**Что такое Enums**

*"Чай, кофе, капучино?"*

В жизни часто у нас есть выбор из ограниченного количества вариантов. Это можно сравнить с "меню", из которого мы можем выбирать.

Именно **Enum**позволяет создать такое "меню" - ограниченый набор значений, из которого пользователь может выбирать то, что ему нужно.

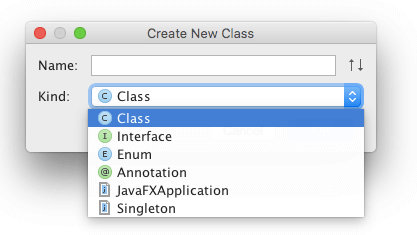
Как представить такое "меню" в коде? Как говорится, лучше один раз увидеть:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public enum myEnum {      WINTER,      SUMMER,      SPRING,      FALL;  } |

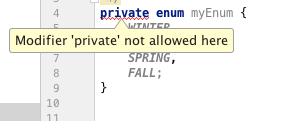
Вот мы и создали наш первый **Enum**! Выглядит просто, правда?

Давайте обратим внимание на ключевые моменты:

* **Enum**- это отдельная структура. Он может находится в отдельном файле, а может быть частью класса. Но при этом enum не обязательно должен лежать в каком-либо классе. При таком подходе мы как бы создаем еще один класс, только вместо слова "class" пишем "enum". Посмотрите - даже в IDE они выделены отдельно:

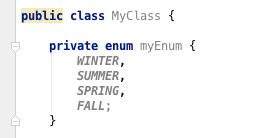


* У **enum-ов**есть модификатор доступа. Если Ваш **enum** не лежит внутри какого-нибудь класса, он должен быть объявлен **public**. Если сделать его **private**, Вы получите ошибку:

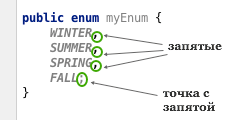


И правильно - как тогда им пользоваться?

Если же **enum** будет "внутри" класса, он может быть объявлен **private**:



* Все "опции" (а на деле - **объекты**), которые лежат в **enum**-е, принято писать большими буквами. Если написать их маленькими ничего плохого не случится. Мы рекомендуем писать большими буквами, поскольку это общепринятая практика.
* Обратите внимание, что объекты, которые лежат в **enum**-е, пишутся через запятую:



В конце мы написали точку с запятой.

Концепция понятна - как его применить? Как обратиться к этому **enum**-у? Давайте посмотрим на примере.

**Пример 1**

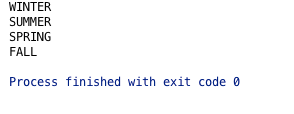
Мы уже описывали один **enum**выше. Давайте создадим такой же, только для наглядности назовем его не **myEnum**, а **Seasons**("времена года"):

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public enum Seasons {      WINTER,      SUMMER,      SPRING,      FALL;  } |

Теперь создадим такой **main**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Test {      public static void main(String[] args) {          System.out.println(Seasons.WINTER);          System.out.println(Seasons.SUMMER);          System.out.println(Seasons.SPRING);          System.out.println(Seasons.FALL);      }  } |

Получим:



Отлично! Давайте посмотрим на код внимательнее:

* Когда мы вызываем объекты из **enum**-а, мы сначала пишем название самого **enum**-а, а потом через точку - название объекта:

**Seasons.***WINTER*

Ну а что, если в **enum**-е много объектов? Опять руками писать?

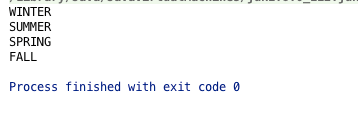
Давайте оптимизируем с помощью **for each**.

**Пример 2**

Если Вы не знаете, что такое **for each**, прочитайте сначала эту статью:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Test {      public static void main(String[] args) {          for (Seasons s : Seasons.values()) {              System.out.println(s);          }      }  } |

Запустим этот код - получим:



Что это значит?

* У **enum**-ов есть следующий метод - **values()**. Он возвращает массив значений, которые хранятся в данном **enum**-е.
* С помощью оператора **for each**мы проходимся по этому массиву и выводим его значения.

