Содержание

4 Специализация: ООП и исключения **1**
4.1 В предыдущем разделе 1
4.2 В этом разделе 1
4.3 Перечисления 2
4.4 Внутренние и вложенные классы 4
4.5 Исключения 9

4. Специализация: ООП и исключения

4.1. В предыдущем разделе

Была рассмотрена реализация объектно-ориентированного программирования в Java. Рассмотрели классы и объекты, а также наследование, полиморфизм и инкапсуляцию. Дополнительно был освещён вопрос устройства памяти.

4.2. В этом разделе

В дополнение к предыдущему, будут разобраны такие понятия, как внутренние и вложенные классы; процессы создания, использования и расширения перечислений. Более детально будет разобрано понятие исключений и их тесная связь с многопоточностью в Java. Будут рассмотрены исключения с точки зрения ООП, процесс обработки исключений.

— Перечисление;
— Внутренний класс;
— Вложенный класс;
— Локальный класс;
— Исключение;
— Искл. (событие);
— Искл. (объект);
— Обработчик искл.;
— `throw`;
— Stacktrace;
— `try...catch`;
— `throws`;
— `finally`;
— Подавленное искл.;
— Многопоточность;

4.3. Перечисления

Кроме восьми примитивных типов данных и классов в Java есть специальный тип, выведенный на уровень синтаксиса языка – enum или перечисление. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую.

 Перечисление – это упоминание объектов, объединённых по какому-либо признаку

Перечисления – это специальные классы, содержащие внутри себя собственные статические экземпляры.

Листинг 1: Пример перечисления

```
enum Season { WINTER, SPRING, SUMMER, AUTUMN }
```

Перечисление, фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её. Переменная типа перечисления может хранить любой объект этого исключения.

Листинг 2: Переменная типа перечисления

```
Season current = Season.SPRING;
System.out.println(current);
```

Интересно также то, что вывод в терминал и запись в коде у исключений полностью совпадают, поэтому, в результате выполнения этого кода, в терминале будет выведено

```
SPRING
```

Каждое перечисление имеет статический метод values(), возвращающий массив всех констант перечисления.

Листинг 3: Вывод всех элементов перечисления

```
Season[] seasons = Season.values();
for (Season s : seasons) {
   System.out.printf("s ", s);
}
```

Именно в этом примере используется цикл foreach для прохода по массиву, для лаконичности записи. Данный цикл берёт последовательно каждый элемент перечисления, присваивает ему имя s точно также, как это сделано в примере выше, делает эту переменную доступной в теле цикла в рамках одной итерации, на следующей итерации будет взят следующий элемент, и так далее.

```
WINTER, SPRING, SUMMER, AUTUMN
```

Также, в перечисления встроен метод ordinal(), возвращающий порядковый номер определенной константы (нумерация начинается с 0). Обратите внимание на синтаксис,

метод можно вызвать только у конкретного экземпляра перечисления, а при попытке вызова у самого класса перечисления, ожидаемо компилятор выдаёт ошибку невозможности вызова нестатического метода из статического контекста.

Листинг 4: Метод `ordinal()`

```
System.out.println(current.ordinal());

System.out.println(Seasons.ordinal()); // ошибка
```

Такое поведение возможно, только если номер элемента хранится в самом объекте.

В перечислениях можно наблюдать очень примечательный пример инкапсуляции – неизвестно, хранятся ли на самом деле объекты перечисления в виде массива, но можем вызвать метод values() и получить массив всех элементов перечисления. Неизвестно, хранится ли в каждом объекте перечисления его номер, но можем вызвать его метод `ordinal()`.

Раз перечисление – это класс, возможно определять в нём поля, методы, конструкторы и прочее. Перечисление `Color` определяет приватное поле `code` для хранения кода цвета, а с помощью метода `getCode` он возвращается.

Листинг 5: Расширение объекта перечисления

```
public class Main {
   enum Color {
      RED("#FF0000"), BLUE("#0000FF"), GREEN("#00FF00");
      private String code;
      Color(String code) {
         this.code = code;
      }

      public String getCode(){ return code;}
   }

   public static void main(String[] args) {
      System.out.println(Color.RED.getCode());
      System.out.println(Color.GREEN.getCode());
   }
}
```

Через конструктор передается значение пользовательского поля.

Конструктор по умолчанию имеет модификатор `private`. Любой другой модификатор будет считаться ошибкой.

Создать константы перечисления с помощью конструктора возможно только внутри самого перечисления. И что косвенно намекает на то, что объекты перечисления это статические объекты внутри самого класса перечисления. Также важно, что механизм описания конструкторов класса работает по той же логике, что и обычные конструкторы, то есть, при описании собственного конструктора, конструктор по-умолчанию перестаёт создаваться автоматически. Таким образом, с объектами перечисления можно работать точно также, как с обычными объектами.

