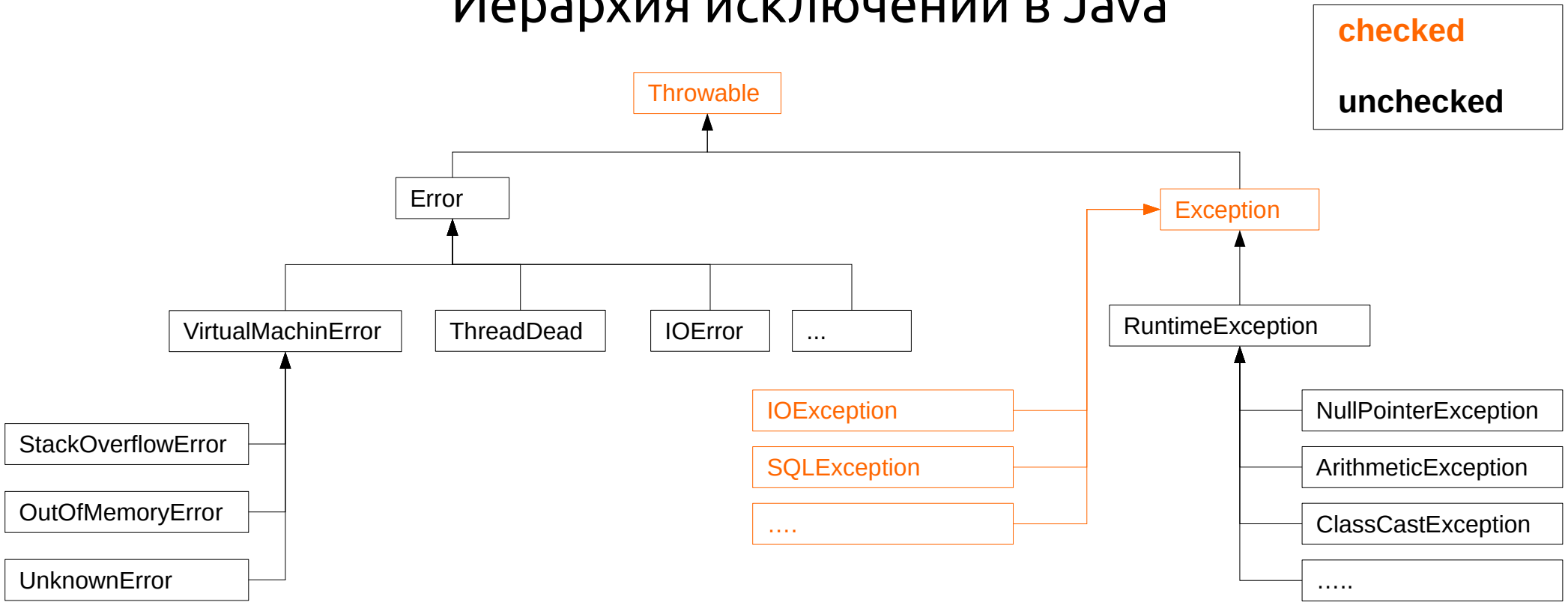
# Работа с исключениями

# Исключение определение

Исключение (Exception) сокращение термина «Исключительное событие» - событие которое происходит во время выполнения программы и нарушает нормальный поток выполнения операторов программы.

Когда в методе возникает исключение, метод создает объект исключения и передает его системе выполнения (что в свою очередь запускает механизм обработки исключения). Объект исключения, содержит информацию об ошибке, включая ее тип и состояние программы на момент возникновения ошибки. Создание объекта исключения и передача его системе времени выполнения называется генерацией исключения.

# Иерархия исключений в Java



# Проверяемые исключения

Проверяемые исключения (checked)- исключения, корректность обработки которых проверяется на этапе компиляции приложения. Наличие проверяемых исключений накладывает ряд ограничений:

- В описании функции (или метода класса) в явном виде перечисляются все типы исключений, которые она может генерировать.
- Функция, вызывающая функцию или метод с объявленными исключениями, для каждого из этих исключений обязана либо содержать обработчик, либо, в свою очередь, указывать этот тип как генерируемый ею в своём описании.
- Компилятор проверяет наличие обработчика в теле функции или записи исключения в её сигнатуре. Если компилятор обнаруживает возможность возникновения исключения, которое не описано в заголовке функции и не обрабатывается в ней, программа считается некорректной и не компилируется.

