Введение в программирование на Java

Лекция 12. Обработка ошибок. Исключения.

Виталий Олегович Афанасьев

21 апреля 2025

Обработка ошибок

В идеальном мире программы всегда работают без ошибок, пользователи вводят только корректные данные, а проблем с аппаратурой никогда не возникает.

В реальности же всегда нужно задумываться о многих вещах:

- Какие ошибки могут возникнуть в программе?
- Как необходимо обрабатывать эти ошибки?
- Как сообщить об ошибке пользователю (и разработчикам)?
- Нужно ли аварийно завершить программу, или же попытаться восстановиться после ошибки и продолжить работу?
- Нужно ли и как сохранить промежуточный результат работы?

Исправимые и неисправимые ошибки

Все ошибки можно разделить на **исправимые (recoverable)** и **неисправимые (unrecoverable)**.

В случая неисправимой ошибки программа аварийно завершается.

В случае исправимой — пытается продолжить корректное выполнение.

Исправимые и неисправимые ошибки

Все ошибки можно разделить на **исправимые (recoverable)** и **неисправимые (unrecoverable)**.

В случая неисправимой ошибки программа аварийно завершается.

В случае исправимой — пытается продолжить корректное выполнение.

Исправимость ошибки сильно зависит от контекста:

- Что можно считать неисправимой ошибкой в мобильной игре "три в ряд"?
- А что можно считать неисправимой ошибкой в бортовом ПО самолёта?

Способы обработки исправимых ошибок

- Возврат кода ошибки (или объекта-ошибки)
  - C, Go
- Исключения
  - Java, Python, C++, C#
- Возврат специальных объектов-обёрток над результатом
  - Java, Rust, Haskell, Scala

Исключения

Исключение (exception) — событие, происходящее во время выполнения программы и нарушающее нормальный ход выполнения.

Исключительная ситуация отличается от обычной ошибки тем, что в текущем контексте (например, в текущем методе) нет возможности обработать эту ошибку и восстановить нормальную работу.

Исключения

Исключение (exception) — событие, происходящее во время выполнения программы и нарушающее нормальный ход выполнения.

Исключительная ситуация отличается от обычной ошибки тем, что в текущем контексте (например, в текущем методе) нет возможности обработать эту ошибку и восстановить нормальную работу.

Исключения сильно влияют на производительность программы, поэтому возникать они должны **только в исключительных случаях**.

Обработка исключений

Для обработки исключений используется конструкция `try-catch`.

Если при выполнении кода в `try`-блоке возникает исключение, которое обрабатывается блоком `catch` — управление переходит к нему.

```
1 public int parseIntOrGetMin(String str) {
2     try {
3         return Integer.parseInt(str);
4     } catch (NumberFormatException e) {
5         System.out.println("Ошибка: строка '" + str + "' не int");
6         return Integer.MIN_VALUE;
7     }
8 }
```

Если же тип возникшего исключения не обрабатывается, то текущий метод завершается с исключением.

Генерация исключений

Для генерации исключений используется ключевое слово `throw`.

В качестве аргумента `throw` выступает объект типа `Throwable`.

