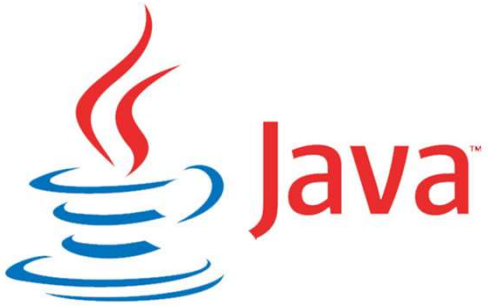
Тема 8.
Принципы ООП



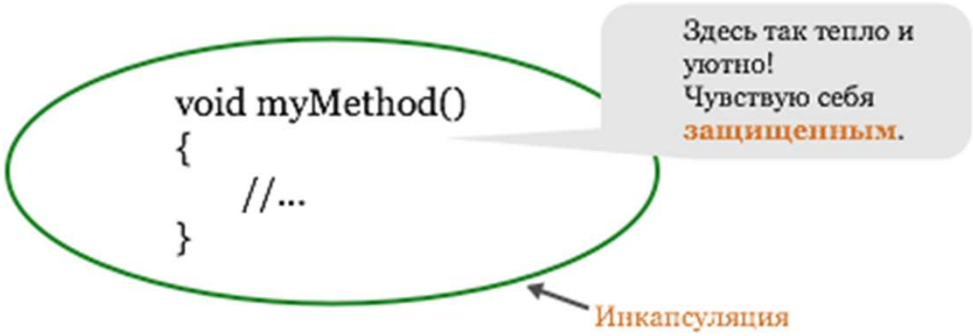
Парадигмы ООП

Парадигмы ООП

Инкапсуляция

Инкапсуляция – это «упаковка», «сокрытие» данных и каких то методов их обработки в один компонент.

В программировании инкапсуляция реализуется при помощи механизма классов.



Парадигмы ООП

Инкапсуляция

Возьмем нашего жука и опишем класс

класс ЖУК:
✔ 4 колеса
✔ МКПП
✔ ДВС

✔ Увеличить скорость
✔ Снизить скорость
✔ Повернуть

Данные/знания

Методы/функции

Класс ЖУК объединяет данные о движущей части и методы работы с ней в единое целое!

Класс ЖУК ИНКАПСУЛИРУЕТ их в себе.

Парадигмы ООП

Инкапсуляция. Другая сторона силы. Сокрытие.

Возьмем нашего многострадального жука

- Жук – объект класса ЖУК, который мы описали ранее, а значит, как мы помним, умеет разгоняться, тормозить и поворачивать.
- Проще говоря – умеет ездить.
- Но большинство людей понятия не имеют, как он это делает и почему. Да и вообще, что происходит, когда они едут в авто.
- Однако им и не надо. Им достаточно того, что авто может их отвезти куда нужно.
- Таким образом наш жук скрывает в себе *реализацию процесса движения*.

Жук хранит в себе знание о том, как ехать, но не раскрывает его!

Жук инкапсулирует в себе это знание.

Парадигмы ООП

Наследование

Наследование – механизм, позволяющий описать класс на основе уже существующего.

Родитель – класс, от которого происходит наследование.

Потомок/наследник – класс, который произошел (был унаследован) от какого-то базового класса (класса-родителя).

Между родителем и потомком устанавливается отношение ***«является»***.

Парадигмы ООП

Наследование

Рассмотрим автомобиль и нашего жука***.***

Все мы знаем, что, вообще говоря, ***наш жук – это автомобиль***.
Как и VW Passat, как и BMW X6 и все остальные.

Жук ЯВЛЯЕТСЯ автомобилем.

