *проверяемыми* (checked exceptions). На диаграмме классов выше непроверяемые исключения выделены голубым, а проверяемые — зелёным.

Метод, внутри которого может быть выброшено проверяемое исключение, должен либо обработать его в блоке `catch`, либо передать выше и явно сказать об этом в объявлении с помощью ключевого слова `throws` (и тогда разбираться с ним будут уже где-то выше). На непроверяемые исключения компилятор не налагает таких требований.

```
void throwsIOException() throws IOException {
    if (MoonPhase.getCurrent() == MoonPhase.FULL) {
        // непроверяемое исключение; можно обработать, но компилятор не заставляет
        throw new IllegalStateException();
    } else {
        // проверяемое исключение
        throw new IOException();
    }
}

void handlesIOException() {
    try {
        throwsIOException();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Важно! В стандартной библиотеке есть одно нарушение этого правила. Статический метод `Class.newInstance` передаёт во внешний код любые исключения, в том числе проверяемые, что позволяет обойти проверки времени компиляции:

```
public class Thrower {
    private Thrower () throws IOException {
        throw new IOException();
    }

    public static void main(String[] args) {
        try {
            Thrower t = Thrower.class.newInstance();
        } catch (IllegalAccessException | InstantiationException e) {
            // Ну хотя бы эти нужно обрабатывать, но уже поздно
        }
    }
}
```

Здесь `main` упадёт по `IOException`, хотя обычно компилятор заставил бы нас объявить его как `throws IOException`. Мораль в том, что при использовании рефлексии всегда нужно быть осторожными, потому что она предоставляет внеязыковой механизм манипуляции классами и методами.

Ошибки (Error)

Исключения, производные от класса `Error`, являются непроверяемыми и обладают объединяющими свойствами:

1. Они сигнализируют о серьёзных ошибках низкого уровня, после которых восстановление обычно невозможно.
2. Многие из них (в частности, `OutOfMemoryError`, `LinkageError` и `ThreadDeath`) могут возникнуть практически в любом месте программы, поэтому настраиваться на них заранее обычно бессмысленно. Кроме того, место, где выбрасывается исключение, обычно не связано с местом логического возникновения нижележащей проблемы.

Ловить и обрабатывать исключения, производные от `Error`, обычно не следует. Они выбрасываются затем, чтобы как можно быстрее дать потоку завершиться аварийно, выполнив при этом все блоки `finally` при раскрутке стека.

Вот некоторые наиболее распространённые исключения типа `Error`:

`StackOverflowError`
: Переполнение стека вызовов. Обычно это указывает на бесконечную рекурсию.

`OutOfMemoryError`
: Переполнение кучи, причём сборщик мусора уже попытался её почистить и беспомощно развёл руками. Обычно это указывает на утечки памяти, вызванные хранением ссылок на множество ненужных объектов в корневых переменных.

`LinkageError`
: Базовый класс для различных ошибок при загрузке классов, необходимых для работы выполняемого кода. Сюда относятся в том числе `NoClassDefFoundError` и `ExceptionInInitializerError`.

`NoClassDefFoundError`
: JVM не может найти класс, к которому пытается обратиться код. Это может случиться, если код был скомпилирован с зависимостью от какого-то класса или библиотеки, но запущен при отсутствии этого класса/библиотеки в classpath.

`ExceptionInInitializerError`
: При инициализации статического поля класса или внутри блока `static` выбросилось исключение, к которому можно доступиться через метод `getCause`.

`ThreadDeath`
: В многопоточной программе чужой поток нагло и бесцеремонно завершил работу нашего потока методом `Thread.stop`. Если поток завершается по этому исключению, по умолчанию сообщение об ошибке подавляется и не пишется в консоль.

`AssertionError`
: Нарушено базовое условие, которое в корректно работающей программе должно выполняться всегда. Исключения этого класса бросает инструкция `assert` при запуске JVM с параметром `-ea` (enable assertions), а также библиотека JUnit при провале тестов.

Как правило, не стоит выбрасывать исключения типа `Error` инструкцией `throw`, кроме исключения `AssertionError`. Его принято выбрасывать как "невозможное" исключение, чтобы "заткнуть" выбросом исключения пути выполнения, которые заведомо никогда не будут выполнены, но компилятор об этом не знает. Часто это приходится делать при использовании перечислимых типов в инструкции `switch`:

```
public enum Stoplight { RED, YELLOW, GREEN }
public enum CarState { STOPPED, STOPPING, MOVING }

public class Car {
    public CarState approachIntersection(Intersection intersection) {
        switch (intersection.getStoplight()) {
        case RED:
            stop();
            return CarState.STOPPING;
        case YELLOW:
            if (stopIfPossible()) {
                return CarState.STOPPING;
            } else {
                return CarState.MOVING;
            }
        case GREEN:
            keepMoving();
            return CarState.MOVING;
        default:
            // не может случиться
            throw new AssertionError();
        }
    }

    public CarState handleStoplightSwitch(Stoplight stoplight) {
        switch (stoplight) {
        case RED:
            return CarState.STOPPED;
        case YELLOW:
            prepareToMove();
            return CarState.STOPPED;
        case GREEN:
            startMoving();
            return CarState.MOVING;
        default:
            // не может случиться
            throw new AssertionError();
        }
    }
}
```

В нашем случае у светофора всего три возможных состояния, но компилятор тем не менее требует, чтобы блок `switch` обработал невозможную ситуацию `default`. С помощью `throw new AssertionError` мы затыкаем компилятор и говорим сопровождающему код, что эта строка кода заведомо не выполнится.

Ещё один частый use case для `AssertionError` — блок `catch` для исключения, невозможного по определению. Например, стандартные классы `URLEncoder` и `URLDecoder` имеют методы, принимающие два параметра: перекодируемую строку и строку с именем кодировки.

```
String url = "https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA";
String decodedUrl = URLDecoder.decode(url, "UTF-8");
```

Этот код просто так не скомпилируется, потому что метод `URLDecoder.decode` объявлен как `throws UnsupportedEncodingException`. Однако кодировка UTF-8 гарантированно

поддерживается в любой реализации JVM, поэтому на самом деле это исключение никогда не выбросится (но компилятор, увы, об этом не знает). Нам придётся обернуть этот код в идиому "невозможное исключение" (impossible exception):

```
try {
    String url = "https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA";
    String decodedUrl = URLDecoder.decode(url, "UTF-8");
    // https://ru.wikipedia.org/wiki/Стек
} catch (UnsupportedEncodingException e) {
    // не может случиться
    throw new AssertionError(e);
}
```

Важно! Большинство методов, работающих с кодировками, имеют версии, принимающие вместо строки с именем кодировки объект класса `Charset` и не бросающие `UnsupportedEncodingException`. Им можно передавать константы из класса `StandardCharsets`:

```
List<String> fileContents = Files.readAllLines(
      Paths.get("отчёт.txt"), StandardCharsets.UTF_8);
fileContents.forEach(System.out::println);
```

К сожалению, некоторые старые классы, в том числе `URLEncoder` и `URLDecoder`, так и не были обновлены для поддержки параметров типа `Charset`.

Исключения времени выполнения (RuntimeException)

Строго говоря, *все* исключения возникают во время выполнения, но класс `RuntimeException` занимает особое положение. Во-первых, `RuntimeException` и все производные от него исключения являются непроверяемыми. Во-вторых, многие из этих исключений сигнализируют об ошибках в программном коде и в корректно написанной программе не должны возникать вообще. Такие исключения полезно выбрасывать в собственных `public`-методах в качестве *предусловий* (preconditions), краткий смысл которых состоит в том, чтобы сказать вызывающему коду: "Ты виноват, такой ситуации вообще не должно быть".

Вот некоторые самые важные из них:

`NullPointerException`
: Программа попыталась обратиться к полю или методу ссылки `null`, либо значение `null` было передано в метод, не допускающий `null` в качестве параметра.

`IllegalArgumentException`
: Метод был вызван с недопустимым значением параметра. Например, один из конструкторов стандартного класса `Color`, принимающий три целых числа RGB, выбрасывает это исключение, если переданные числа не лежат в диапазоне 0–255.

`IllegalStateException`
: Объект находится в состоянии, в котором выполнение операции невозможно. Например, реализация стека может выбросить это исключение при попытке извлечь элемент из пустого стека.

`IndexOutOfBoundsException`
: Попытка обратиться к упорядоченной последовательности элементов (массиву, строке или списку) по индексу, лежащему вне допустимых значений. Часто происходит при ошибке на единицу, когда в качестве индекса используется `length` или `size`.

`NoSuchElementException`
: Выбрасывается итератором при попытке прочитать элемент за концом последовательности (когда `hasNext() == false`). В корректной реализации итератора это исключение возможно только при ручной работе с итератором и невозможно при использовании цикла for-each.

`UnsupportedOperationException`
: Вызываемая операция в принципе запрещена для этого объекта. Например, неизменяемые коллекции выбрасывают это исключение при попытке изменить их методами `set`, `add` или `remove`.

`ArithmeticException`
: Вызвана недопустимая арифметическая операция, например, деление на ноль. Операции над числами с плавающей точкой не выбрасывают это исключение, вместо этого при недопустимой операции они возвращают значение `NaN` (а при делении ненулевого числа на ноль — бесконечность).

`ClassCastException`
: Попытка привести объект к несовместимому типу, например, `(String) new Object()`.

Все эти исключения объединяет то, что в корректно написанной программе они не должны выбрасываться. Если одно из этих исключений ловится в программе, это знак неверной логики кода. Часто отлов такого исключения можно заменить проверкой; например, вместо отлова `ClassCastException` использовать оператор `instanceof`, вместо `IndexOutOfBoundsException` — проверку диапазона (`index >= 0 && index < size()`), а вместо `NoSuchElementException` — проверку `Iterator.hasNext()`.

Прочие исключения (Exception)

Исключения, производные от `Exception`, но не `RuntimeException`, являются проверяемыми. Их очень много, и сторонние библиотеки часто добавляют свои собственные классы исключений. Вот лишь несколько примеров:

`ReflectiveOperationException`
: Базовый класс для исключений, возникающих при работе с механизмом рефлексии. Сюда относятся `IllegalAccessException` (попытка доступиться извне к членам класса с доступом private, protected или package-private), `NoSuchMethodException` (метод не найден) и `InvocationTargetException` (оборачивает исключение, выброшенное вызываемым через рефлексию методом). Отдельно стоит выделить...

`ClassNotFoundException`
: Не путать с `NoClassDefFoundError`. Выбрасывается методом `Class.forName`, если класс с таким именем не найден. Это исключение, в отличие от `NoClassDefFoundError`, предназначено для того, чтобы его перехватывали и обрабатывали.

`CloneNotSupportedException`
: По умолчанию выбрасывается `protected`-методом `Object.clone`, если объект не реализует интерфейс `Cloneable`. По каким-то непонятным причинам конченые укурки, проектировавшие Java 1.0, сделали это исключение проверяемым, и при реализации `Cloneable`-классов это исключение приходится подавлять.

`SQLException`
: Базовый класс для ошибок при работе с базами данных через JDBC API.

`IOException`
: Базовый класс для исключений при операциях ввода-вывода, к которым относятся операции, в том числе, с файловой системой и сетью. У этого класса очень много

подклассов, обозначающих конкретные ситуации; например, `ConnectException` бросается при неудачной попытке соединения с сервером.

`FileSystemException`

Подкласс `IOException`, являющийся базовым классом для исключений при операциях с файловой системой.

`AccessDeniedException`

Попытка обратиться к файлу, на доступ к которому у пользователя нет прав — например, попытка открыть защищённый системный файл для записи.

`NoSuchFileException`

Попытка обратиться к несуществующему файлу.

> **Важно!** Класс `NoSuchFileException` появился в Java 7 и является частью нового файлового API (классы `Path` и `Files`). В старых API, спроектированных для Java 6 и ниже, можно встретить более старое исключение `FileNotFoundException`, присутствовавшее ещё в Java 1.0. К сожалению, это исключение не позволяет различить ситуации "файл не найден" и "доступ запрещён" и выбрасывается старыми API в обоих этих случаях.

Использование проверяемых и непроверяемых исключений

Тема проверяемых и непроверяемых исключений, а также правил их использования, сломала немало копий (и кода). Полного консенсуса по этому вопросу нет, но можно пользоваться эмпирическим правилом.

Выбрасывайте проверяемые исключения только при одновременном выполнении трёх условий:

1. Вызывающий код не может гарантированно избежать ошибочной ситуации, то есть она находится вне его полного контроля.
2. Ошибочная ситуация является "стандартной", то есть в любом случае автору вызывающего кода нужно думать о том, что произойдёт при возникновении ошибки. Например, при попытке открыть файл может оказаться, что файл недоступен.
3. Вызывающий код может осмысленно обработать ошибку или даже восстановиться после неё.

Хорошими примерами проверяемых исключений являются исключения при работе с файловой системой, сетью, базами данных, окнами, другими процессами в системе и т.д. Все эти сущности объединяет то, что это внешние ресурсы, корректную работу которых вызывающий код не может полностью гарантировать. Плохой пример проверяемого исключения — исключение `CloneNotSupportedException`, которое может возникнуть только при логической ошибке в использовании метода `clone`. Если бы класс `CloneNotSupportedException` был написан в наши дни, скорее всего, это исключение было бы непроверяемым.

Если хотя бы одно из этих трёх условий не выполняется, используйте непроверяемые исключения. Вот примеры ошибок, для которых подходят непроверяемые исключения:

- Низкоуровневые ошибки самой JVM и загруженных классов (`Error`).
- Ошибки в логике программы, необнаружимые на этапе компиляции (`RuntimeException`).
- Ошибки, при которых продолжение выполнения кода не имеет смысла. Например, в серверных приложениях это могут быть ошибки конфигурации сервера, при которых приложение вообще не может запуститься, или ошибки доступа к основной базе данных, с которой работает серверное приложение (здесь само приложение всё равно не сможет восстановиться). Как правило, такие исключения тоже наследуют от `RuntimeException`.

Создание исключения

Объекты исключений, как правило, создаются прямо в инструкции `throw`:

`throw new SomeException(параметры);`

По соглашению у большинства классов исключений есть четыре конструктора, позволяющие задать исключению сообщение и/или причину — более низкоуровневое исключение.

`SomeException()`

`SomeException(String message)`

`SomeException(Throwable cause)`

`SomeException(String message, Throwable cause)`

Это позволяет, с одной стороны, соблюдать инкапсуляцию, не проталкивая низкоуровневые исключения вверх, а с другой — сохранять информацию о низкоуровневых исключениях для отладочных целей.