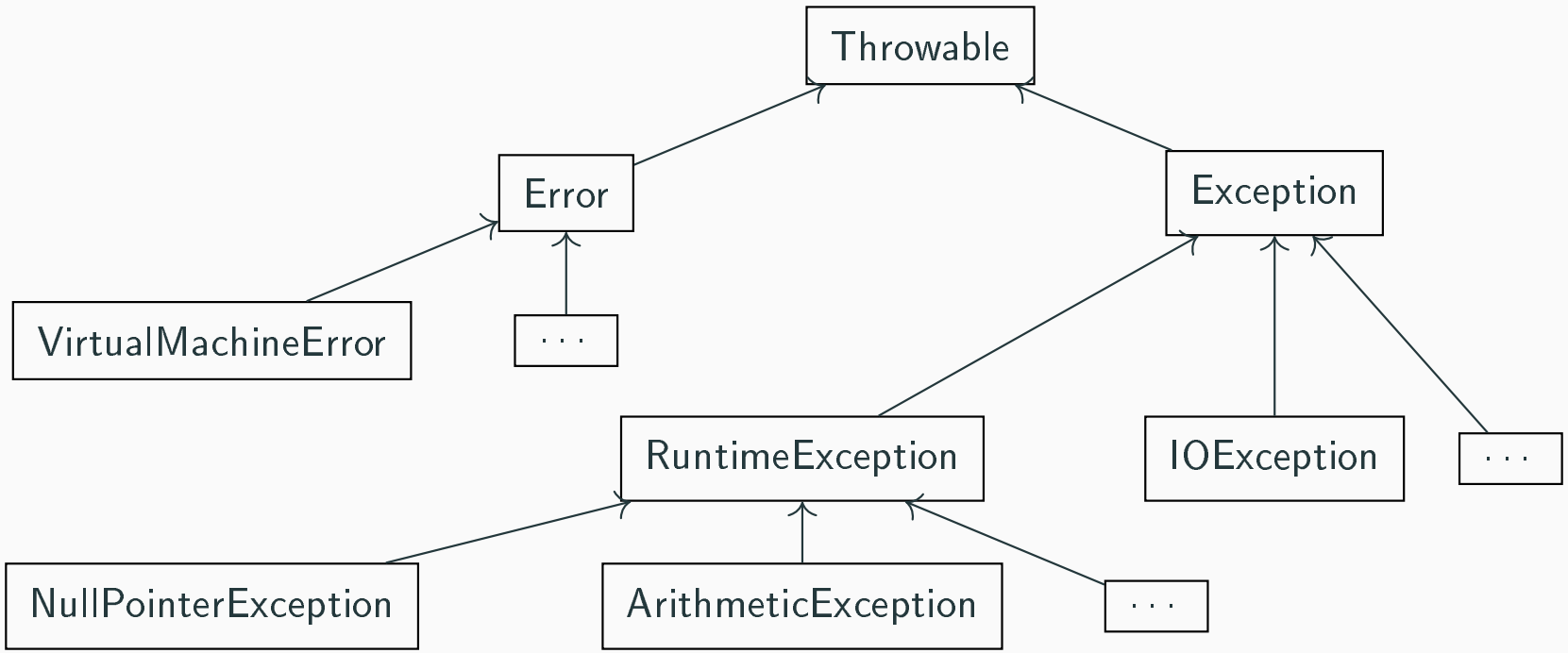
```
1 public static int lengthOfSegment(int start, int end) {
2     if (start > end) {
3         throw new IllegalArgumentException(
4             "Неправильный отрезок [" + start + ";" + end + "]"
5         );
6     }
7     return end - start;
8 }
```

Если брошенное исключение не обрабатывается, то текущий метод завершается с исключением.

Пример обработки исключений в случае нескольких методов

```
1  public static void main(String[] args) {
2      foo("Hello");
3      foo("");
4      foo(null);
5  }
6
7  private static void foo(String s) {
8      try {
9          bar(s);
10     } catch (IllegalArgumentException e) {
11         System.out.println("Error: " + e.getMessage());
12     }
13     System.out.println("foo finished successfully");
14 }
15
16 private static void bar(String s) {
17     if (s == null) throw new NullPointerException("s is null");
18     if (s.isEmpty()) throw new IllegalArgumentException("s is empty");
19     System.out.println(s);
20 }
```

Иерархия исключений



Примеры исключений

Иерархия стандартных исключений довольно большая (более 150 классов).

Из интересных:

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `StringIndexOutOfBoundsException`
- `ArithmeticException`
- `StackOverflowError`
- `OutOfMemoryError`

Обработка родительского типа исключения

Catch обрабатывает не только конкретный тип исключения, но и всех его наследников.

```
1 public int parseIntOrGetMin(String str) {
2     try {
3         return Integer.parseInt(str);
4     } catch (RuntimeException e) {
5         System.out.println("Ошибка: строка '" + str + "' не int");
6         return Integer.MIN_VALUE;
7     }
8 }
```

НО! правильнее всегда конкретизировать, какие ошибки вы можете обработать. **Не нужно слепо ловить** Throwable **всегда и везде.**

Rethrow исключений

Иногда возникают случаи, когда исходное исключение содержит очень мало информации, поэтому хотелось бы его отловить и добавить дополнительное сообщение.

Такую ситуацию называют rethrow — исключение ловится, а затем повторно бросается.

Для таких случаев у стандартных исключений есть конструктор, принимающий два аргумента: сообщение и исходное исключение.

```
1 public void doSomething(User user) {
2     try {
3         // Операции с user
4     } catch (RuntimeException e) {
5         throw new IllegalStateException("Error for user " + user, e);
6     }
7 }
```

Bad practice: пустой обработчик

Исключения не должны ”затыкаться” и ”тихо” игнорироваться!

```
1 public void doSomething() {
2     try {
3         ...
4     } catch (Exception e) {
5         // Никаких действий
6     }
7 }
```

Bad practice: генерация слишком общего типа исключения

Тип исключения несёт информацию о том, что произошло. Бросать исключения слишком общих типов является плохой практикой.

```
1 public void doSomething() {
2     if (...) {
3         throw new RuntimeException(...);
4     }
5 }
```

