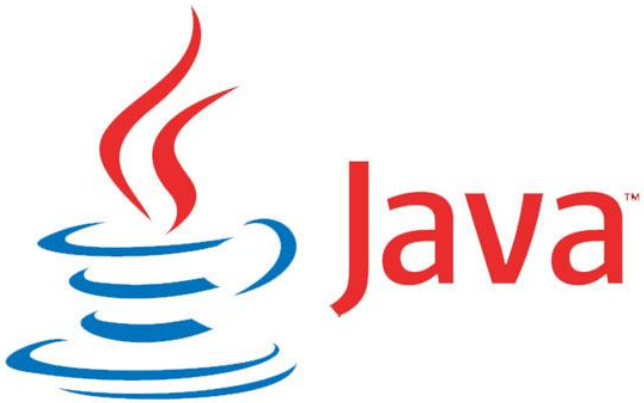
Тема 9.
Исключения.



Понятие исключения

Исключение — это проблема(ошибка) возникающая во время выполнения программы. Исключения могут возникать во многих случаях, например:

- Пользователь ввел некорректные данные.
- Файл, к которому обращается программа, не найден.
- Сетевое соединение с сервером было утеряно во время передачи данных.

Все исключения в Java являются объектами. Поэтому они могут порождаться не только автоматически при возникновении исключительной ситуации, но и создаваться самим разработчиком.

Типы исключений

Контролируемые исключения (Checked exception) — контролируемое исключение представляет собой вид исключения, которое **происходит на стадии компиляции**, их также именуют исключениями периода компиляции. Обозначенные исключения не следует игнорировать в ходе компиляции, они требуют должного обращения (разрешения) со стороны программиста.

К примеру, если вы используете класс **FileReader** в вашей программе для считывания данных из файла, в случае, если указанный в конструкторе файл не существует, происходит **FileNotFoundException**, и программист **обязан** создать механизм обработки данной ситуации.

Типы исключений


```
import java.io.File;
import java.io.FileReader;

public class Test {

   public static void main(String args[]) {
      File f = new File("D://java/file.txt");
      FileReader fr = new FileReader(f);
   }
}
```

При попытке компиляции обозначенной выше программы будут выведены следующие исключения:

```
C:\>javac Test.java
Test.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
      FileReader fr = new FileReader(f);
                      ^
1 error
```

FileNotFoundException относится к **checked exceptions**. Вы не сможете скомпилировать программу, пока не напишете код, который, в случае этой ошибки, позволит программе работать дальше.

Типы исключений

Неконтролируемые исключения (Unchecked exception) — представляет собой исключение, которое п**роисходит во время выполнения**.

Данная категория может включать погрешности программирования, такие как логические ошибки либо неверный способ использования API. **Исключения на этапе выполнения игнорируются в ходе компиляции.**

К примеру, если вами в вашей программе был объявлен массив из 5 элементов, попытка вызова 6-го элемента массива повлечет за собой возникновение **ArrayIndexOutOfBoundsExceptionexception**.

Типы исключений

```
public class Test {

   public static void main(String args[]) {
      int array[] = {1, 2, 3};
      System.out.println(array[4]);
   }
}
```