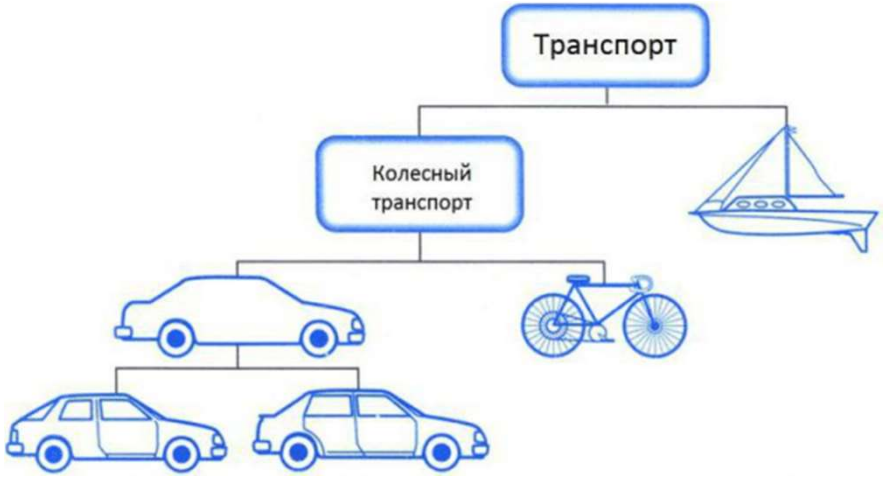
BMW X5 ЯВЛЯЕТСЯ автомобилем.
Ferrari F40 ЯВЛЯЕТСЯ автомобилем.

«Класс-потомок» ЯВЛЯЕТСЯ «Класс-родитель»

Парадигмы ООП

А в чем смысл?

- ***Класс-потомок может пользоваться данными и методами класса-родителя!***
- ***Класс-потомок может добавлять новые данные и методы!***
- ***Класс потомок может переопределять методы класса-родителя!***
- **А ЗНАЧИТ мы можем строить иерархии классов!**
- **Следовательно, нам нужно писать меньше кода!**

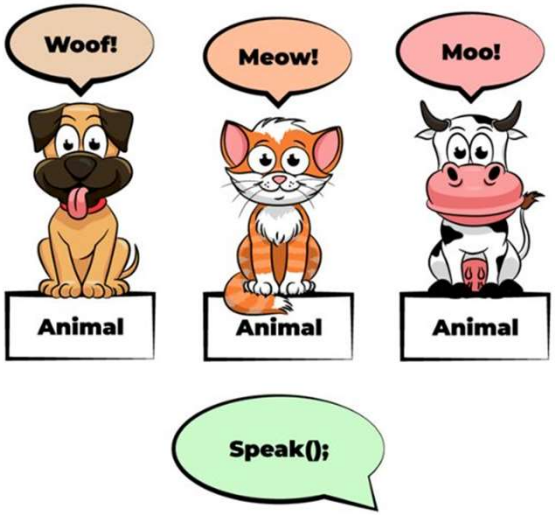


Парадигмы ООП

Полиморфизм

Полиморфизм – способность функции обрабатывать различные типы входных данных.

Полиморфизмом называется возможность работать с несколькими типами так, как будто это ***один и тот же тип*** и в то же время поведение каждого типа будет уникальным в зависимости от его реализации.



Парадигмы ООП

Наглядный пример

Давайте опишем класс Person, представляющий отдельного человека:

```java
public class Person {

    private String name;
    private String surname;

    public Person(String name, String surname){

        this.name=name;
        this.surname=surname;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }

    public void displayInfo(){

        System.out.println("Имя: " + name + " Фамилия: " + surname);
    }
}
```

Парадигмы ООП

Наглядный пример

- В последствии, мы хотели бы расширить имеющуюся систему классов, добавив в нее класс, описывающий сотрудника предприятия – класс Employee;
- Сотрудник предприятия является человеком и, следовательно, имеет тот же функционал, что и класс Person;
- Поэтому, ничто не мешает нам сделать класс Employee производным от класса Person:

```
1 class Employee extends Person{
2
3 }
```

- Чтобы объявить один класс наследником другого, нужно использовать после имени класса-наследника ключевое слово extends, после которого указывается имя базового класса.

Парадигмы ООП

Наглядный пример

- В классе Employee могут быть определены свои поля, методы и конструктор:

```
1  class Employee extends Person{
2
3      private String company;
4
5      public Employee(String name, String surname, String company) {
6
7          super(name, surname);
8          this.company=company;
9      }
10 }
```

Парадигмы ООП

Наглядный пример

- Еще одно определение полиморфизма: полиморфизм – это способность к изменению функциональности, унаследованной от базового класса.
- Переопределим метод displayInfo() класса Person в классе Employee:

```
1  class Employee extends Person{
2
3      private String company;
4
5      public Employee(String name, String surname, String company) {
6
7          super(name, surname);
8          this.company=company;
9      }
10
11     public void displayInfo(){
12         super.displayInfo();
13         System.out.println("Компания: " + company);
14     }
15 }
```

