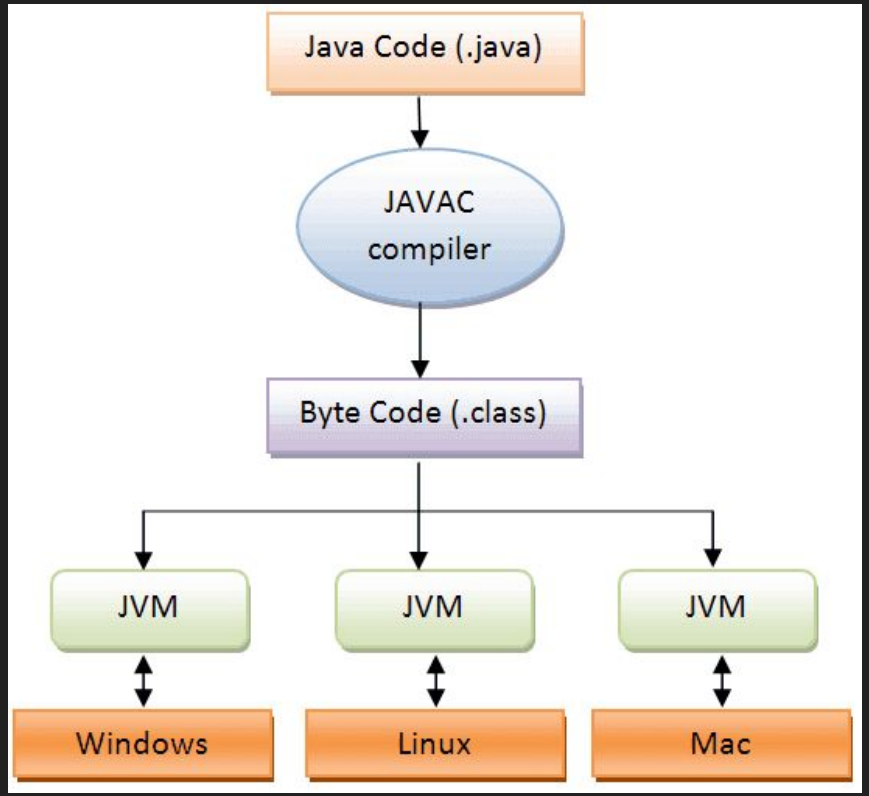
Механизм исключений в Java

Грищенко Илья, Иванов Андрей
5030102/30101

Этап 1: виртуальная машина Java (JVM)

1. Программист пишет код на Java – файлы **.java**
2. Компилятор Java переводит человеко-читаемый код в байт-код. Это специальный промежуточный формат – файлы **.class**
3. НО: байт-код это не машинные команды. Процессор не может выполнить байт-код
4. Поэтому нужен инструмент, который сможет интерпретировать байт-код в машинные команды – это и есть JVM
5. Байт-код можно будет использовать везде, где есть JVM

Этап 1: виртуальная машина Java (JVM)



Есть разные версии JVM - для Windows, для Linux и для Mac

JVM производит сразу несколько операций с байт-кодом:

- загружает байт-код
- интерпретирует байт-код
- управляет памятью
- следит за ошибками

Этап 2: определение исключения

- Во время выполнения программы могут возникать исключительные ситуации
- При возникновении исключения в приложении создается объект, который описывает это исключение.
- То есть каждой исключительной ситуации поставлен в соответствие некоторый класс, экземпляр которого инициируется при ее появлении.