После успешной компиляции, уже на этапе выполнения будет получено следующее исключение:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Exceptions.Test.main(Test.java:8)
```

Программа закроется с ошибкой. Такие исключения не обязательно как-то предусматривать в коде. Но желательно хотя бы продумывать код таким образом, чтобы они не возникали.

Типы исключений

Ошибки (Error) — не являются исключениями, однако представляют проблемы, которые возникают независимо от пользователя либо программиста.

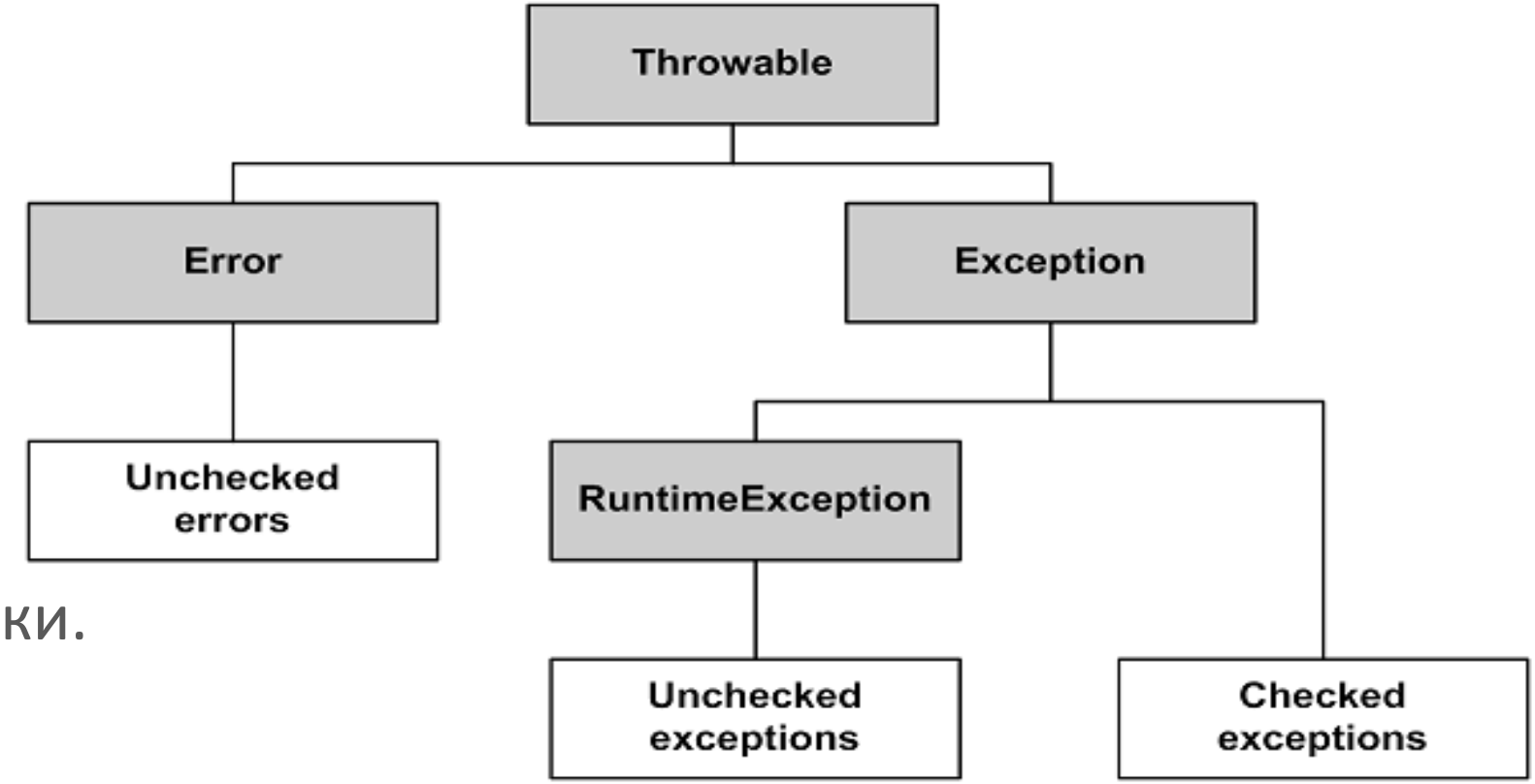
Ошибки в вашем коде обычно игнорируются в виду того, они связаны с проблемами уровня JVM. Например, исключения такого рода возникают, если закончилась память, доступная виртуальной машине. Программа дополнительную память всё равно не сможет обеспечить для JVM.

На этапе компиляции они также игнорируются.

Иерархия исключений

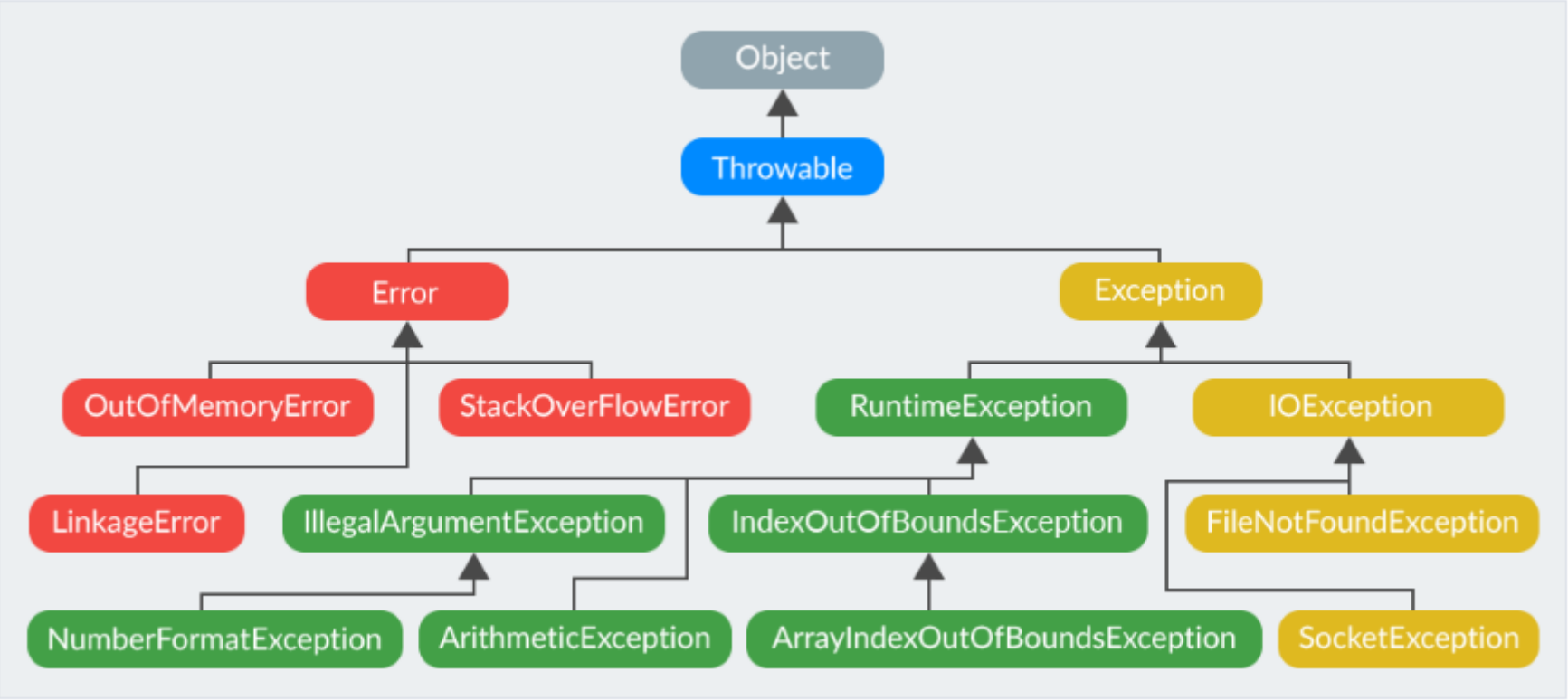
Исключения делятся на несколько классов, но все они имеют общего предка — класс **Throwable**. Его потомками являются подклассы **Exception** и **Error**.

- **Error** - критические ошибки. Нет смысла обрабатывать.
- **RuntimeException** - ошибки в ходе выполнения. Обрабатываются по желанию.
- **Checked Exception** - прогнозируемые ошибки. Необходимо обрабатывать.



Иерархия исключений

На деле каждый вид этих исключений содержит под собой огромное количество уже готовых специфических исключений. Ниже приведена малая их часть.



Методы исключений

- **public String getMessage()**
  Возврат подробного сообщения о произошедшем исключении.
- **public String toString()**
  Возврат имени класса, соединенного с результатом getMessage().
- **public void printStackTrace()**
  Выведение результата toString() совместно с трассировкой стека в System.err, поток вывода ошибок.
- **public StackTraceElement [] getStackTrace()**
  Возврат массива, содержащего каждый элемент в трассировке стека. Элемент с номером 0 представляет вершину стека вызовов, последний элемент массива отображает метод на дне стека вызовов.

StackTrace