В Java это наследники класса java.lang.Exception и java.lang.Throwable

# Необходимость проверяемых исключений

Проверяемые исключения снижают количество ситуаций, когда исключение, которое могло быть обработано, вызвало критическую ошибку в программе, поскольку за наличием обработчиков следит компилятор. Это особенно полезно при изменениях кода, когда метод, который не мог ранее выбрасывать исключение типа X, начинает это делать; компилятор автоматически отследит все случаи его использования и проверит наличие соответствующих обработчиков.

Другим полезным качеством проверяемых исключений является то, что они способствуют осмысленному написанию обработчиков: программист явно видит полный и правильный список исключений, которые могут возникнуть в данном месте программы, и может написать на каждое из них осмысленный обработчик вместо того, чтобы создавать «на всякий случай» общий обработчик всех исключений, одинаково реагирующий на все нештатные ситуации.

# Непроверяемые исключения

Непроверяемые исключения - исключения, корректность обработки которых не проверяется на этапе компиляции приложения. К непроверяемым исключениям обычно относят исключения двух принципиально разных видов:

- Исключения времени выполнения, обычно связанные с ошибками программиста. Такие исключения возникают из-за логических ошибок разработчика или недостаточности проверок в коде. Например, ошибка обращения по неинициализированному (нулевому) указателю, как правило, означает, что программист либо пропустил где-то инициализацию переменной, или ошибка выхода за пределы массива. Как первое, так и второе требует исправления кода программы, а не создания обработчиков. В Java это наследники класса java.lang.RuntimeException

- Исключения, представляющие собой серьезные ошибки, которые, «по идее», возникать не должны, и которые в обычных условиях не следует обрабатывать программой. Такие ошибки могут возникать как во внешней относительно программы среде, так и внутри неё. Примером такой ситуации может быть ошибка среды исполнения программы на Java. Она потенциально возможна при исполнении любой команды; за редчайшими исключениями в прикладной программе не может быть осмысленного обработчика подобной ошибки — ведь если среда исполнения работает неверно, на что указывает сам факт исключения, нет никакой гарантии, что и обработчик будет исполнен правильно. В Java это наследники класса java.lang.Error

# Обработка исключений

Обработка исключений (обработка исключительных ситуаций) - механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Обработка исключений программой - процесс передачи управления заранее определенному обработчику исключений при возникновении исключительной ситуации.

Обработчик исключений - блок кода, процедура, функция, выполняющая необходимые действия при возникновении исключения.

Обработка исключения по умолчанию - набор действий (в Java это вывод на экран трассировки исключения, завершение приложения), выполняемых при отсутствии обработчика возникшего исключения.

# Различные механизмы обработчиков исключений

**Обработка с возвратом** подразумевает, что обработчик исключения ликвидирует возникшую проблему и приводит программу в состояние, когда она может работать дальше по основному алгоритму. В этом случае после того, как выполнится код обработчика, управление передается обратно в ту точку программы, где возникла исключительная ситуация, и выполнение программы продолжается. **В Java в явном виде не реализованы.**

**Обработка без возврата** заключается в том, что после выполнения кода обработчика исключения управление передается в некоторое заранее заданное место программы, и с него продолжается исполнение. То есть, фактически, при возникновении исключения команда, во время работы которой оно возникло, заменяется на безусловный переход к заданному оператору. **Под обработкой исключений в Java подразумевается именно такой механизм.**

# Обработчик try - catch

Блок контролируемого кода вкладывается в блок try. Блок try определяет локальную область видимости (поэтому все что объявлено в нем не доступно за его пределами).

Обработчик исключений реализован с помощью синтаксической конструкции catch. Блок характеризуется описанием типа обрабатываемого исключения и блока кода, выполняемого при обработке исключения. Исключение обрабатывается или в случае точного совпадения типа, или в случае, если в блоке catch описан суперкласс обрабатываемого исключения. Т.е. блок catch полиморфен.