```
try {
    readConfigFile("server-config.xml");
} catch (XMLStreamException e) {
    throw new ServerStartupException("Invalid server configuration format", e);
} catch (IOException e) {
    throw new ServerStartupException("Cannot access server configuration", e);
}
```

У некоторых очень старых классов исключений, написанных до выхода Java 1.4, может не быть конструкторов с параметром `cause`. Для них можно установить причину после создания объекта с помощью метода `initCause`:

```
throw new OldException("Message").initCause(e);
```

Получение информации об исключении

Все исключения поддерживают три базовых операции:

`String getMessage()`
: Возвращает короткое, в одно-два предложения, сообщение об ошибке, переданное в объект исключения при его создании.

`Throwable getCause()`
: Возвращает исключение, послужившее причиной данного исключения, или `null`, если таковое не было задано. У причины, в свою очередь, тоже может быть причина; с помощью последовательных вызовов `getCause` можно восстановить причинную цепь исключений (causal chain).

`void printStackTrace()`
`void printStackTrace(PrintStream s)`
`void printStackTrace(PrintWriter s)`
: Выводит трассировку стека (stack trace) в указанный байтовый или символьный поток (по умолчанию — в `System.err`). Варианты с `PrintStream` и `PrintWriter` бывают полезны, чтобы записать трассировку стека в файл или сетевой поток, либо получить её в виде строки с помощью `StringWriter`.

Если у исключения есть причина, методы `printStackTrace` вслед за трассировкой стека самого исключения печатают трассировку стека его причины, затем — причины причины и так

далее.

По умолчанию для всех исключений, кроме `ThreadDeath`, при завершении потока по необработанному исключению трассировка стека записывается в `System.err` с помощью `printStackTrace`. Это относится и к главному потоку при аварийном завершении метода `main`.

Как пример, вот такая неправильная программа

```
public class ExceptionChain {
    private static class BadInitializer {
        private static final char CONSTANT = "Hello".charAt(5);
    }

    public static void main(String[] args) {
        System.out.println(BadInitializer.CONSTANT);
    }
}
```

при выполнении выдаст:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
    at ExceptionChain.main(ExceptionChain.java:11)
Caused by: java.lang.StringIndexOutOfBoundsException: String index out of range: 5
    at java.lang.String.charAt(String.java:658)
    at ExceptionChain$BadInitializer.<clinit>(ExceptionChain.java:7)
    ... 1 more
```