Листинг 6: Вывод значений пользовательского поля перечисления

```
for (Color c : Color.values()) {
   System.out.printf("s(s)\n", c, c.getCode());
}
```

```
RED(#FF0000)
BLUE(#0000FF)
GREEN(#00FF00)
```

4.3.1. Задания для самопроверки

1. Перечисления нужны, чтобы: 3
   (a) вести учёт созданных в программе объектов;
   (b) вести учёт классов в программе;
   (c) вести учёт схожих по смыслу явлений в программе;
2. Перечисление – это: 2
   (a) массив
   (b) класс
   (c) объект
3. каждый объект в перечислении – это: 3
   (a) статическое поле
   (b) статический метод
   (c) статический объект

4.4. Внутренние и вложенные классы

В Java есть возможность создавать классы внутри других классов, все такие классы разделены на следующие типы:

1. Non-static nested (inner) classes — нестатические вложенные (внутренние) классы;
   — локальные классы (local classes);
   — анонимные классы (anonymous classes);
2. Static nested classes — статические вложенные классы.

Для рассмотрения анонимных классов понадобятся дополнительные знания об интерфейсах, поэтому будут рассмотрены позднее.

4.4.1. Внутренние классы

Листинг 7: Вывод значений пользовательского поля перечисления

```
public class Orange {
   public void squeezeJuice() {
      System.out.println("Squeeze juice ...");
   }
   class Juice {
      public void flow() {
         System.out.println("Juice dripped ...");
      }
   }
}
```

Внутренние классы создаются внутри другого класса. Рассмотрим на примере апельсина с реализацией, как это предлагает официальная документация Oracle. В основной программе необходимо создать отдельно апельсин, отдельно его сок через интересную форму вызова конструктора, показанную в листинге 8, что позволяет работать как с апельсином, так и его соком по отдельности.

Листинг 8: Обычный апельсин Oracle

```
Orange orange = new Orange();
Orange.Juice juice = orange.new Juice();
orange.squeezeJuice();
juice.flow();
```

Важно помнить, что когда в жизни апельсин сдавливается, из него сам по себе течёт сок, а когда апельсин попадает к нам в программу он сразу снабжается соком.

Листинг 9: Необычный апельсин GeekBrains

```
public class Orange {
   private Juice juice;
   public Orange() {
      this.juice = new Juice();
   }
   public void squeezeJuice() {
      System.out.println("Squeeze juice ...");
      juice.flow();
   }
   private class Juice {
      public void flow() {
           System.out.println("Juice dripped ...");
      }
   }
}
```

Итак, был создан апельсин, при создании объекта апельсина у него сразу появляется сок. Ниже в классе описано потенциальное наличие у апельсина сока, как его части, поэтому внутри класса апельсин создан класс сока. При создании апельсина создали сок, так или иначе – самостоятельную единицу, обладающую своими свойствами и поведением, отличным от свойств и поведения апельсина, но неразрывно с ним связанную. При попытке выдавить сок у апельсина – объект сока сообщил о том, что начал течь

Листинг 10: Использование апельсина GeekBrains

```
Orange orange = new Orange();
orange.squeezeJuice();
```

Таким образом у каждого апельсина будет свой собственный сок, который возможно выжать, сдавив апельсин. В этом смысл внутренних классов не статического типа – нужные методы вызываются у нужных объектов.

Такая связь объектов и классов называется композицией. Существуют также ассоциация и агрегация.

Если класс полезен только для одного другого класса, то часто бывает удобно встроить его в этот класс и хранить их вместе. Использование внутренних классов увеличивает ин-

капсуляцию. Оба примера достаточно отличаются реализацией. Пример не из документации подразумевает «более сильную» инкапсуляцию, так как извне ко внутреннему классу доступ получить нельзя, поэтому создание объекта внутреннего класса происходит в конструкторе основного класса – в апельсине. С другой стороны, у примера из документации есть доступ извне ко внутреннему классу сока, но всё равно, только через основной класс апельсина, как и создать объект сока можно только через объект апельсина, то есть подчёркивается взаимодействие на уровне объектов.

Особенности внутренних классов:

— Внутренний объект не существует без внешнего. Это логично – для этого `Juice` был создан внутренним классом, чтобы в программе не появлялись апельсиновые соки из воздуха.
— Внутренний объект имеет доступ ко всему внешнему. Код внутреннего класса имеет доступ ко всем полям и методам экземпляра (и к статическим членам) окружающего класса, включая все члены, даже объявленные как `private`.
— Внешний объект не имеет доступа ко внутреннему без создания объекта. Это логично, так как экземпляров внутреннего класса может быть создано сколь угодно много, и к какому именно из них обращаться?
— У внутренних классов есть модификаторы доступа. Это влияет на то, где в программе возможно создавать экземпляры внутреннего класса. Единственное сохраняющееся требование — объект внешнего класса тоже обязательно должен существовать и быть видимым.
— Внутренний класс не может называться как внешний, однако, это правило не распространяется ни на поля, ни на методы;
— Во внутреннем классе нельзя иметь не-final статические поля. Статические поля, методы и классы являются конструкциями верхнего уровня, которые не связаны с конкретными объектами, в то время как каждый внутренний класс связан с экземпляром окружающего класса.
— Объект внутреннего класса нельзя создать в статическом методе «внешнего» класса. Это объясняется особенностями устройства внутренних классов. У внутреннего класса могут быть конструкторы с параметрами или только конструктор по умолчанию. Но независимо от этого, когда создаётся объект внутреннего класса, в него неявно передаётся ссылка на объект внешнего класса.
— Со внутренними классами работает наследование и полиморфизм.