В общем случае конструкция выглядит следующим образом:

```
try{
    Контролируемый код
} catch (Exception_type e){
    Код выполняемый при обработке исключения
}
```

# Пример обработчика try - catch

```
import java.io.File;
import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        File file = new File("a.txt");

        try {

            file.createNewFile();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Контролируемый код

Обработчик catch

# Блок try-catch полиморфен

```
int[] arr = new int[3];

try {

    arr[5] = 10;

} catch (RuntimeException e) {

    System.out.println("catch " + e.getClass());

}
```

◄ Создается исключение ArrayIndexOutOfBoundsException

◄ Перехват RuntimeException

В блоке контролируемого кода возникает ArrayIndexOutOfBoundsException. Но обработка исключения все равно происходит, так как ArrayIndexOutOfBoundsException это наследник RuntimeException на который «нацелен» блок catch.

# Несколько блоков catch принадлежащих одному блоку try

Блоку try может соответствовать несколько блоков catch. Это используется для обработки исключений разных типов которые могут быть запущенны в блоке try. Порядок проверки на соответствие блоков catch сверху вниз. Первый подходящий блок catch выполняется, остальные пропускаются.

```
try{
        Контролируемый код
} catch (Exception_type e){
        Код выполняемый при обработке исключения
} catch (Exception_type e){
        Код выполняемый при обработке исключения
}
```

# Несколько блоков catch принадлежащих одному блоку try

Предположим есть файл в котором сохранена стоимость товара. Нужно вычитать эту стоимость из файла. Но ведь может произойти две ситуации:

1) Адрес файла указан не верно
2) Формат данных в файле не верен

```
File file = new File("price.txt");
Integer price = null;

try {

    Scanner sc = new Scanner(file);
    price = sc.nextInt();

} catch (IOException e) {
    System.out.println("File not found");
} catch (InputMismatchException e) {
    System.out.println("Error file format");
}
System.out.println("price = " + price);
```

◄ Обрабатываем неверный адрес

◄ Обрабатываем неверный формат

# Правило при наличии нескольких блоков catch

**Нельзя описывать catch перехватывающий подкласс исключения после блока catch который перехватывает суперкласс этого исключения!**

```
int[] array = new int[3];

try {
    array[5] = 10;
} catch (Exception e) {
    System.out.println(":)");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println(":(");
}
```

◄ Ошибка компиляции

Тут мы получим ошибку компиляции в виде недостижимого кода. Дело в том, что любое исключение это наследник Exception и будет перехвачено первым блоком catch. Второй блок не выполниться не при каких условиях, по сути это и есть недостижимый код.

# Обработка нескольких исключений одним блоком catch

В случае, когда нужно обработать насколько исключений разного типа с помощью одного блока catch, можно использовать мультиобработчик исключений. Для его использования в блоке catch можно перечислить несколько типов исключений разделив с помощью оператора «|». В таком случае этот блок catch перехватит любое из перечисленных типов исключений.

```
int[] array = new int[10];
try {
    array[20] = 1;

} catch (ArrayIndexOutOfBoundsException | NullPointerException e) {
    System.out.println(e);
}
```

Мультиобработчик

В примере блок catch способен перехватить исключение типа `ArrayIndexOutOfBoundsException` или `NullPointerException`.

# Вложенность блоков try-catch

Блоки try-catch можно вкладывать в друг друга. Глубина вложенности не ограничивается. Если исключение возникает во вложенном блоке try и при этом не перехватывается, то выполняется проверка для внешнего.

```
int[] array = new int[10];
try {

    try {
        array[20] = 10;
    } catch (NullPointerException e) {
        System.out.println(e);
    }

} catch (ArrayIndexOutOfBoundsException e) {
        System.out.println(e);
}
```

Возбуждение исключения

Перехват исключения

# Блок с гарантированным завершением

Блок с гарантированным завершением - блок кода, выполнение которого гарантируется вне зависимости от результата обработки исключения. В Java реализован в виде блока **finally**. Этот блок не обрабатывает исключение, а лишь гарантирует выполнение определенного набора операций. Блок finally всегда должен быть завершающим.