Методы Throwable

```
1 public class Throwable {
2     public String getMessage() { ... }
3     public Throwable getCause() { ... }
4     public void printStackTrace() { ... }
5     public void printStackTrace(PrintStream s) { ... }
6     public StackTraceElement[] getStackTrace() { ... }
7     ...
8 }
```

catch с несколькими типами (1)

При наличии нескольких catch-блоков, подходящий обработчик ищется сверху вниз.

```
1 try {
2     ...
3 } catch (IllegalArgumentException e) {
4     // IllegalArgumentException и наследники
5 } catch (RuntimeException e) {
6     // RuntimeException и наследники, но не IllegalArgumentException
7 }
```

catch с несколькими типами (2)

Иногда возникает потребность обработать несколько исключений одинаковым образом:

```
try {
    ...
} catch (IllegalArgumentException e) {
    System.out.println("Error: " + e.getMessage());
} catch (IllegalStateException e) {
    System.out.println("Error: " + e.getMessage());
}
```

Такие catch-блоки можно объединить в один:

```
try {
    ...
} catch (IllegalArgumentException | IllegalStateException e) {
    System.out.println("Error: " + e.getMessage());
}
```

Проверяемые и непроверяемые исключения

В Java существует разделение исключений на **проверяемые (checked)** и **непроверяемые (unchecked)**.

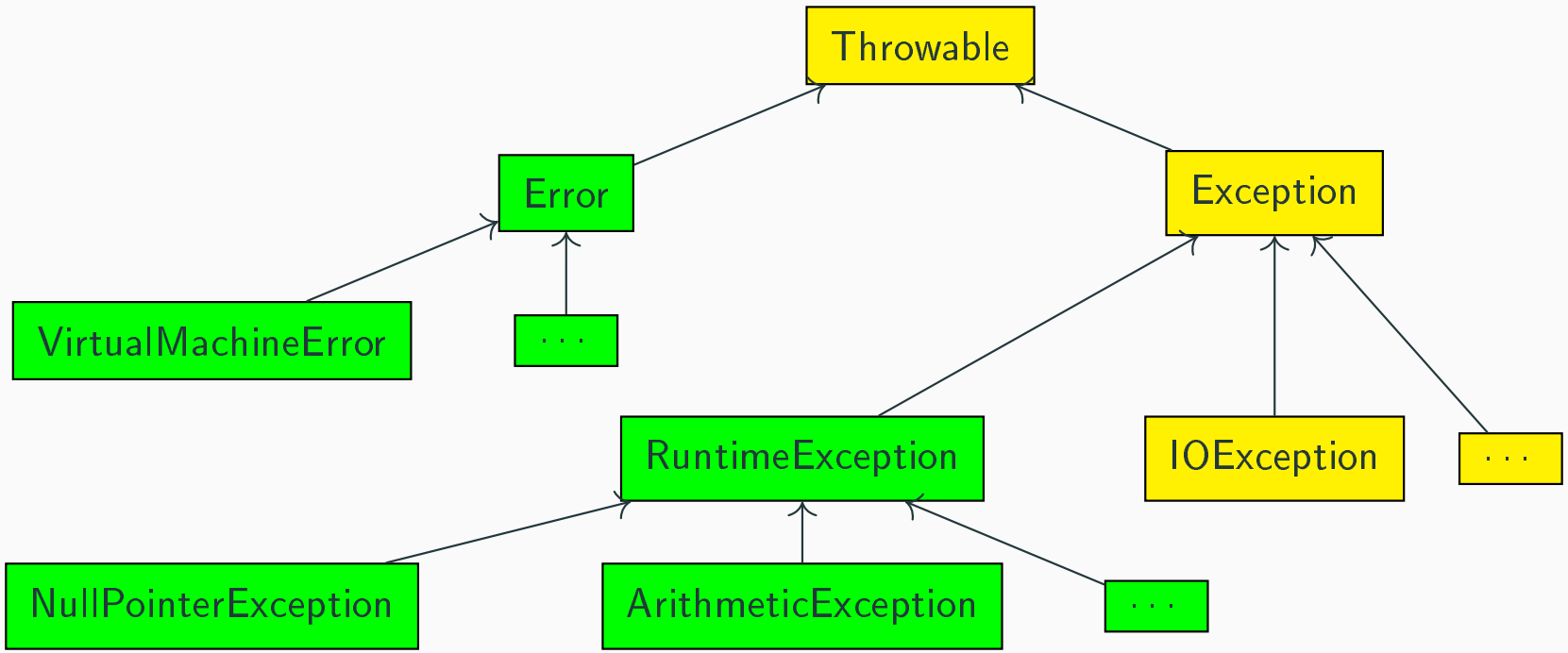
Unchecked исключения можно бросать и ловить без каких-либо хитростей.