**Пример 3**

Еще **enum**-ы очень удобно применять в операторах **if** или **switch-case (**[**Урок 16: Условный оператор if в Java. Оператор switch**](https://vertex-academy.com/tutorials/ru/vetvlenie-v-java/)**)**. Можно сказать, что enum-ы созданы для этого! Давайте попробуем.

Давайте считать, что у нас есть такой же **enum**как и в прошлый раз - **enum Seasons**. Запустим используя такой **main**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Test {      public static void main(String[] args) {          Seasons arg = Seasons.FALL;            switch (arg)          {              case WINTER:              System.out.println("It's winter! Christmas time!"); break;              case SUMMER:                  System.out.println("It's summer! Let's go to the beach!"); break;              case SPRING:                  System.out.println("It's spring! Easter is coming!"); break;              case FALL:                  System.out.println("It's fall! Helloween is coming!"); break;          }      }  } |

Если запустить, в консоли увидим:

https://vertex-academy.com/tutorials/wp-content/uploads/2018/05/Capture-d%E2%80%99%C3%A9cran-2018-05-18-%C3%A0-19.16.17.png

Комментарии:

* Как видите, мы создали объект типа **enum**:

Seasons arg = Seasons.FALL;

* Потом мы создали оператор **switch**, который печатает сообщение в зависимости от времени года. Обратите внимание - в **case** мы не должны писать полный путь (например, Seasons.FALL), а только название объекта - например, FALL:

case WINTER:

 Абстрактный класс **Enum**появился в 5-й версии Java. Теперь, благодаря абстрактному классу Enum мы можем быстро и легко создавать перечисления без написания повторяющегося кода.

В этой статье мы рассмотрим на конкретных примерах следующее:

**1. К Enum можно применять методы:**

* name() - возвращает имя
* ordinal() - возвращает порядковый номер
* equals()
* hashCode()
* toString()
* finalize()
* clone()
* values()
* valueOf()

**2. Enum реализовывает интерфейс Comparable**

**3. Enum реализовывает интерфейс Serializable**

Иииииии сразу в бой  Создадим простой **enum Color**

|  |  |
| --- | --- |
| 1  2  3 | enum Color {      RED, GREEN, BLUE  } |

Методы name() и ordinal()

У каждого **enum**есть имя и порядковый номер. Получить их можно с помощью методов **name()** и **ordinal()**

|  |  |
| --- | --- |
| 1  2 | System.out.println(Color.RED.name()); //output: RED  System.out.println(Color.RED.ordinal()); //output: 0 |

Посмотрим как это реализовано в классе **Enum**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public abstract class Enum<E extends java.lang.Enum<E>>          implements Comparable<E>, Serializable {        private final String name;        public final String name() {          return name;      }        private final int ordinal;        public final int ordinal() {          return ordinal;      }        protected Enum(String name, int ordinal) {          this.name = name;          this.ordinal = ordinal;      }  ....  } |

Пояснения:

* Конструктор невидим для разработчиков, но используется самой Java для корректной работы перечислений.

Возникает вопрос, а где же ключевое слово **extends**возле декларации перечисления **Color?** Все дело в ключевом слове **enum**, именно оно даёт понять программе, что вы хотите ***не просто класс, а именно перечисление***, что и дает Вам уйму возможностей.

Методы equals(), hashcode(), toString(), finalize() и clone()

**Enum**переопределяет базовые методы класса **Object.**Так что их можно использовать сразу же в наших перечислениях.

**Пример 1: equals()**

|  |  |
| --- | --- |
| 1  2  3  4  5 | boolean isEqualToItself = Color.RED.equals(Color.RED);  boolean isEqualToDifferentColor = Color.RED.equals(Color.GREEN);    System.out.println(isEqualToItself); //output: true  System.out.println(isEqualToDifferentColor);//output: false |

**Пример 2:  hashCode()**

|  |  |
| --- | --- |
| 1  2  3  4  5 | int hashOfRed = Color.RED.hashCode();  int hashOfGreen = Color.GREEN.hashCode();    System.out.println(hashOfRed); //output would be different every time: 366712642  System.out.println(hashOfGreen); //output would be different every time: 1829164700 |

Поскольку мы использовали hashCode(), каждый раз будет выводиться разное значение, сгенерированное автоматически. Когда мы запускали код, получили числа 366712642 и 1829164700. Вы наверняка получите другие числа.