```
try{
        Контролируемый код
} catch (Exception_type e){
        Код выполняемый при обработке исключения
}finally{
        Блок кода который гарантированно выполнится
}
```

# Пример try-catch-finally

```
File file = new File("price.txt");
Integer price = null;
try {
    Scanner sc = new Scanner(file);
    price = sc.nextInt();
} catch (IOException e) {
    System.out.println("File not found");
} catch (InputMismatchException e) {
    System.out.println("Error file format");
} finally {
    System.out.println("Thank you for using our service :) ");
}

System.out.println("price = " + price);
```

Контролируемый код

Обработчики catch

Блок с гарантированным завершением

# Использование блока try-catch-finally

Для блока try-catch-finally блок кода finally сработает в в следующих случаях:

1) Блок try завершился успешно
2) Блок try завершился с исключением (независимо от того перехвачено оно или нет блоком catch)
3) В блоке try или catch есть оператор возврата из метода (return)

Блок finally не сработает в следующих случаях:

1) В блоке try инструкция остановки JVM (System.exit(0), Runtime.getRuntime().exit(0))

Внимание! Блок finally не обработает исключение. Он просто выполниться и все.

# Особенности применения блока try-catch-finally

```
public static void main(String[] args) {
    int[] array = new int[] { 0, -2, 7 };

    System.out.println(getElement(array, 10));
}

public static int getElement(int[] array, int index) {
    try {
        int number = array[index];
        return number;
    } catch (ArrayIndexOutOfBoundsException e) {
        return -1;
    } finally {
        System.out.println("Finally");
    }
}
```

Выполнится перед возвратом управления

И хотя в блоках try-catch есть оператор return сначала выполнится блок finally и только потом метод вернет управление.

# Особенности применения блока try-catch-finally

```
public static int getElement(int[] array, int index) {
    try {
        int number = array[index];
        return number;
    } catch (ArrayIndexOutOfBoundsException e) {
        return -1;
    } finally {
        return 11;
    }
}
```

Метод всегда будет возвращать число 11

Если в блоке finally находится оператор return, то метод будет возвращать всегда именно это значение.

# Особенности применения блока try-catch-finally

```
public static int getElement(int[] array, int index) {
    try {
        int number = array[index];
        return number;
    } catch (NullPointerException e) {
        return -1;
    } finally {
        return 11;
    }
}
```

Метод всегда будет возвращать число 11

Если в блоке finally находится оператор return и в методе try возникает исключение, то оно будет отброшено и метод вернет значение указанное оператором return в блоке finally.

# Вопросы и ответы к ним

**Стоит ли использовать оператор return в блоке finally?**

Нет. Это плохая практика так, можно скрыть что в блоке try-catch произошло исключение.

**Можно ли в блоке finally использовать код который может привести к генерации исключения?**

Да. Однако это тоже негативная практика по аналогичной причине.

**Можно ли опустить блок catch в конструкции try-catch-finally?**

Да. Получится блок try-finally. Который не будет обрабатывать исключения, а только выполнять определенные набор действий (описанный в блоке finally) при их возникновении. Правда стоит отметить, что такая практика не очень распространена.

# Использование блока try-finally

Для блока try-finally блок кода finally сработает в в следующих случаях:

1) Блок try завершился успешно
2) Блок try завершился с исключением
3) В блоке try есть оператор возврата из метода (return)

Блок finally не сработает в следующих случаях:

1) В блоке try инструкция остановки JVM (System.exit(0), Runtime.getRuntime().exit(0))

Внимание! Блок finally не обработает исключение. Он просто выполниться и все.

# Пример применения блока try-finally

