Содержание

4 Специализация: ООП и исключения **1**
4.1 В предыдущем разделе 1
4.2 В этом разделе 1
4.3 Перечисления 1
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;
— 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

```
1 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

```
1 Season current = Season.SPRING;
2 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

```
1 Season[] seasons = Season.values();
2 for (Season s : seasons) {
3     System.out.printf("s ", s);
4 }
```

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()`

```
1 System.out.println(current.ordinal());
2
3 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 инкапсуляцию. 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, сперва 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;
   }

```

```
12     static class Voice {
13        private final int volume;
14        public Voice(int volume) {
15           this.volume = volume;
16        }
17        public void sayMur() {
18           System.out.printf("A cat purrs with volume %d\n", volume);
19        }
20     }
21  }
```

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

```
1  Cat.Voice voice = new Cat.Voice(100);
2  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

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, фреймворки, 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?

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».

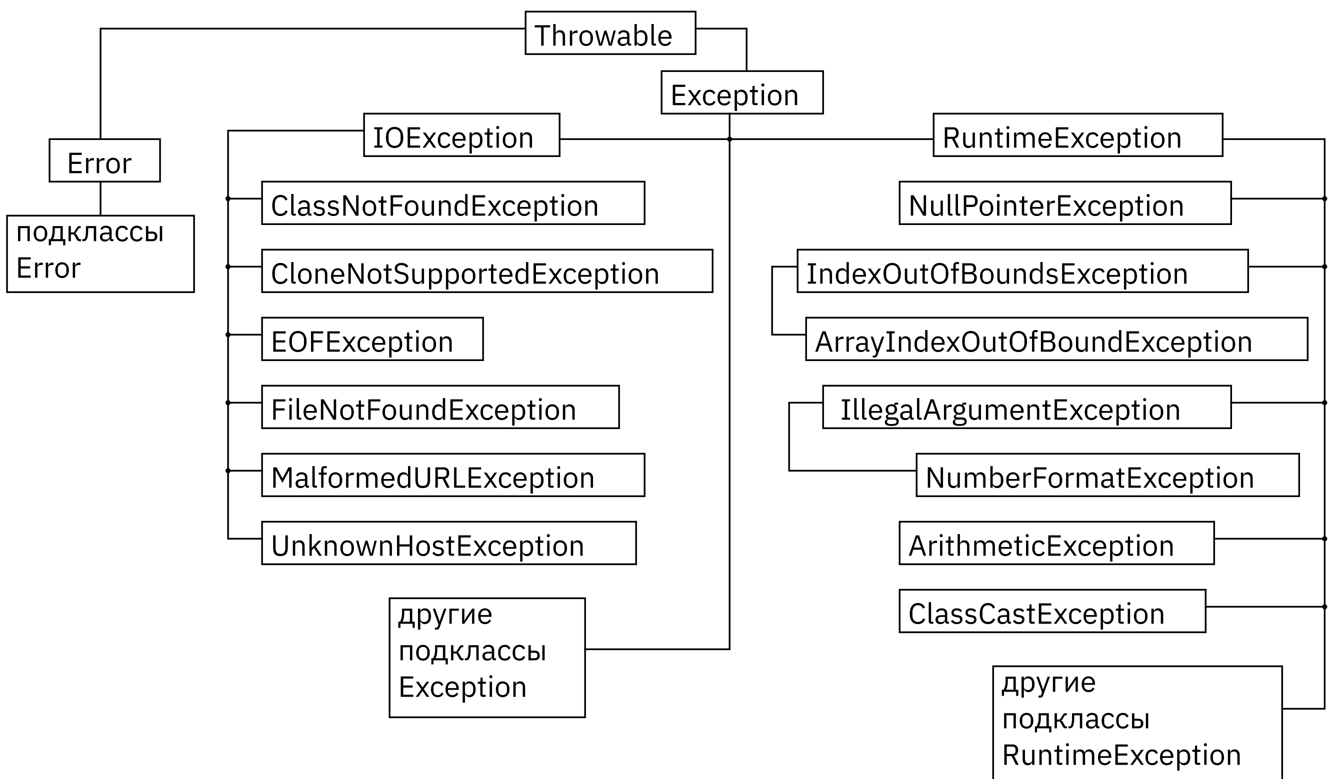

Рис. 1: Часть иерархии исключений

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 аппа-

ратным 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, кроме наследников `RuntimeException` необходимо обрабатывать.

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 5х5 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

Вложенный класс	статический класс, объявленный внутри другого класса.
Внутренний класс	нестатический класс, объявленный внутри другого класса.
Исключение	
Многопоточность	
Перечисление	это упоминание объектов, объединённых по какому-либо признаку. Фактически, представляет новый тип данных, поэтому возможно определить переменную данного типа и использовать её.