**Пример 3: toString()**

|  |  |
| --- | --- |
| 1  2 | String red = Color.RED.toString();  System.out.println(red); //output: RED |

Посмотрим как они реализованы в классе **Enum**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public String toString() {      return name;  }    public final boolean equals(Object other) {      return this==other;  }    public final int hashCode() {      return super.hashCode();  } |

Пояснения:

* метод **toString()** возвращает имя значения перечисления. Назвали значение **WHITE**, это же значение и получим при вызове **toString()**или **name();**
* метод **equals()**сравнивает значения перечислений по ссылкам. Почему? Потому, что значения в перечислениях являются константными (уникальными), существует всего один экземпляр цвета **RED**, один цвета **GREEN** и один **BLUE,**значит ссылка на этот экземпляр будет всего одна, значит их можно сравнивать с помощью **==.**Вы можете сами убедиться в этом, написав **Color.RED** == **Color.RED** или **Color.GREEN** == **COLOR.BLUE**;
* метод **hashCode()**использует стандартную реализацию из класса **Object.**

**Пример 4: finalize(), clone()**

|  |  |
| --- | --- |
| 1  2  3  4  5 | protected final void finalize() { }    protected final Object clone() throws CloneNotSupportedException {      throw new CloneNotSupportedException();  } |

Пояснения:

* метод **finalize()**пустой, а это значит, что не нужно закрывать "ресурсы" перед сборщиком мусора.  Мы говорим о тех "ресурсах", которые используются в try-with-resources. Да и вообще метод **finalize()**пережиток прошлых лет и в **Java 9**данный метод уже помечен как **@Deprecated**(устаревший метод, который уберут в последующих реализациях);
* метод **clone()**мы можем вызвать только внутри самого перечисления т.к. он помечен ключевым словом **protected**. Но даже если мы попытаемся сделать это, то ничего мы не получим, кроме **CloneNotSupportedException.**Нужно это для того чтобы нельзя было создать несколько экземпляров одного и того же перечисления. Ведь в реальной жизни у нас нет двух цифр "1", нет двух значений скорости света, так и с перечислениями.

**Метод values() - позволяет**получить массив всех значений Enum

Сделать это можно с помощью метода **values()**

|  |  |
| --- | --- |
| 1  2 | Color[] colors = Color.values();  System.out.println(Arrays.toString(colors)); //output: [RED, GREEN, BLUE] |

Посмотрим на реализацию метода **values()**в классе **Enum**

|  |  |
| --- | --- |
| 1 | public static E[] values(); |

Пояснения:

* полной реализации метода в классе **Enum**нет, так как он синтетический (искусственно добавляется во время компиляции). Из документации узнаем, что метод просто возвращает массив всех значений перечисления в порядке их объявления.

Метод valueOf() - позволяет получить значения перечисления по его строковому представлению

Для получения значение перечисления по его строковому представлению у **Enums**есть метод **valueOf().**Посмотрим на его использование

|  |  |
| --- | --- |
| 1 | System.out.println(Color.valueOf("RED").ordinal()); //output: 0 |

Если же такого значения в перечислении нет, то мы получим **IllegalStateException**

|  |  |
| --- | --- |
| 1 | Color.valueOf("BLACK"); //output: java.lang.IllegalArgumentException: No enum constant Color.BLACK |

Посмотрим на реализацию метода **valueOf()**в классе **Enum**

|  |  |
| --- | --- |
| 1 | public static E valueOf(String name); |

Пояснения:

* как и в случае с **values()**этот метод тоже синтетический и поэтому полной реализации данного метода в классе **Enum**нет. В официальной документации написано, что метод просто возвращает значение перечисления по его строковому представлению. Проверка строгая, поэтому никаких пробелов вначале, в конце или между буквами не должно быть.

Есть еще один способ получить значение перечисления

|  |  |
| --- | --- |
| 1 | System.out.println(Enum.valueOf(Color.class, "BLUE").ordinal()); //output: 2 |

но он является менее распространённым.