4.4.2. Задания для самопроверки

1. Внутренний класс: 1
   (a) реализует композицию;
   (b) это служебный класс;
   (c) не требует объекта внешнего класса;
2. Инкапсуляция с использованием внутренних классов: 2
   (a) остаётся неизменной
   (b) увеличивается
   (c) уменьшается

3. Статические поля внутренних классов: 2
   (a) могут существовать
   (b) могут существовать только константными
   (c) не могут существовать

4.4.3. Локальные классы

Классы – это новый тип данных для программы, поэтому технически возможно создавать классы, а также описывать их, например, внутри методов. Это довольно редко используется но синтаксически язык позволяет это сделать. **Локальные классы** — это подвид внутренних классов. Однако, у локальных классов есть ряд важных особенностей и отличий от внутренних классов. Главное заключается в их объявлении.

Локальный класс объявляется только в блоке кода. Чаще всего — внутри какого-то метода внешнего класса.

Листинг 11: Пример локального класса

```
public class Animal {
   void performBehavior(boolean state) {
      class Brain {
         void sleep() {
            if (state)
               System.out.println("Sleeping");
            else
               System.out.println("Not sleeping");
         }
      }
      Brain brain = new Brain();
      brain.sleep();
   }
}
```

Например, некоторое животное, у которого устанавливается состояние спит оно или нет. Метод `performBehavior()` принимает на вход булево значение и определяет, спит ли животное. Мог возникнуть вопрос: зачем? Итоговое решение об архитектуре проекта всегда зависит от структуры, сложности и предназначения программы.

Особенности локальных классов:

— Локальный класс сохраняет доступ ко всем полям и методам внешнего класса, а также ко всем константам, объявленным в текущем блоке кода, то есть полям и аргументам метода объявленным как `final`. Начиная с JDK 1.8 локальный класс может обращаться к любым полям и аргументам метода объявленным в текущем блоке кода, даже если они не объявлены как `final`, но только в том случае если их значение не изменяется после инициализации.

— Локальный класс должен иметь свои внутренние копии всех локальных переменных, которые он использует (эти копии автоматически создаются компилятором). Единственный способ обеспечить идентичность значений локальной переменной и ее копии – объявить локальную переменную как `final`.

— Экземпляры локальных классов, как и экземпляры внутренних классов, имеют окружающий экземпляр, ссылка на который неявно передается всем конструкторам локальных классов. То есть, спервa должен быть создан экземпляр внешнего класса, а только затем экземпляр внутреннего класса.

4.4.4. Статические вложенные классы

При объявлении такого класса используется ключевое слово `static`. Для примера в классе котика и заменим метод `voice()` на статический класс.

Листинг 12: Статический вложенный класс

```
public class Cat {
   private String name;
   private String color;
   private int age;
   public Cat()
   public Cat(String name, String color, int age) {
      this.name = name;
      this.color = color;
      this.age = age;
   }

   static class Voice {
      private final int volume;
      public Voice(int volume) {
         this.volume = volume;
      }
      public void sayMur() {
         System.out.printf("A cat purrs with volume %d\n", volume);
      }
   }
}
```

То есть, такое мурчание котика может присутствовать без видимости и понимания, что именно за котик присутствует в данный момент. Также, добавлена возможность установить уровень громкости мурчанья.

Основное отличие статических и нестатических вложенных классов в том, что объект статического класса не хранит ссылку на конкретный экземпляр внешнего класса.

Без объекта внешнего класса объект внутреннего просто не мог существовать. Для статических вложенных классов это не так. Объект статического вложенного класса может существовать сам по себе. В этом плане статические классы более независимы, чем нестатические. Довольно важный момент заключается в том, что при создании такого объекта нужно указывать название внешнего класса,

Листинг 13: Использование статического класса

```
Cat.Voice voice = new Cat.Voice(100);
voice.sayMur();
```

Статический вложенный класс может обращаться только к статическим полям внешнего класса. При этом неважно, какой модификатор доступа имеет статическая переменная во внешнем классе.

Не следует путать объекты с переменными. Если речь идёт о статических переменных — да, статическая переменная класса существует в единственном экземпляре. Но применительно ко вложенному классу `static` означает лишь то, что его объекты не содержат ссылок на объекты внешнего класса.

4.4.5. Задания для самопроверки

1. Вложенный класс: 1
   (a) реализует композицию;
   (b) это локальный класс;
   (c) всегда публичный;
2. Статический вложенный классо бладает теми же свойствами, что: 2
   (a) константный метод
   (b) внутренний класс
   (c) статическое поле

4.5. Исключения

4.5.1. Понятие

Язык программирования – это, в первую очередь, набор инструментов. Например, есть художник. У художника есть набор всевозможных красок, кистей, холстов, карандашей, мольберт, ластик и прочие. Это всё его инструменты. Тоже самое для программиста. У программиста есть язык программирования, который предоставляет ему инструменты: циклы, условия, классы, функции, методы, ООП, фрейморки, библиотеки. Исключения – это один из инструментов. Исключения всегда следует рассматривать как ещё один инструмент для работы программиста.