Парадигмы ООП

Наглядный пример

- Класс Employee определяет дополнительное поле для хранения компании, в которой работает сотрудник. Кроме того, это поле устанавливается в конструкторе.
- Так как поля name и surname в базовом классе Person объявлены с модификатором доступа private, к ним нельзя обратиться из класса Employee напрямую. Однако, в данном случае, в этом нет необходимости, так как, чтобы их установить, нужно обратиться к конструктору базового класса с помощью ключевого слова super, после которого в скобках идет перечисление передаваемых аргументов.
- С помощью ключевого слова super можно обратиться к любом члену базового класса – методу или полю, если они не определены с модификатором доступа private.
- Также в классе Employee переопределяется метод displayInfo() базового класса.
- В нем, с помощью ключевого слова super, также идет обращение к методу displayInfo(), но уже БАЗОВОГО класса, а затем выводится дополнительная информация, относящаяся только к Employee.

Парадигмы ООП

Наглядный пример

- Используя обращение к методам базового класса, можно было бы переопределить метод displayInfo() следующим образом:

```
public void displayInfo(){
    System.out.println("Имя: " + super.getName() + " Фамилия: "
        + super.getSurname() + " Компания: " + company);
}
```

- При этом совершенно не обязательно переопределять все методы базового класса. В данном случае, не переопределяются методы getName() и getSurname(), поэтому для этих методов класс-наследник будет использовать реализацию из базового класса.
- Можно использовать эти методы в основной программе:

```
public static void main(String[] args) {
    Employee empl = new Employee("Tom", "Simpson", "Oracle");
    empl.displayInfo();
    String firstName = empl.getName();
    System.out.println(firstName);
}
```

Пример

Немного о private в наследовании

```
public class Main   {
    public static void main(String[] args) {
        Child child = new Child();
        child.privateHello();
        System.out.println(child.str);
        child.protectedHello();
    }
}
```

```
class Parent {
    private String str = "Private";
    private void privateHello() {
        System.out.println(str);
    }

    protected void protectedHello() {
        privateHello();
        System.out.println(str + " protected");
    }
}
```

```
class Child extends Parent {
    public void test()  {
        privateHello();
        System.out.println(str);
    }
}
```

Анонимные классы

Почему «анонимные»? ☺

Предположим, что имеется основная программа, которая постоянно работает и что-то делает. Перед разработчиком стоит задача создать для этой программы систему мониторинга из нескольких модулей:

- Модуль для отслеживания общих показателей работы и ведения лога;
- Модуль для фиксации и регистрации ошибки в журнале ошибок;
- Модуль для отслеживания подозрительной активности (например, попытки несанкционированного доступа и прочие связанные с безопасностью вещи).

Поскольку все три модуля должны, по сути, просто стартовать в начале программы и работать в фоновом режиме, будет хорошей идеей создать для них общий интерфейс:

```
public interface MonitoringSystem {

    public void startMonitoring();
}
```

Анонимные классы

Почему «анонимные»? ☺

Данный интерфейс будут реализовывать 3 конкретных класса-модуля:

```
public class GeneralIndicatorsMonitoringModule implements MonitoringSystem {

@Override
    public void startMonitoring() {
        System.out.println("Мониторинг общих показателей стартовал!");
    }
}


public class ErrorMonitoringModule implements MonitoringSystem {

    @Override
    public void startMonitoring() {
        System.out.println("Мониторинг отслеживания ошибок стартовал!");
    }
}


public class SecurityModule implements MonitoringSystem {

    @Override
    public void startMonitoring() {
        System.out.println("Мониторинг безопасности стартовал!");
    }
}
```

Анонимные классы

Почему «анонимные»? ☺

В данном примере все выглядит логично:

- Есть система, состоящая из нескольких модулей;
- У каждого модуля есть собственное поведение;
- Можно легко добавлять новые модули, ведь имеется интерфейс, который просто реализовать.