```
int[] array = new int[10];

try {
    array[20] = 1;
} finally {
    System.out.println("Finally ");
}
```

Исключение возникшее в блоке try приводит к завершению программы, но перед этим выполнится содержимое блока finally.

# Оператор throw

Оператор throw предназначен для запуска процесса обработки исключения. Синтаксис применения данного оператора:

**throw** Throwable_ref;

где Throwable_ref — ссылка на объект класса наследника от Throwable.

Внимание! Создание объекта класса исключение с помощью оператора new не запустит процесс его обработки. Для запуска процесса обработки используется оператор **throw**.

# Создание объекта исключения не запускает процесс его обработки

```
NullPointerException npe = new NullPointerException(":)");

System.out.println("We work anyway");
```

И хотя был создан объект класса NullPointerException, это не запустит процесс его обработки.

# Запуск процесса обработки исключения

```
NullPointerException npe = new NullPointerException(":)");

throw npe;
```

◄ Запуск процесса обработки исключения

```
throw new NullPointerException(":)");
```

◄ Запуск процесса обработки исключения

```
int[] array = new int[10];

try {
    array[20] = 1;
} catch (ArrayIndexOutOfBoundsException e) {

    throw e;
}
```

◄ Запуск процесса обработки перехваченного исключения

# Вопросы и ответы к ним

**Исключения какого типа можно запустить в обработку с помощью оператора throw?**

Любые доступный тип исключения. По сути любой наследник Throwable.

**Что будет если передать значение ссылки null?**

В таком случае будет возбужденно исключение NullPointerException.

**Для чего запускать обработку перехваченного исключения?**

Это довольно распространенный прием. Его суть в том, что бы выполнить ряд действий при перехвате исключения и запустить его обработку снова, для его дальнейшего продвижения по стеку вызовов методов.

# Работа с IllegalArgumentException

В Java существует тип исключения IllegalArgumentException (не проверяемое исключение), этот тип исключения запускается в обработку если значение входящего параметра метода некорректно. Обычно связанно с тем, что подобное значение аргумент может получить только в результате ошибки полученной ранее. Желательно не перехватывать исключения такого типа.

```
public static int calculateNewPrice(int oldPrice, int discountPercent) {
    if (discountPercent < 0 || discountPercent > 100) {
        throw new IllegalArgumentException("Invalid discount value");
    }
    int newPrice = oldPrice - oldPrice * discountPercent / 100;
    return newPrice;
}
```

В примере реализован метод для вычисления новой стоимости с учетом скидки (выражена в процентах). Если размер скидки меньше 0 и больше 100 процентов, то это явно логическая ошибка допущенная ранее. В таком случае создается и запускается исключение типа IllegalArgumentException.

# Оператор throws

Оператор **throws** используется для указания того, что в методе может возникнуть проверяемое исключение, которое в самом методе не обрабатывается. Важным моментом является то, что это имеет смысл только для проверяемых исключений. Для непроверяемых является разве что элементом документации. Данный оператор становится частью сигнатуры метода (нужно учитывать при переопределении и реализации). Синтаксис его применения таков:

type method_name(parameters) **throws** exception_list

exception_list — список типов проверяемых исключений перечисленных через запятую.

Если в сигнатуре метода указанно, что он генерирует проверяемые исключения то вызов этого метода должен производиться в блоке try-catch или в методе в сигнатуре которого также указанно генерация этого исключения.

С помощью оператора throws реализуется механизм всплытия для исключений проверяемого типа.

# Механизм всплытия исключения

Для исключений непроверяемого типа по умолчанию реализуется механизм «всплытия» исключения. Поиск обработчика для исключения начинается от точки его возникновения и передвигаясь вверх по стеку вызовов методов. Если обработчик будет найден, то управление будет передано ему.

```
public static void main(String[] args) {
	int[] array = new int[] { -2, 0, 7, -1 };
	System.out.println(getRandomElement(array));
}

public static int getRandomElement(int[] array) {
	int randomIndex = (int) (Math.random() * array.length * 1.5);
	int number = 0;
	try {
		number = getElement(array, randomIndex);
	} catch (ArrayIndexOutOfBoundsException e) {
		e.printStackTrace();
	}
	return number;
}

public static int getElement(int[] array, int index) {
	int result = array[index];
	return result;
}
```

Перехват исключения и его обработка

Запуск процесса обработки исключения

# Демонстрация оператора throws