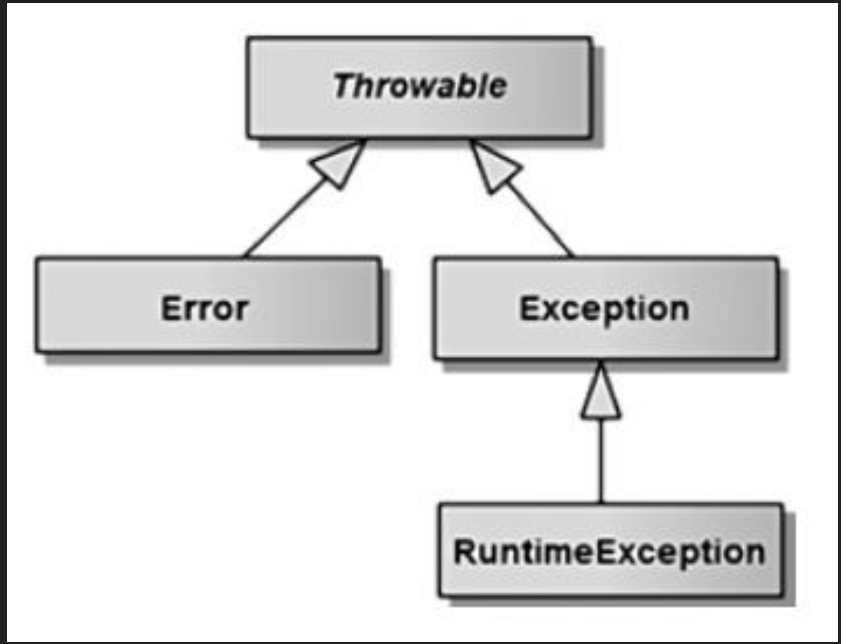
Этап 2: определение исключения, состав исключения

Что находится внутри класса исключения:

- **Тип исключения**
- **Сообщение об ошибке** – текстовое описание проблемы, которое читает человек.
- **Стек вызовов** – самая главная часть. Она позволяет разобраться где произошла проблема.
- **Причина** – часто одна ошибка является следствием другой. Объект-исключение может хранить ссылку на другое исключение, которое стало его причиной

Этап 3: Иерархия исключений

- Все исключения являются наследниками суперкласса Throwable
- Далее, есть два подкласса: Error и Exeption
- Подкласс Error – это фатальные для приложения ошибки, после них обычно нельзя восстановиться
- Подкласс Exception – это обрабатываемые исключения. Их можно и нужно обработать



Этап 3: Иерархия исключений, Error

Исключительные ситуации типа Error – это ошибки, связанные с работой JVM и системы.

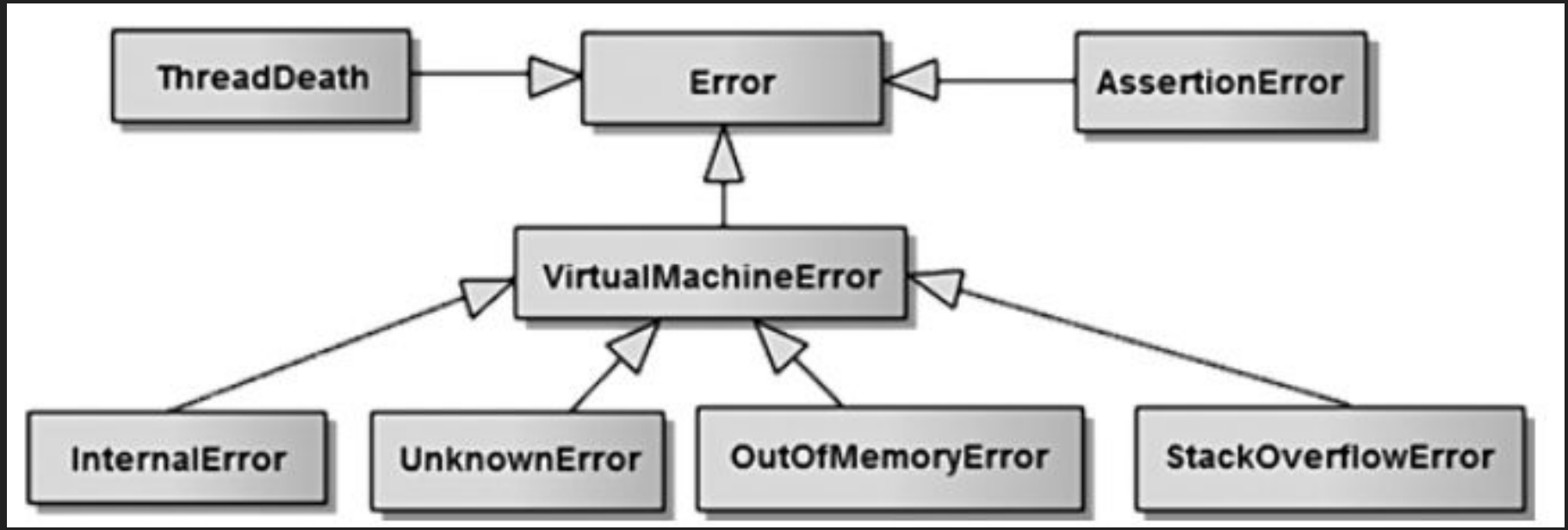
Такие исключения, связанные с серьезными ошибками, к примеру, с переполнением стека, не подлежат исправлению.