Все логично, но: чтобы система заработала, нужно просто создать 3 объекта - GeneralIndicatorsMonitoringModule, ErrorMonitoringModule, SecurityModule – и вызвать метод startMonitoring() у каждого из них.

```
public class Main {

    public static void main(String[] args) {

        GeneralIndicatorsMonitoringModule generalModule = new GeneralIndicatorsMonitoringModule();
        ErrorMonitoringModule errorModule = new ErrorMonitoringModule();
        SecurityModule securityModule = new SecurityModule();

        generalModule.startMonitoring();
        errorModule.startMonitoring();
        securityModule.startMonitoring();
    }
}
```

Анонимные классы

Почему «анонимные»? ☺

Вывод в консоль:
Мониторинг общих показателей стартовал!
Мониторинг отслеживания ошибок стартовал!
Мониторинг безопасности стартовал!

И для такой небольшой работы была написана целая система аж с 3, по сути, одноразовыми классами и целым интерфейсом! ☺

Вопрос ☺

Что можно изменить?

Анонимные классы

Почему «анонимные»? ☺

Здесь на помощь приходят анонимные внутренние классы.

```
public class Main {

   public static void main(String[] args) {

       MonitoringSystem generalModule = new MonitoringSystem() {
           @Override
           public void startMonitoring() {
               System.out.println("Мониторинг общих показателей стартовал!");
           }
       };


       MonitoringSystem errorModule = new MonitoringSystem() {
                   @Override
                   public void startMonitoring() {
                       System.out.println("Мониторинг отслеживания ошибок стартовал!");
                   }
               };

       MonitoringSystem securityModule = new MonitoringSystem() {
           @Override
           public void startMonitoring() {
               System.out.println("Мониторинг безопасности стартовал!");
           }
       };

       generalModule.startMonitoring();
       errorModule.startMonitoring();
       securityModule.startMonitoring();
   }
}
```

Анонимные классы

Что же тут происходит? ☺

Выглядит все так, как будто создается объект интерфейса, но его объект создать нельзя.

В тот момент, когда пишется

```
MonitoringSystem generalModule = new MonitoringSystem() {

};
```

внутри Java-машины происходит следующее:

1. Создается безымянный Java-класс, реализующий интерфейс MonitoringSystem;
2. Компилятор, увидев такой класс, начинает требовать от разработчика реализовать все методы интерфейса MonitoringSystem (что и происходит 3 раза);
3. Создается один объект этого класса.

Анонимные классы

Что же тут происходит? ☺

Обратите внимание на код – в конце стоит точка с запятой. И стоит не просто так: здесь одновременно объявляется класс с помощью **{}** и создается его объект с помощью **();**.

Каждый из наших трех объектов переопределил метод startMonitoring() по-своему.

И осталось лишь вызвать этот метод у каждого из них:

```
generalModule.startMonitoring();
errorModule.startMonitoring();
securityModule.startMonitoring();
```

Задача выполнена!

Все три модуля успешно запущены и работают, а структура программы стала намного проще.

Анонимные классы

Что же тут происходит? ☺

Если каждому из анонимных классов-модулей понадобится какое-то отличающееся поведение, свои специфические методы, которых нет у других, их можно легко дописать:

```
MonitoringSystem generalModule = new MonitoringSystem() {

    @Override
    public void startMonitoring() {
        System.out.println("Мониторинг общих показателей стартовал!");
    }

    public void someSpecificMethod() {

        System.out.println("Специфический метод только для первого модуля");
    }
};
```

В документации Oracle приведена хорошая рекомендация: «Применяйте анонимные классы, если вам нужен локальный класс для одноразового использования».

Анонимные классы

Особенности анонимных классов

- Анонимный класс — это полноценный внутренний класс, поэтому у него есть доступ к переменным внешнего класса, в том числе к статическим и private:

```
public class Main {

    private static int currentErrorsCount = 23;

    public static void main(String[] args) {

        MonitoringSystem errorModule = new MonitoringSystem() {

            @Override
            public void startMonitoring() {
                System.out.println("Мониторинг отслеживания ошибок стартовал!");
            }

            public int getCurrentErrorsCount() {

                return currentErrorsCount;
            }
        };
    }
}
```

Анонимные классы

Особенности анонимных классов

- Анонимные классы видны только внутри того метода, в котором определены: любые попытки обратиться к объекту errorModule за пределами метода main() будут неудачными;
- Анонимный класс не может содержать статические переменные и методы:

```
//ошибка! Inner classes cannot have static declarations
public static int getCurrentErrorsCount() {

    return currentErrorsCount;
}
```