```
public static void main(String[] args) {

     File file = new File("price.txt");
     try {
          System.out.println(getTextFromFile(file));
     } catch (IOException e) {
          e.printStackTrace();
     }
}

public static String getTextFromFile(File file) throws IOException {
     Scanner sc = new Scanner(file);
     try {
          String result = "";
          for (; sc.hasNextLine();) {
               result += sc.nextLine() + System.lineSeparator();
          }
          return result;
     } finally {
          sc.close();
     }
}
```

Обработка в точке вызова метода

Метод может генерировать исключение

# Особенности работы throws

Следует помнить о следующих особенностях работы оператора throws:

1) Вы можете указать после throws несколько исключений
2) Вы можете указать после throws суперкласс генерируемого исключения
3) Метод может не генерировать указанное исключение
4) При переопределении метода с thows в сигнатуре, нельзя указывать суперкласс исключения указанного в методе суперкласса

# Перечисление нескольких исключений

```
public static void main(String[] args) {

     File file = new File("price.txt");
     try {
          System.out.println(getTextFromFile(file));
     } catch (IOException e) {
          e.printStackTrace();
     }

}

public static String getTextFromFile(File file) throws FileNotFoundException, IllegalStateException {
     Scanner sc = new Scanner(file);
     try {
          String result = "";
          for (; sc.hasNextLine();) {
               result += sc.nextLine() + System.lineSeparator();
          }
          return result;
     } finally {
          sc.close();
     }
}
```

Метод может генерировать несколько исключений

# Указание в throws суперкласса генерируемого исключения

```
public static String getText(File file) throws IOException{

    throw new FileNotFoundException();
}
```

В примере указанно, что метод может генерировать IOException. В теле метода генерируется FileNotFoundException (подкласс IOException). Перехватывать нужно указанное (а не возбужденное) исключение.

```
try {
    System.out.println(getText(new File("price.txt")));
} catch (IOException e) {
    e.printStackTrace();
}
```

# Метод не генерирует указанное исключение

```
public static void main(String[] args) {

     String catCSV = "Vaska;4";

     try {
          Integer age = getCatAge(catCSV);
          System.out.println(age);
     } catch (ParseException e) {
          e.printStackTrace();
     }
}

public static Integer getCatAge(String catCSV) throws ParseException {
     String[] arr = catCSV.split(";");
     return Integer.valueOf(arr[1]);
}
```

В примере указанно, что метод может генерировать ParseException. При этом в теле метода не генерируется это исключение, хотя при вызове его нужно обрабатывать. Такой подход часто используют при проектировании методов, которые в дальнейшем возможно начнут генерировать исключения указанного типа.

# Переопределение методов с throws в сигнатуре

```
class A {
     public String getText() throws IOException {
          return "Hello";
     }
}
```

```
class B extends A {

     @Override
     public String getText() {
          return "World";
     }
}
```

Можно не указывать тип исключения

```
class C extends A{

     @Override
     public String getText() throws FileNotFoundException {
          return "Java";
     }
}
```

Можно указать подкласс указанного исключения

```
class D extends A{

     @Override
     public String getText() throws Exception {
          return "Oops";
     }
}
```

Ошибка компиляции

Нельзя указать суперкласс указанного исключения!

# Связанные исключения (цепочки исключений)

В ряде случаев одно исключение может порождать появление другого и дальше происходит распространение второго исключения. Но в ряде случаев нужно узнать, что стало причиной появления исключения. В таком случае используются связанные исключения (цепочки исключений).

Для поддержки связанных исключений в Java определенны два конструктора:

- Throwable(String, Throwable)
- Throwable(Throwable)

Параметром конструктора выступает исключение которое стало причиной появление текущего исключения.

Также определенны два метода для удобства работы с такими исключениями:

- Throwable qetCause() - вернет прикрепленное исключение
- Throwable initCause (Throwable thr) — прикрепить исключение (thr) к текущему, для получения связанных исключений.  Для существующего исключения можно вызвать только один раз.

# Пример применения связанных исключений