Они не могут обрабатываться приложением.

Основные примеры:

- Нехватка памяти
- Переполнение стека вызовов
- Внутренние ошибки JVM

Этап 3: Иерархия исключений, Error



Этап 3: Иерархия исключений, Exception

Exception — это подкласс Throwable, представляющий ошибки, которые можно и нужно обрабатывать в прикладном коде

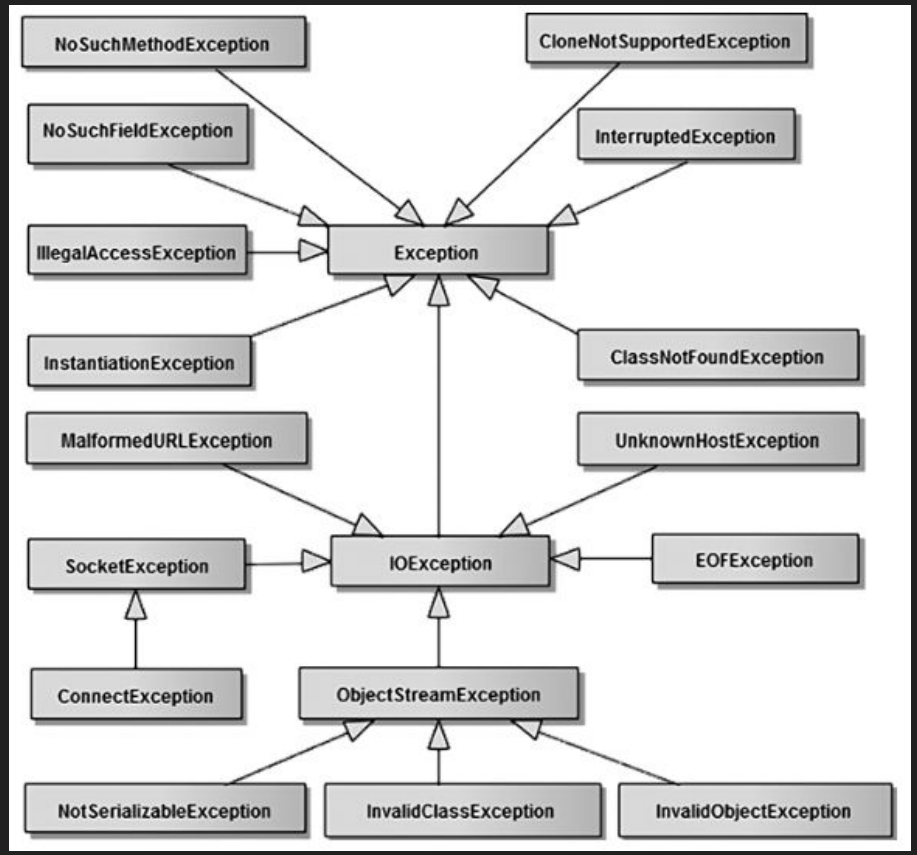
В отличие от Error, исключения Exception указывают на проблемы, от которых программа может восстановиться или корректно завершить свою работу

Они делятся на проверяемые (checked exceptions), требующие обязательной обработки, и непроверяемые (unchecked exceptions или RuntimeExceptions), возникающие обычно из-за ошибок программиста.