Рассмотрим его подробнее:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name) {      T result = enumType.enumConstantDirectory().get(name);      if (result != null)          return result;      if (name == null)          throw new NullPointerException("Name is null");      throw new IllegalArgumentException(          "No enum constant " + enumType.getCanonicalName() + "." + name);  } |

Пояснения:

* метод **enumConstantDirectory()**возвращает **Map,**где ключ - строковое значение перечисления, а значение - реальное значение перечисления;
* если мы нашли нужное значение в **Map -**возвращаем его;
* если мы передали **null** в метод **valueOf()** - обратно получим **NullPointerException**;
* если же значения, которое мы указали в **valueOf()** нет - мы получим **IllegalArgumentException**.

Enum реализовывает интерфейс Comparable

Что такое интерфейс Comparable мы рассматривали в статье ["Интерфейсы Comparable и Comparator"](https://vertex-academy.com/tutorials/ru/interfejsy-comparable-comparator-java/).

Зачем же Enum реализовывает интерфейс Comparable? Сделано это для того, чтобы перечисления можно было сравнивать друг с другом при сортировке. При этом сравнение происходит по **ordinal()**перечисления. Вспомним порядок объявления значений в **Color**

|  |  |
| --- | --- |
| 1  2  3 | enum Color {      RED, GREEN, BLUE  } |

Теперь посмотрим на сравнение элементов перечисления с помощью метода **compareTo()**

|  |  |
| --- | --- |
| 1  2  3  4 | System.out.println(Color.GREEN.compareTo(Color.RED)); //output: 1  System.out.println(Color.GREEN.compareTo(Color.GREEN)); //output: 0  System.out.println(Color.GREEN.compareTo(Color.BLUE)); //output: -1  System.out.println(Color.RED.compareTo(Color.BLUE)); //output: -2 |

Результат показывает как располагаются значения перечисления относительно друг друга:

* Число **1**означает, что значение **GREEN**находится правее на одну позицию от значения **RED**
* Число **0**означает, что значение **GREEN**равно само себе
* Число **-1** означает, что значение **GREEN**находится левее от значения **BLUE** на одну позицию
* Число **-2** означает, что значение**RED**находится левее от значения **BLUE**на две позиции

А теперь посмотрим на использование в коллекции **List**

|  |  |
| --- | --- |
| 1  2  3  4  5 | List<ColorEnum> colors = new ArrayList<>(List.of(Color.GREEN, Color.RED, Color.BLUE));  System.out.println(colors); //output: [GREEN, RED, BLUE]    Collections.sort(colors);  System.out.println(colors); //output: [RED, GREEN, BLUE] |

Метод **Collections.sort(colors)**отсортировал список **colors**благодаря тому, что **Enum**реализовывают интерфейс **Comparable.**Посмотрим на реализацию метода **compareTo()** в классе **Enum**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public final int compareTo(E o) {      Enum<?> other = (Enum<?>)o;      Enum<E> self = this;      if (self.getClass() != other.getClass() &&          self.getDeclaringClass() != other.getDeclaringClass())          throw new ClassCastException();      return self.ordinal - other.ordinal;  } |

Пояснения:

* сравнивать перечисления можно только между своими типами. Нельзя сравнивать перечисления типа **Color** с перечислением типа **Car**. Мало того, что компилятор не даст вам это сделать с помощью своих подсказок так еще и в самом методе есть проверка на тип класса перечислений.

Enum реализовывает интерфейс Serializable

Но как и в случае с **clone()**воспользоваться мы им не можем

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | private void readObject(ObjectInputStream in) throws IOException,      ClassNotFoundException {      throw new InvalidObjectException("can't deserialize enum");  }    private void readObjectNoData() throws ObjectStreamException {      throw new InvalidObjectException("can't deserialize enum");  } |

Пояснения:

* причина того, что эти методы приватные да и к тому же бросают исключения при их вызове так же, что и в случае с **clone().**Если бы эта возможность была открыта, тогда легко можно было бы сохранить перечисление в файл, затем считать его обратно и получить на выходе два экземпляра одного значения перечисления. Этот как два значения числа "1".

Но что, если нам недостаточно тех методов, которые предоставляет **enum**? Об этом Вы узнаете в этой статье.

Мы рассмотрим на конкретных примерах следующее:

**1. Конструкторы и переменные**

**2. Методы в перечислениях**

**3. Специальные коллекции для перечислений**

Как и в предыдущей статье, нам понадобится **enum**

|  |  |
| --- | --- |
| 1  2  3 | enum Country {      CANADA, POLAND, GERMANY  } |

Поехали.