Исключение – это отступление от общего правила, несоответствие обычному порядку вещей

В общем случае, возникновение исключительной ситуации, это ошибка в программе, но основным вопросом является следующий. Возникшая ошибка – это:

— ошибка в коде программы;
— ошибка в действиях пользователя;
— ошибка в аппаратной части компьютера?

4.5.2. Общие сведения

При возникновении ошибок создаётся объект класса «исключение», и в этот объект записывается какое-то максимальное количество информации о том, какая ошибка произошла, чтобы потом прочитать и понять, где проблема. Соответственно эти объекты возможно «ловить и обрабатывать».

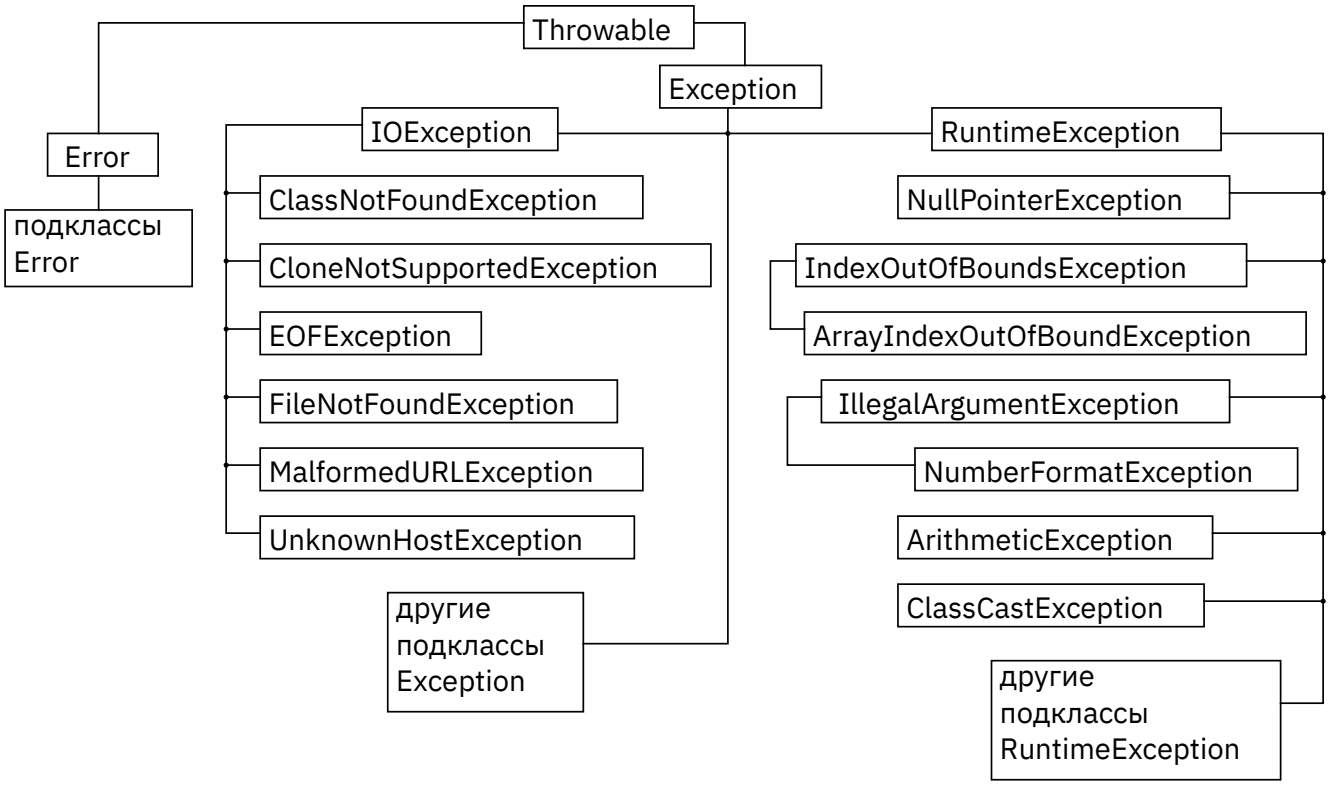

Рис. 1: Часть иерархии исключений

Все исключения наследуются от класса `Throwable` и могут быть как обязательные к обработке, так и необязательные. Есть ещё подкласс `Error`, но он больше относится к аппаратным сбоям или серьёзным алгоритмическим или архитектурным ошибкам, и на данном этапе интереса не представляет, потому что поймав, например, `OutOfMemoryError` средствами Java прямо в программе с ним ничего сделать невозможно, такие ошибки необходимо обрабатывать и не допускать в процессе разработки ПО.

Для изучения и примеров, воспользуемся двумя подклассами `Throwable` – `Exception` – `RuntimeException` и `IOException`.

 Все исключения (**checked**), кроме наследников `RuntimeException` (**unchecked**), необходимо обрабатывать.