Этап 3: Иерархия исключений, разница Checked и Unchecked

- Checked Exceptions – это исключения, которые проверяются компилятором, и программист обязан либо обработать их в блоке try-catch, либо включить в throws.
- Unchecked Exceptions (RuntimeException и его наследники) не проверяются компилятором, и их обработка остается на усмотрение программиста.
- Обычно Checked Exceptions представляют внешние, ожидаемые ошибки (вроде "файл не найден"), а Unchecked Exceptions обычно указывают на ошибки в логике программы (вроде NullPointerException).

Этап 3: Иерархия исключений, Checked Exceptions



Этап 3: Иерархия исключений, Unchecked Exceptions

Исключение	Значение
ArithmeticException	Арифметическая ошибка: деление на ноль и др.
ArrayIndexOutOfBoundsException	Индекс массива находится вне его границ
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
ConcurrentModificationException	Некорректный способ модификации коллекции
IllegalArgumentException	При вызове метода использован некорректный аргумент
IllegalMonitorStateException	Незаконная операция монитора на разблокированном объекте
IllegalStateException	Среда или приложение находятся в некорректном состоянии
IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока
IndexOutOfBoundsException	Некоторый тип индекса находится вне границ коллекции
NegativeArraySizeException	Попытка создания массива с отрицательным размером
NullPointerException	Недопустимое использование ссылки на null
NumberFormatException	Невозможное преобразование строки в числовой формат
StringIndexOutOfBoundsException	Попытка индексации вне границ строки
MissingResourceException	Отсутствие файла ресурсов properties или имени ресурса в нем
EnumConstantNotPresentException	Несуществующий элемент перечисления
UnsupportedOperationException	Встретилась неподдерживаемая операция

Способы обработки исключений

На практике используется один из двух способов обработки исключений:

- перехват и обработка исключения в блоке **try-catch** метода;
- объявление исключения в секции **throws** метода и передача вызывающему методу.

try-catch метод

Конструкция **try-catch** по виду напоминает **if-else**, в блоке **try** пишется код где может возникнуть исключение (например, происходит открытие файла). В блок **catch**, пишется обработка исключений.

```
try {
// Блок кода, проверяемый на наличие ошибок.
}
catch (ExcepType1 exOb) {
// Обработчик исключения типа ExcepType1.
}
catch (ExcepType2 exOb) {
// Обработчик исключения типа ExcepType2.
}
```

try-catch метод

Если в блоке **try** может быть сгенерировано в разных участках кода несколько типов исключений, то необходимо наличие нескольких блоков **catch**, если только блок **catch** не обрабатывает все типы исключений. С версии **Java 7**, блоки **catch**, можно объединять в один с помощью логического символа “или” (|).

```
try {
  // some code
} catch(NumberFormatException e) {
  e.printStackTrace(); // or log
} catch(ClassNotFoundException e) {
  e.printStackTrace(); // or log
} catch(InstantiationException e) {
  e.printStackTrace(); // or log
}
```

```
try {
  // some code
} catch(NumberFormatException | ClassNotFoundException | InstantiationException e){
  e.printStackTrace();
}
```

try-catch метод

Операторы **try** можно вкладывать друг в друга. Если у оператора **try** низкого уровня нет раздела **catch**, соответствующего возникшему исключению, поиск будет развернут на одну ступень выше и будут проверены разделы **catch** внешнего оператора **try**.

```
try { // outer block
  int a = (int) (Math.random() * 2) - 1;
  System.out.println("a = " + a);
  try { // inner block
    int b = 1 / a;
    StringBuilder builder = new StringBuilder(a);
  } catch (NegativeArraySizeException e) {
    System.err.println("invalid buffer size: " + e);
  }
} catch (ArithmeticException e) {
  System.err.println("divide by zero: " + e);
}
```

Блок **finally**

Возможна ситуация, при которой нужно выполнить некоторые действия по завершении программы вне зависимости от того, произошло исключение или нет. В этом случае используется блок **finally**, который обязательно выполняется после или инструкции **try**, или **catch**.