Но за checked исключениями внимательно следит компилятор. Если в каком-то методе бросается checked исключение, то:

- Либо оно должно быть обработано;
- Либо этот метод должен объявлять, что он бросает исключение данного типа.

Если ни одно из условий не выполнено — код не будет компилироваться.

Иерархия исключений: checked и unchecked



Проверяемые и непроверяемые исключения: пример (1)

Метод объявляет, что он бросает checked исключение при помощи `throws` в объявлении метода.

```
1 public String readFromFile(String fileName) throws IOException {
2     if (/*файл не существует*/) {
3         throw new IOException(fileName + " doesn't exist");
4     }
5     ...
6 }
```

Проверяемые и непроверяемые исключения: пример (2)

При вызове данного метода можно либо обработать исключение, либо написать `throws`.

```
1  public String handle(String fileName) {
2      try {
3          return readFromFile(fileName);
4      } catch (IOException e) {
5          throw new IllegalStateException("Error for " + fileName, e);
6      }
7  }
8
9  public String dontHandle(String fileName) throws IOException {
10     return readFromFile(fileName);
11 }
```

Проверяемые и непроверяемые исключения: наследование

В случае наследования переопределяющие методы могут определять `throws` с более конкретным типом исключения (либо вовсе без него).

Но указывать более общий тип исключения в переопределяющих методах нельзя.

```
1  public class Parent {
2      public void foo() throws Exception { ... }
3      public void bar() throws Exception { ... }
4      public void qux() throws Exception { ... }
5      public void zoo() throws Exception { ... }
6  }
7  public class Child extends Parent {
8      @Override public void foo() throws Exception { ... } // OK
9      @Override public void bar() throws IOException { ... } // OK
10     @Override public void qux() { ... } // OK
11     @Override public void zoo() throws Throwable { ... } // ERROR
12 }
```

Собственные исключения

Стандартных исключений обычно не хватает, т.к. их типы слишком общие.

Можно определить свой тип исключения, создав класс-наследник от одного из уже определённых классов-исключений.

```
1  public class CalculatorStateException extends IllegalStateException {
2      public CalculatorStateException() {
3      }
4
5      public CalculatorStateException(String message) {
6          super(message);
7      }
8
9      public CalculatorStateException(String message, Throwable cause) {
10         super(message, cause);
11     }
12
13     public CalculatorStateException(Throwable cause) {
14         super(cause);
15     }
16 }
```

Блок finally (1)

Блок `finally` выполняется всегда — и при успешном завершении `try`, и при возникновении исключения.

Код из данного блока выполняется после выхода из `try` или `catch`.

Он полезен для гарантии очистки используемых ресурсов (например, для удаления временных файлов).

```
1  public int foo(String fileName) throws IOException {
2      File file = null;
3      try {
4          file = new File(fileName);
5          file.createNewFile();
6          // Какая-то работа с файлом
7      } catch (IOException e) {
8          throw new IOException("Файл: " + fileName, e);
9      } finally {
10         if (file != null) {
11             file.delete();
12         }
13     }
14 }
```

Блок finally (2)

Блок `finally` можно указывать даже при отсутствии `catch`.

```
1  public int foo(String fileName) {
2      File file = null;
3      try {
4          file = new File(fileName);
5          f.createNewFile();
6          // Какая-то работа с файлом
7      } finally {
8          if (file != null) {
9              file.delete();
10         }
11     }
12 }
```

Нюансы работы finally (1)

Если в try, catch и finally блоках есть return, то результатом всегда будет значение из finally.

```
1  public int foo(int a, int b) {
2      try {
3          return a / b;
4      } catch (ArithmeticException e) {
5          return -1;
6      } finally {
7          // Метод всегда будет возвращать 0
8          return 0;
9      }
10 }
```

Нюансы работы finally (2)

Также нужно учитывать это при работе с исключениями: если в блоке `finally` используются операторы `return`, `break`, `continue`, `throw`, которые могут передать управление вовне этого блока (например, завершить метод), то возникшее исключение будет "тихо" проигнорировано.

```
1 public int foo() {
2     try {
3         throw new IllegalStateException();
4     } finally {
5         // Метод всегда будет возвращать 0
6         return 0;
7     }
8 }
```

Это нужно учитывать особенно в том случае, если в `finally` находится код, который сам может сгенерировать исключение.

Гарантии безопасности

- **Гарантия отсутствия исключений.**
- **Сильные гарантии.** При возникновении исключения состояние объектов остаётся таким же, как и до возникновения.
- **Слабые гарантии.** При возникновении исключения объекты остаются в корректном состоянии (но изменённом).
- **Отсутствие утечек.** При возникновении исключения все ресурсы освобождаются.
- **Отсутствие гарантий.**