Опишем на простом примере, один метод вызывает другой, второй вызывает третий и последний всё портит:

Листинг 14: Цепочка методов

```
1  private static int div0(int a, int b) {
2      return a / b;
3  }
4
5  private static int div1(int a, int b) {
6      return div0(a, b);
7  }
8
9  private static int div2(int a, int b) {
10     return div1(a, b);
11 }
```

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at ru.gb.jcore.Main.div0(Main.java:5)
    at ru.gb.jcore.Main.div1(Main.java:8)
    at ru.gb.jcore.Main.div2(Main.java:11)
    at ru.gb.jcore.Main.main(Main.java:14)
```

Рис. 2: Результат запуска цепочки методов (листинг 14)

`ArithmeticException` является наследником класса `RuntimeEcxeption` поэтому статический анализатор кода его не подчеркнул, и «ловить» его не обязательно.

При работе с исключениями часто можно встретить слова, похожие на сленг, но это не так. Чаще всего, то, что звучит как сленг – просто перевод ключевых слов языка, осуществляющих то или иное действие.

— try – (англ. пробовать) пробовать, пытаться;
— catch – (англ. ловить) ловить, поймать, хватать;
— throw – (англ. бросать) выбрасывать, бросать, кидать;
— NullPointerException – НПЕ, налпоинтер;
— и другие...

Если посмотреть на метод `div0(int a, int b)` с точки зрения программирования, он написан очень хорошо – алгоритм понятен, метод с единственной ответственностью, однако, из поставленной перед методом задачи очевидно, что он не может работать при всех возможных входных значениях. То есть если вторая переменная равна нулю, то это ошибка. Необходимо запретить пользователю передавать в качестве делителя ноль. Самое простое – ничего не делать, но в программе на языке Java так нельзя, если мы объявили, что метод имеет возвращающее значение, он обязан что-то вернуть.

Листинг 15: Ошибка – нельзя ничего не возвращать

```
1  private static int div0(int a, int b) {
2      if (b != 0)
3          return a / b;
4      return ???; // ошибка
5  }
```

А что вернуть, неизвестно, ведь от метода ожидается результат деления. Поэтому, возможно руками сделать проверку `(b == 0f)` и «выбросить» пользователю так называемый **объект исключения** с текстом ошибки, объясняющим произошедшее, а иначе вернём `a / b`.

Листинг 16: Цепочка методов

```
1  private static int div0(int a, int b) {
2      if (b == 0f)
3          throw new RuntimeException("parameter error");
4      return a / b;
5  }
```

Следовательно, если делитель не равен нулю произойдёт обычное деление, а если равен – будет «выброшено» исключение.

```
Exception in thread "main" java.lang.RuntimeException Create breakpoint : parameter error
    at ru.gb.jcore.Main.div0(Main.java:5)
    at ru.gb.jcore.Main.div1(Main.java:9)
    at ru.gb.jcore.Main.div2(Main.java:12)
    at ru.gb.jcore.Main.main(Main.java:15)
```

Рис. 3: Исключение, выброшенное «на наших условиях» (листинг 16)

Очевидно, что ключевое слово `new` вызывает конструктор, нового объекта какого-то класса, в который передаётся какой-то параметр, в данном конкретном случае это строка с сообщением.

4.5.3. Объект исключения

Ключевое слово `throw` заставляет созданный объект исключения начать свой путь по родительским методам, пока этот объект не встретится с каким-то обработчиком. В данном конкретном случае – это обработчик виртуальной машины (по-умолчанию), который в специальный поток error выводит так называемый stacktrace, и завершает дальнейшее выполнение метода (технически, всего потока целиком).

Рис. 4: Принципиальная схема работы приложения

Все программы в Java всегда многопоточны. На старте программы запускаются так называемые потоки, которые работают псевдопараллельно и предназначены каждый для решения своих собственных задач, например, это основной поток, поток сборки мусора, поток обработчика ошибок, потоки графического интерфейса. Основная задача этих потоков – делать своё дело и иногда обмениваться информацией.

В **stacktrace**, содержащийся в объекте исключения, кладётся максимальное количество информации о типе исключения, его сообщении, иерархии методов, вызовы которых привели к исключительной ситуации.

Важно научиться читать stacktrace на как можно более раннем этапе изучения программирования.

Итак стектрейс. В стектрейсе на рис. 3 видно, что исключение было создано в потоке main, и является объектом класса `RuntimeException`, сообщение также было предусмотрительно приложено автором кода. Важно понять, что исключение – это объект класса.

Далее, можно просто читать последовательно строку за строкой – в каком методе создался этот объект, на какой строке, в каком классе. Далее видно, какой код вызвал этот метод, на какой строке, в каком классе.

Если не написать явного выбрасывания никакого исключения, оно всё равно будет выброшено. Это общее поведение исключения. Оно где-то случается, прекращает выполнение текущего метода, и начинает лететь по стеку вызовов вверх. Возможно даже долетит до обработчика по-умолчанию. Некоторые исключения создаются в коде явно, некоторые самой Java, они вполне стандартные, например выход за пределы массива, деление на ноль, и классический `NullPointer`.

Создадим экземпляр класса исключения внутри метода, вызываемого из `main`:

Листинг 17: Инициализация объекта исключения