Каждому разделу **try** должен соответствовать по крайней мере один раздел **catch** или блок **finally**. Код блока выполняется перед выходом из метода даже в том случае, если перед ним были выполнены такие инструкции, как **throws**, **return**, **break**, **continue**.

```
try {
  // code
} catch(OneException e) {
  // code // not required
} catch(TwoException e) {
  // code // not required
} finally {
  // executed after try or after catch */
}
```

Пример

```
public class Auth {
    4 usages
    private Map<String, String> users = new HashMap<>();

    1 usage
    public Auth() {
        users.put("admin", "123456");
        users.put("user", "password");
    }

    5 usages
    public boolean login(String username, String password) {
        try {
            // Проверка входных данных
            if (username == null || username.trim().isEmpty()) {
                throw new IllegalArgumentException("Имя пользователя не может быть пустым");
            }

            if (password == null || password.isEmpty()) {
                throw new IllegalArgumentException("Пароль не может быть пустым");
            }
```

Пример

```
            // Проверка существования пользователя
            if (!users.containsKey(username)) {
                throw new RuntimeException("Пользователь не найден");
            }

            // Проверка пароля
            String storedPassword = users.get(username);
            if (!storedPassword.equals(password)) {
                throw new RuntimeException("Неверный пароль");
            }

            System.out.println("Успешный вход: " + username);
            return true;

        } catch (IllegalArgumentException e) {
            System.out.println("Ошибка ввода: " + e.getMessage());
            return false;
        } catch (RuntimeException e) {
            System.out.println("Ошибка авторизации: " + e.getMessage());
            return false;
        } finally {
            System.out.println("Попытка входа завершена для: " + username);
        }
    }
```

Оператор **throw**

Метод может генерировать исключения, которые сам не обрабатывает, а передает для обработки другим методам, вызывающим данный метод. В этом случае метод должен объявить о таком поведении с помощью ключевого слова **throws**, чтобы вызывающий метод мог защитить себя от этих исключений. В вызывающем методе должна быть предусмотрена или обработка этих исключений, или последующая передача соответствующему методу.

Пример

```
class AuthService {
    4 usages
    private Map<String, String> users = new HashMap<>();

    1 usage
    public AuthService() {
        users.put("user1", "pass1");
        users.put("user2", "pass2");
    }

    1 usage
    public boolean login(String username, String password)
            throws IllegalArgumentException, IllegalStateException {

        if (username == null || username.trim().isEmpty()) {
            throw new IllegalArgumentException("Имя пользователя не может быть пустым");
        }

        if (password == null || password.trim().isEmpty()) {
            throw new IllegalArgumentException("Пароль не может быть пустым");
        }
```

Пример

```
        if (!users.containsKey(username)) {
            throw new IllegalStateException("Пользователь не найден");
        }

        if (!users.get(username).equals(password)) {
            throw new IllegalStateException("Неверный пароль");
        }

        System.out.println("Успешный вход: " + username);
        return true;
    }
}

public class Main {
    public static void main(String[] args) {
        AuthService auth = new AuthService();
        try {
            auth.login( username: "user1",  password: "pass11");
        } catch (IllegalStateException e) {
            System.out.println("Ошибка входа: " + e.getMessage());
        } catch (IllegalArgumentException e) {
            System.out.println("Ошибка данных: " + e.getMessage());
        }
```

Жизненный цикл исключений в java

1. Создание исключения (Exception Creation)
2. Бросок исключения (Throwing)
    a. Нормальный поток выполнения прерывается
    b. JVM начинает поиск обработчика исключения
    c. Текущий метод завершается досрочно
    d. Управление передается механизму обработки исключений
3. Перехват исключения (Catching)
4. Проброс исключения (Propagating)
    a. Когда исключение брошено в методе, его выполнение немедленно прерывается
    b. JVM ищет обработчик исключения в текущем методе
    c. Если обработчик не найден, исключение "пробрасывается" в вызывающий метод
    d. Процесс повторяется рекурсивно до тех пор, пока не будет найден подходящий обработчик или не достигнуто начало стека
5. Обработка или завершение

Список источников:

- Java from EPAM
- metanit.com/java/tutorial/4.2.php
- https://javarush.com/groups/posts/1944
- https://progatochkapy.blogspot.com/p/jvm-java.html