```
public static void main(String[] args) {

     String text = null;
     try {
          System.out.println(createDateFromString(text));
     } catch (ParseException e) {
          e.printStackTrace();
     }
}

public static Date createDateFromString(String text) throws ParseException {
     SimpleDateFormat sdf = new SimpleDateFormat("dd:MM:yyyy");
     try {
          Date date = sdf.parse(text);
          return date;
     } catch (NullPointerException e) {
          throw (ParseException) new ParseException("Wrong", 0).initCause(e);
     }
}
```

Связываем NullPointerException с ParseException

В примере исключение может возникнуть по двум причинам: неверный формат строки и значение ссылки равное null. При перехвате NullPointerException оно прикрепляется к новому исключению типа ParseException в качестве причины и отправляется в обработку.

# Создание пользовательских исключений

Для создания исключений пользовательского типа используется механизм наследования от уже имеющихся классов представляющих исключения. Рекомендованной практикой является следующая:

- Если пользовательское исключение должно быть проверяемым, то стоит использовать к качестве суперкласса Exception
- Если пользовательское исключение должно быть непроверяемым, то стоит использовать в качестве суперкласса RuntimeException
- Имя типа исключения должно заканчиваться на Exception

Пользовательские исключения не генерируются автоматически средой выполнения. Создание и запуск обработки нужно производить вручную.

# Создание пользовательских исключений

```
public class NegativeValueException extends Exception{

     public NegativeValueException() {
          super();
     }

     public NegativeValueException(String message, Throwable cause) {
          super(message, cause);
     }

     public NegativeValueException(String message) {
          super(message);
     }

     public NegativeValueException(Throwable cause) {
          super(cause);
     }
}
```

Создание проверяемого пользовательского исключения (наследник Exception). Реализованы конструкторы для удобства использования в случае связанных исключений.

# Использование пользовательского исключения

```
File file = new File("price.txt");
Integer price = null;

try {
    Scanner sc = new Scanner(file);
    price = sc.nextInt();
    if (price < 0) {
        throw new NegativeValueException("Negative Value");
    }
} catch (IOException e) {
    System.out.println("File not found");
} catch (InputMismatchException e) {
    System.out.println("Error file format");
} catch (NegativeValueException e) {
    System.out.println(e.getMessage());
}

System.out.println("price = " + price);
```

Возбуждение пользовательского исключения

Обработка пользовательского исключения

# Задание для самостоятельной проработки. Основной уровень.

1) Создать класс Human.
   Поля:
   - String name (имя)
   - String lastName (фамилия)
   - Gender gender (пол. Реализовать с помощью Enum)

   Методы:
   - Стандартные (методы получения и установки, toString() и т. д.)
2) Создать класс Student как подкласс Human.
   Поля:
   - int id (номер зачетки)
   - String groupName (название группы где он учится)

   Методы:
   - Стандартные (методы получения и установки, toString() и т. д.)
3) Создать классы GroupOverflowException, StudentNotFoundException (наследники Exception) в качестве пользовательских исключений.
4) Создать класс Group
   Поля:
   - String groupName (название)
   - Student[] studens = new Student[10]; (массив из 10 студентов)

   Методы:
   - Стандартные (методы получения и установки, toString() и т. д.)
   - public void addStudent(Student student) throws GroupOverflowException (метод добавления студента в группу. В случае добавления 11 студента должно быть возбужденно пользовательское исключение)
   - public Student searchStudentByLastName(String lastName) throws StudentNotFoundException (метод поиска студента в группе. Если студент не найден должно быть возбужденно пользовательское исключение)
   - public boolean removeStudentByID(int id) (метод удаления студента по номеру зачетки, вернуть true если такой студент был и он был удален и false в противном случае)

# Задание для самостоятельной проработки. Продвинутый уровень.

1) Реализуйте метод для сортировки массива студентов по фамилии. Примените его в методе toString() класса Group что бы получить список студентов в алфавитном порядке.

# Список литературы

1) Герберт Шилдт Java 8. Полное руководство 9-е издание ISBN 978-5-8459-1918-2
2) https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/IllegalArgumentException.html
3) https://docs.oracle.com/javase/tutorial/essential/exceptions/chained.html