```
1 RuntimeException e = new RuntimeException();
```

Если оставить программу в таком виде и запустить, то ничего не произойдёт, исключение нужно выкинуть (активировать, возбудить, сгенерировать). Для этого есть ключевое слово

Листинг 18: Выбрасывание объекта исключения

```
1 throw e;
```

Компилятор ошибок не обнаружил и всё пропустил, а интерпретатор наткнулся на класс исключения, и написал в консоль, что в основном потоке программы возникло исключение в пакете в классе на такой-то строке. По стэктрейсу возможно проследить что откуда вызвалось и как программа дошла до исключительной ситуации. Возможно также наследоваться от какого-то исключения и создать свой класс исключений.

Исключения, наследники `RuntimeException`, являются **Unchecked**, то есть не обязательные для обработки на этапе написания кода. *Все остальные* `Throwable` – обязательные для обработки, статический анализатор кода не просто их выделяет, а обязывает их обрабатывать на этапе написания кода. И просто не скомпилирует проект если в коде есть необработанные исключения, также известные как **Checked**.

4.5.4. Обработка

Первое, и самое важное, что нужно понять – почему что-то пошло (или пойдёт) «не так», поэтому не пытайтесь что-то ловить, пока не поймёте что именно произошло, от этого понимания будет зависеть *способ* ловли.

Исключение ловится двухсекционным оператором **`try...catch`**, а именно, его первой секцией `try`. Это секция, в которой предполагается возникновение исключения, и предполагается, что его возможно обработать. А в секции `catch` пишется имя класса исключения,

которое будет поймано ловим, и имя объекта (идентификатор), через который внутри секции можно к пойманному объекту обращаться. Секция `catch` ловит указанное исключение и *всех его наследников*.

Рекомендуется писать максимально узко направленные секции `catch`, потому что надо стараться досконально знать как работает программа, и какие исключения она может выбрасывать. Также, потому что разные исключения могут по-разному обрабатываться.

Секций `catch` может быть любое количество. Как только объект исключения обработан, он уничтожается и в следующие `catch` не попадает. Однако, объект возможно явно отправить на обработчик «выше», ключевым словом `throw` (чаще всего, используется `RuntimeException` с конструктором копирования).

Когда какой-то метод выбрасывает исключение у разработчика есть два основных пути:

— обязанность вынести объявление этого исключения в сигнатуру метода, что будет говорить тем, кто его вызывает о том, что в методе может возникнуть исключение;
— исключение необходимо непосредственно в методе обработать, иначе ничего не скомпилируется.

В случае, если объявление исключения выносится в сигнатуру, вызывающий метод должен обработать это исключение точно таким-же образом – либо в вызове, либо вынести в сигнатуру. Исключением из этого правила является класс `RuntimeException`. Все его наследники, включая его самого, обрабатывать не обязательно. Обычно, уже по названию понятно что случилось, и, помимо говорящих названий, там содержится много информации, например, номер строки, вызвавшей исключительную ситуацию.

Общее правило работы с исключениями одно – если исключение штатное – его надо сразу обработать, если нет – надо дождаться, пока программа упадёт.

Общий вид оператора try...catch можно описать следующим образом:

```
try {
    метод, выбрасывающий исключение
} catch (имя класса исключения и идентификатор) {
    команды, обрабатывающие исключение
}
```

Если произошло исключение, объект исключения попадает в `catch`, и управление ходом выполнения программы попадает в эту секцию. Чаще всего, здесь содержится код, помогающей программе не завершиться. Очень часто в процессе разработки нужно сделать так, чтобы в процессе выполнения что-то конкретное об исключении выводилось на экран, для этого у экземпляра есть метод `getMessage()`.

Листинг 19: Получение сообщения из объекта исключения

```
1 System.out.println(e.getMessage());
```

Ещё чаще бывает, что выполнение программы после выбрасывания исключения не имеет смысла и нужно, чтобы программа завершилась. В этом случае принято выбрасывать новое RuntimeException, передав в него экземпляр пойманного исключения, используя конструктор копирования.

Листинг 20: «Проброс» исключения на основе пойманного

```
try {
   // ...
} catch(Exception e) {
   throw new runtimeexception(e);
}
```

Второй вариант обработки исключений – в сигнатуре метода пишется

Листинг 21: Обработка исключений в сигнатуре

```
throws IOException,
```

и, через запятую, все остальные возможные исключения этого метода. После этого, с ним не будет проблем исполнения, но у метода который его вызовет – появилась необходимость обработать все checked исключения вызываемого. И так далее наверх.

4.5.5. Пример

Для примера обработки исключений, возникающих на разных этапах работы приложения (жизненного цикла объекта) предлагается описать класс (листинг 22), бизнес логика которого подразумевает создание, чтение некоторой информации, например, как если бы нужно было прочитать байт из файла, и закрытие потока чтения, то есть возврат файла обратно под управление ОС.

Листинг 22: Экспериментальный класс