**1. Конструкторы и переменные в enum**

У каждой страны есть своя валюта, чтобы её задать необходимо создать конструктор в **enum** **Country**и добавить поле **currency**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | enum Country {      CANADA("CAD"), POLAND("PLN"), GERMANY("EUR");        String currency;        Country(String currency) {          this.currency = currency;      }  } |

Поскольку **enum** - это класс с дополнительными методами, то к нему применяется все то, что вы уже знаете о классах. Т.е. для **enum** можно создавать конструкторы, поля и методы.

Допустим у нас появилась страна, у которой еще нет валюты. В этом случае нам необходимо будет добавить конструктор без параметров, или же **конструктор по умолчанию**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | enum Country {      CANADA("CAD"),      POLAND("PLN"),      GERMANY("EUR"),      LAOPAPAS,      ZIMKABU();        String currency;        Country(String currency) {          this.currency = currency;      }        Country() {      }  } |

Обратите внимание на значения **LAOPAPAS**и**ZIMKABU()**. Использовать конструктор по умолчанию - можно и так и так.

Пример использования **enum** **Currency**

|  |  |
| --- | --- |
| 1  2  3 | for (Country country : Country.values()) {      System.out.println(country + ", " + country.currency);  } |

Вывод в консоль:

|  |  |
| --- | --- |
| 1  2  3  4  5 | CANADA, CAD  POLAND, PLN  GERMANY, EUR  LAOPAPAS, null  ZIMKABU, null |

Поля и методы в перечислениях как и в классах могут иметь модификаторы доступа **private, protected, default, public,**а вот конструкторы в перечислениях всегда будут иметь модификатор  **private.**При попытке указать другой модификатор мы получим ошибку компиляции.

|  |  |
| --- | --- |
| 1  2  3 | public Country(String currency) { // Modifier 'public' is not allowed here      this.currency = currency;  } |

**2. Методы в перечислениях**

В перечислениях можно использовать как обычные (общие) так и абстрактные методы, чтобы задать уникальную логику каждому значению перечисления.

Общие методы в перечислениях

Добавим в перечисление метод **hasCurrency(),**который будет возвращать **true** или **false** в зависимости от того, есть валюта у страны или нет

|  |  |
| --- | --- |
| 1  2  3  4  5 | ...      boolean hasCurrency() {          return currency != null;      }  } |

И рассмотрим использование

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for (Country country : Country.values()) {      if (country.hasCurrency()) {          System.out.println(country + " has currency, it's " + country.currency);      } else {          System.out.println(country + " has no currency");      }  } |

Вывод в консоль

|  |  |
| --- | --- |
| 1  2  3  4  5 | CANADA has currency, it's CAD  POLAND has currency, it's PLN  GERMANY has currency, it's EUR  LAOPAPAS has no currency  ZIMKABU has no currency |

Абстрактные методы в перечислениях

С абстрактными методами немного сложнее, добавим абстрактный метод **void sayHello()**

|  |  |
| --- | --- |
| 1  2  3 | ....      abstract void sayHello();  } |

Теперь нам нужно реализовать этот метод **во всех значениях перечисления**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | CANADA("CAD") {      @Override      void sayHello() {          System.out.println("Hello");      }  },  POLAND("PLN") {      @Override      void sayHello() {          System.out.println("Cześć");      }  },  GERMANY("EUR") {      @Override      void sayHello() {          System.out.println("Hallo");      }  },  LAOPAPAS {      @Override      void sayHello() {          System.out.println("Lapapioooo");      }  },  ZIMKABU() {      @Override      void sayHello() {          System.out.println("Shakalaka");      }  }; |

И посмотрим на использование этого метода.

|  |  |
| --- | --- |
| 1  2  3  4 | for (Country country : Country.values()) {      System.out.print(country + " ");      country.sayHello();  } |

Вывод в консоль

|  |  |
| --- | --- |
| 1  2  3  4  5 | CANADA Hello  POLAND Cześć  GERMANY Hallo  LAOPAPAS Lapapioooo  ZIMKABU Shakalaka |

Если же мы оставим одно значение не переопределённым то получим ошибку компиляции

|  |  |
| --- | --- |
| 1 | ZIMKABU(); 'Country' is abstract; cannot be instantiated |