```
Exception in thread "main" java.lang.NullPointerException
        at com.example.myproject.Book.getTitle(Book.java:16)
        at com.example.myproject.Author.getBookTitles(Author.java:25)
        at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

Простыми словами, **трассировка стека** – это список методов, которые были вызваны до момента, когда в приложении произошло исключение.

Это пример очень простой трассировки. Если пойти по списку строк вида «at…» с самого начала, мы можем понять, где произошла ошибка. Мы смотрим на верхний вызов функции. В нашем случае, это:

```
at com.example.myproject.Book.getTitle(Book.java:16)
```

Именно в этой строчке что-то вызвало нашу ошибку. И теперь мы знаем точно куда смотреть.

Обработка

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 создается и передается дальше.
- Ваш код может перехватить исключение при помощи блока **catch** и обработать его.
- Любой код, который следует выполнить обязательно после завершения блока try, помещается в блок **finally**. Этот **блок является не обязательным**.

```
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
}
```

Обработка

Ниже представлен массив с заявленными двумя элементами. Попытка кода получить доступ к третьему элементу массива повлечет за собой генерацию исключения.

```
import java.io.*;

public class Test {

   public static void main(String args[]) {
      try {
         int array[] = new int[2];
         System.out.println("Доступ к третьему элементу:" + array[3]);
      }catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Исключение:" + e);
      }
      System.out.println("Вне блока");
   }
}
```

ArrayIndexOutOfBoundsEcpetion относится к **RuntimeException** и мы не обязаны его обрабатывать. Однако мы можем это сделать, если считаем необходимым.

Вследствие этого будет получен следующий результат:

```
Исключение:java.lang.ArrayIndexOutOfBoundsException: 3
Вне блока
```

Обработка

За блоком **try** могут следовать несколько блоков **catch**. Синтаксис многократных блоков **catch** выглядит следующим образом:

```
try {
   // Защищенный код
}catch(ИсключениеТип1 e1) {
   // Блок catch
}catch(ИсключениеТип2 e2) {
   // Блок catch
}catch(ИсключениеТип3 e3) {
   // Блок catch
}
```

В случае возникновения исключения в защищенном коде, исключение выводится **в первый блок catch** в списке. Если тип данных генерируемого исключения совпадает с ИсключениеТип1, он перехватывается в указанной области.

В обратном случае, исключение переходит **ко второму оператору catch** и наше исключение сравнивается с ИсключениеТип2.

Это продолжается до тех пор, пока не будет произведен перехват исключения, либо оно не пройдет через все операторы, в случае чего выполнение текущего метода будет прекращено, и исключение будет перенесено к предшествующему методу в стеке вызовов.

Чаще всего это используется когда код в блоке try может вызвать одну из несколько не связанных друг с другом ошибок. И каждый случай необходимо обрабатывать по разному.

Обработка

Блок **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**.

Блок будет выполнен, независимо от того, передано исключение или нет.

Оператор **finally** не обязателен, однако каждый оператор try требует наличия либо **catch**, либо **finally**.

Код в блоке finally будет выполнен всегда, catch - нет.

```
try {

    // Сделать что-то здесь.
} catch (Exception1 e) {

    // Сделать что-то здесь.
} catch (Exception2 e) {

    // Сделать что-то здесь.
} finally {

    // Блок finally всегда выполняется.
    // Сделать что-то здесь.
}
```

Обработка

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).

В фрагменте **список_исключений** можно указать список исключений через запятую.

```
тип имя_метода(список_параметров) throws список_исключений {
    // код внутри метода
}
```

Обработка

Оператор **throw** служит для принудительной генерации исключения в самом методе. В купе с **throws** он заставляет вызывающий метод обработать это исключение, либо передать его еще выше.

```
// Без изменений
public void createCat() throws NullPointerException {
        throw new NullPointerException("Кота не существует");
}

// Щелчок кнопки
public void onClick(View v) {
        try {
                createCat();
        } catch (NullPointerException e) {
                // TODO: handle exception
        }
}
```

Custom exceptions

Хотя имеющиеся в стандартной библиотеке классов Java классы исключений описывают большинство исключительных ситуаций, которые могут возникнуть при выполнении программы, все таки иногда требуется создать свои собственные классы исключений со своей логикой.

Custom exceptions

При создании собственных классов исключений следует принимать во внимание следующие аспекты:

- Все исключения должны быть дочерними элементами **Throwable**.
- Если вы планируете создать исключение этапа компиляции, вам следует расширить класс **Exception**.
- Если вы хотите создать исключение этапа выполнения, вам следует расширить класс **RuntimeException**.

Custom exceptions

Чтобы создать свой класс исключений, надо унаследовать его от класса **Exception** или **RuntimeException**.

```
class FactorialException extends Exception{

    private int number;
    public int getNumber(){return number;}
    public FactorialException(String message, int num){

        super(message);
        number=num;
    }
}
```