```
public class TestStream {
  TestStream() {
    System.out.println("constructor");
  }
  int read() {
    System.out.println("read");
    return 1;
  }
  public void close() {
    System.out.println("close");
  }
}
```

То есть, способ работы с объектом данного класса (полностью без ошибок и других нештатных ситуаций) будет иметь следующий вид

Листинг 23: Работа в штатном режиме

```
TestStream stream = new TestStream();
int a = stream.read()
stream.close()
```

Для примера, внутри метода чтения создаётся `FileInputStream` который может генерировать обязательный к проверке на этапе написания кода `FileNotFoundException`,

который является наследником IOException, который, в свою очередь, наследуется от Exception.

Возникает два варианта: либо обернуть в try...catch, либо совершенно непонятно, как должна обрабатываться данная исключительная ситуация, и обработать её должна сторона, которая вызывает метод чтения, в таком случае пишется, что метод может выбрасывать исключения. И тогда TestStream компилируется без проблем, а вот main скомпилироваться уже не может. В нём нужно оборачивать в try...catch.

Листинг 24: Метод чтения

```
  int read() {
    FileInputStream s = new FileInputStream("file.txt");
    System.out.println("read");
    return 1;
  }
}
```

Листинг 25: Обработка исключения

```
try {
   TestStream stream = new TestStream();
   int a = stream.read()
   stream.close()
} catch (FileNotFoundException e) {
  e.printStackTrace();
}
```

Важный момент. Задачи бывают разные. Исключения – это инструмент, который нетривиально работает. Важно при написании кода понять, возникающая исключительная ситуация – штатная, или нештатная. В большинстве случаев – ситуации нештатные, поэтому надо «уронить» приложение и разбираться с тем, что именно произошло. Допустим, для вашего приложения обязательно какой-то файл должен быть, без него дальше нет смысла продолжать. Что делать, если его нет? Ситуация явно нештатная. Самое плохое, что можно сделать – ничего не делать. Это самое страшное, когда программа повела себя как-то не так, а ни мы, разработчики, ни пользователь об этом даже не узнали. Допустим, мы хотим прочитать файл, вывести в консоль, но мы в обработчике исключения просто выводим стектрейс куда-то, какому-то разработчику в среду разработки, и наши действительно важные действия не выполнились. Надо завершать работу приложения. Как завершать? `throw new RuntimeException(e)`. Крайне редко случаются ситуации, когда у исключения достаточно распечатать стектрейс.

Потоки ввода-вывода всегда нужно закрывать. Предположим, что в тестовом потоке открылся файл, из него что-то прочитано, потом метод завершился с исключением, а файл остался незакрытым, ресурсы заняты. Дописав класс TestStream при работе с ним, возвращаем из read единицу и логируем, что всё прочитали в main.

Листинг 26: Экспериментальный класс

```
public class TestStream {
```

```
  TestStream() {
    System.out.println("constructor");
  }
  int read() throws IOException {
    throw new IOException("read except");
    System.out.println("read");
    return 1;
  }
  public void close() {
    System.out.println("close");
  }
}
```

Далее представим, что в методе `read` что-то пошло не так, выбрасываем исключение, и видим в консоли, что поток создан, произошло исключение, конец программы. Очевидно, поток не закрылся. Что делать?

Делать секцию `finally`. Секция `finally` будет выполнена в любом случае, не важно, будет ли поймано секциями `catch` какое-то исключение, или нет. Возникает небольшая проблема видимости, объявление идентификатора тестового потока необходимо вынести за пределы секции `try`.

Теперь немного неприятностей. Написанный блок `finally`, вроде решает проблему с закрытием потока. А как быть, если исключение возникло при создании этого потока, в конструкторе?

Листинг 27: Проблема в конструкторе

```
public class TestStream {
  TestStream() throws IOException {
    throw new IOException("construct except");
    System.out.println("constructor");
  }
  int read() throws IOException {
    throw new IOException("read except");
    System.out.println("read");
    return 1;
  }
  public void close() {
    System.out.println("close");
  }
}
```

Метод закрытия будет пытаться выполниться от ссылки на `null`. Недопустимо.

При возникновении в конструкторе потока `IOException` - получим `NullPointerException` в блоке `finally`.

Очевидное решение – поставить в секции `finally` условие, и если поток не равен `null`, закрывать. Это точно сработает. Меняем тактику.

Листинг 28: Проблема при закрытии

```
public class TestStream {
  TestStream() throws IOException {
    throw new IOException("construct except");
    System.out.println("constructor");
  }
  int read() throws IOException {
```

```
    throw new IOException("read except");
    System.out.println("read");
    return 1;
  }
  public void close() throws IOException {
    throw new IOException("close except");
    System.out.println("close");
  }
}
```

Конструктор отрабатывает нормально. Метод чтения всё ещё генерирует исключение, но и в методе закрытия что-то пошло не так, и вылетело исключение. Нужно оборачивать в `try...catch`. Итоговый код, работающий с классом будет иметь следующий вид.

Листинг 29: Обработка исключений, насколько это возможно