**3. Специальные коллекции для перечислений**

Никто нам не запрещает использовать такие коллекции как **ArrayList** или **HashSet** вместе с перечислениями, это будет работать точно так же как и с другими значениями. Но раз уж мы знаем наперед значения перечислений, знаем их количество и знаем, что новые значения не будут добавляться в перечисление в работе приложения(в **Runtime**) т.к. это невозможно,  то разработчики придумали специальную коллекцию для них. Эта коллекция работает быстрее и эффективнее обычных, используя, особенности перечислений, которые мы только что описали выше. Эта коллекция называется **EnumSet**. При работе с enum хорошей практикой является использовать именно коллекцию  **EnumSet**вместо стандартных коллекций.

Примеры создания EnumSet.

**Пример 1**

Метод **allOf()** создает **EnumSet** из всех значений заданого перечисления:

|  |  |
| --- | --- |
| 1  2 | EnumSet <Country> countries = EnumSet.allOf(Country.class);  System.out.println(countries); //output: [CANADA, POLAND, GERMANY, LAOPAPAS, ZIMKABU] |

**Пример 2**

Метод **of()**создает **EnumSet**,который содержит заданные значения. Метод перегружен, принимая от одного значения до 5 и еще раз перегружен методом **of(E first, E... rest),**который принимает сколько угодно значений. Всё это нужно для оптимизации и быстродействия **EnumSet:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | EnumSet.of(GERMANY);  EnumSet.of(GERMANY, CANADA);  EnumSet.of(GERMANY, CANADA, POLAND);  EnumSet.of(GERMANY, CANADA, POLAND, LAOPAPAS);  EnumSet.of(GERMANY, CANADA, POLAND, LAOPAPAS, ZIMKABU);  EnumSet.of(GERMANY, CANADA, POLAND, LAOPAPAS, ZIMKABU, GERMANY); |

**Пример 3**

Метод **range()**создает **EnumSet**,который содержит все значения перечисления между указанными значениями:

|  |  |
| --- | --- |
| 1  2 | EnumSet<Country> range = EnumSet.range(CANADA, GERMANY);  System.out.println(range); //output: [CANADA, POLAND, GERMANY] |

**Пример 4**

Метод **copyOf()**возвращает копию указанного **EnumSet,**либо же копию любой другой коллекции этого перечисления:

|  |  |
| --- | --- |
| 1  2  3  4 | EnumSet<Country> canadaSet = EnumSet.of(CANADA);  Set<Country> germanySet = Set.of(GERMANY);  EnumSet<Country> canadaSetCopy = EnumSet.copyOf(canadaSet);  EnumSet<Country> germanySetCopy = EnumSet.copyOf(germanySet); |

**Пример 5**

Метод **complementOf()**создает **EnumSet** из элементов  которые **НЕ** содержатся в изначальной коллекции:

|  |  |
| --- | --- |
| 1  2  3 | EnumSet<Country> initialSet = EnumSet.of(CANADA, POLAND, GERMANY);  EnumSet<Country> complement = EnumSet.complementOf(initialSet);  System.out.println(complement); //output: [LAOPAPAS, ZIMKABU] |

**Пример 6**

Метод **noneOf()**возвращает пустой **EnumSet**заданного типа:

|  |  |
| --- | --- |
| 1  2 | EnumSet<Country> noneOf = EnumSet.noneOf(Country.class);  System.out.println(noneOf); //output: [] |

**P.S.**

На самом деле **EnumSet**это абстрактный класс у которого 2 реализации – **RegularEnumSet** и **JumboEnumSet.**Выбор реализации определяется в методах создания **EnumSet**:

- если количество элементов в перечислении, из которого вы хотите создать **EnumSet,**не превышает 64 то выбран будет **RegularEnumSet**,

- если элементов больше 64 - будет выбран **JumboEnumSet**.

Почему именно 64 элемента?

Потому что в **RegularEnumSet** все значения перечисления умещаются в одну переменную типа **long,**максимальное колличество битов в котором равно 64. Все операции в **RegularEnumSet**  основаны на [булевой логике.](https://ru.wikipedia.org/wiki/%D0%91%D1%83%D0%BB%D0%B5%D0%B2%D0%B0_%D0%B0%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0)

В **JumboEnumSet** также используется булевая логика, но вместе с массивом значений.