```
TestStream stream = null;
try {
    stream = new TestStream();
    int a = stream.read()
    stream.close()
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        stream.close();
    } catch (NullPointerException e) {
        e.printStackTrace();
    }
}
```

Но и тут возможно наткнуться на неприятность. Допустим, что необходимо в любом случае ронять приложение.

Листинг 30: Проблемы подавленных исключений

```
  // ...
catch (IOException e) {
    throw new RuntimeException(e);
} finally {
    try {
        stream.close();
    } catch (NullPointerException e) {
        e.printStackTrace();
    }
}
```

Тогда если `try` поймал исключение, и выкинул его, потом `finally` всё равно выполнится, и второе исключение перекроет (подавит) первое, никто его не увидит. Хотя по логике, первое для работы важнее. Так было до Java 1.8.

4.5.6. try-with-resources block

Начиная с версии Java 1.8 разработчику предоставляется механизм **`try-с-ресурсами`**. Поток – это ресурс, абстрактное понятие. Выражаясь строго формально, разработчик должен *реализовать интерфейс* `Closeable`. В этом интерфейсе содержится всего один метод `close()`, который умеет бросать `IOException`. В классе тестового потока нужно всего лишь

переопределить этот метод данного интерфейса.

Листинг 31: Реализация интерфейса закрытия потока

```
public class TestStream implements Closeable {
   // ...

   @Override
   public void close() throws IOException {
      throw new IOException("close except");
      System.out.println("close");
   }
}
```

Все потоки начиная с Java 1.8 реализуют интерфейс `Closeable`. Работа с такими классами имеет лаконичный вид

Листинг 32: Реализация блока try-with-resources

```
try (TestStream stream = new TestStream()) {
   int a = stream.read();
} catch (IOException e) {
   throw new RuntimeException(e)
}
```

В данном коде не нужно закрывать поток явно, это будет сделано автоматически в следствие реализации интерфейса. Если ломается метод `read()`, то try-с-ресурсами всё равно корректно закроет поток. При сломанном методе закрытия и сломанном методе чтения одновременно, JVM запишет наверх основное исключение, но и выведет «подавленное» исключение, вторичное в стектрейс. Рекомендуется по возможности всегда использовать try-с-ресурсами.

4.5.7. Наследование и полиморфизм исключений

Наследование и полиморфизм для исключений – тема не очень большая и не сложная, потому что ничего нового в информацию про классы и объекты исключения не привносят. Застрять внимание на объектно-ориентированном программировании в исключениях не целесообразно, потому что исключения это *тоже классы* и те исключения, которые используются в программе – уже какие-то наследники других исключений.

Генерируются и выбрасываются *объекты исключений*, единственное, что важно упомянуть это то, что подсистема исключений работает не тривиально. Но разработчик может создавать собственные исключения с собственными смыслами и сообщениями и точно также их выбрасывать вместо стандартных. Наследоваться возможно от любых исключений, единственное что важно, это то, что не рекомендуется наследоваться от классов `Throwable` и `Error`, когда описываете исключение.

Механика checked и unchecked исключений сохраняется при наследовании, поэтому создав наследник `RuntimeException` вы получаете не проверяемые на этапе написания кода исключения.

Практическое задание

1. напишите два наследника класса Exception: ошибка преобразования строки и ошибка преобразования столбца
2. разработайте исключения-наследники так, чтобы они информировали пользователя в формате ожидание/реальность
3. для проверки напишите программу, преобразующую квадратный массив целых чисел 5x5 в сумму чисел в этом массиве, при этом, программа должна выбросить исключение, если строк или столбцов в исходном массиве окажется не 5.

Термины, определения и сокращения

`finally`	часть оператора `try...catch`, выполняющаяся вне зависимости от того, возникло ли исключение в секции `try` и было ли оно обработано в секции `catch`.
`throws`	ключевое слово, определяющее обработку исключения в методе. Фактически, это предупреждение для вызывающего о возможном исключении в методе.
`throw`	оператор, активирующий (выбрасывающий) объект исключения.
`try...catch`	двухсекционный оператор языка Java, позволяющий «безопасно» выполнить код, содержащий исключение, поймать и обработать возникшее исключение.
Stacktrace	часть объекта исключения, содержащая максимальное количество информации об иерархии методов, вызовы которых привели к исключительной ситуации.
Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Искл. (объект)	созданный программным кодом или JRE объект, передаваемый от потока, в котором произошло исключительное событие, обработчику исключений
Искл. (событие)	поведение потока исполнения (например, программы), пользователя или аппаратурного окружения, приведшее к исключению. При возникновении исключения создаётся объект исключения и работа потока останавливается.
Исключение	это отступление от общего правила, несоответствие обычному порядку вещей.
Локальный класс	класс, объявленный внутри минимального блока кода другого класса, чаще всего, метода.
Многопоточность	одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU – central processing unit). Каждый поток работает параллельно и имеет свою собственную выделенную стековую память.
Обработчик искл.	объект, работающий в потоке error или его наследники, способный ловить объекты исключений и совершать с ними манипуляции, например, выводить информацию об объекте исключения в консоль.

Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.
Подавленное искл.	исключение, возникшее *первым* в ситуации, когда в одном операторе `try...catch...finally` выброшены исключения как в `try`, так и в `finally`.