- Тот же результат будет, если попробовать объявить статическую переменную:

```
MonitoringSystem errorModule = new MonitoringSystem() {

    //ошибка! Inner classes cannot have static declarations!
    static int staticInt = 10;

    @Override
    public void startMonitoring() {
        System.out.println("Мониторинг отслеживания ошибок стартовал!");
    }

};
```

Перечисления (enum) в Java

Представим, что перед разработчиком поставили задачу создать класс, представляющий дни недели. На первый взгляд, ничего сложного в этом нет, и код будет выглядеть следующим образом:

```
public class DayOfWeek {

    private String title;

    public DayOfWeek(String title) {
        this.title = title;
    }

    public static void main(String[] args) {
        DayOfWeek dayOfWeek = new DayOfWeek("Суббота");
        System.out.println(dayOfWeek);
    }

    @Override
    public String toString() {
        return "DayOfWeek{" +
                "title='" + title + '\'' +
                '}';
    }
}
```

Перечисления (enum) в Java

Вроде бы, все в порядке, НО в конструктор класса DayOfWeek можно передать любой текст. Таким образом, кто-то сможет создать день недели «Лягушка», «Облачко» или «azaza322». А это далеко не то поведение, которое ожидает от класса разработчик и пользователи, ведь реальных дней недели существует всего 7, и у каждого из них есть название.

Перед разработчиком появляется еще одна важная задача - как-то ограничить круг возможных значений для класса «день недели».

До появления Java 1.5 разработчики были вынуждены самостоятельно придумывать решение этой проблемы, поскольку готового решения в самом языке не существовало.

В те времена, если ситуация требовала ограниченного числа значений, делали так:

Перечисления (enum) в Java


```
public class DayOfWeek {

    private String title;

    private DayOfWeek(String title) {
        this.title = title;
    }

    public static DayOfWeek SUNDAY = new DayOfWeek("Воскресенье");
    public static DayOfWeek MONDAY = new DayOfWeek("Понедельник");
    public static DayOfWeek TUESDAY = new DayOfWeek("Вторник");
    public static DayOfWeek WEDNESDAY = new DayOfWeek("Среда");
    public static DayOfWeek THURSDAY = new DayOfWeek("Четверг");
    public static DayOfWeek FRIDAY = new DayOfWeek("Пятница");
    public static DayOfWeek SATURDAY = new DayOfWeek("Суббота");

    @Override
    public String toString() {
        return "DayOfWeek{" +
                "title='" + title + '\'' +
                '}';
    }
}
```

Перечисления (enum) в Java

В чем здесь особенность:

- **Закрытый (private) конструктор.** Если конструктор помечен модификатором private, объект класса нельзя создать с помощью этого конструктора. А поскольку в этом классе конструктор всего один, объект DayOfWeek нельзя создать вообще.

```
public class Main {

        public static void main(String[] args) {

                DayOfWeek sunday = new DayOfWeek();//ошибка!
        }
}
```

- Нужное количество public static объектов, представляющих правильные названия дней недели, что позволяло использовать объекты в других классах.

```
public class Man {

        public static void main(String[] args) {

                DayOfWeek sunday = DayOfWeek.SUNDAY;

                System.out.println(sunday);
         }
}
```

Перечисления (enum) в Java

Такой подход во многом позволял решить задачу. В распоряжении пользователя были 7 дней недели, и при этом никто не мог создать новые.

С выходом Java 1.5 в языке появилось готовое решение для таких ситуаций — перечисление (Enum).

Enum — тоже класс, но он специально «заточен» на решение задач, похожих на пример выше - создание некоторого ограниченного круга значений.

Поскольку у создателей Java уже были готовые примеры (скажем, язык C, в котором Enum уже существовал), они смогли создать оптимальный вариант.

Итак, что же из себя представляет Enum в Java? Снова начнем с примера.

Перечисления (enum) в Java

Давайте опишем с помощью enum тип данных для хранения времени года:

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

Ну и простой пример его использования:

```
Season season = Season.SPRING;
if (season == Season.SPRING) season = Season.SUMMER;
System.out.println(season);
```

В результате выполнения которого на консоль будет выведено ***SUMMER***.

Перечисления (enum) в Java

Элементы перечисления — экземпляры enum-класса, доступные статически.

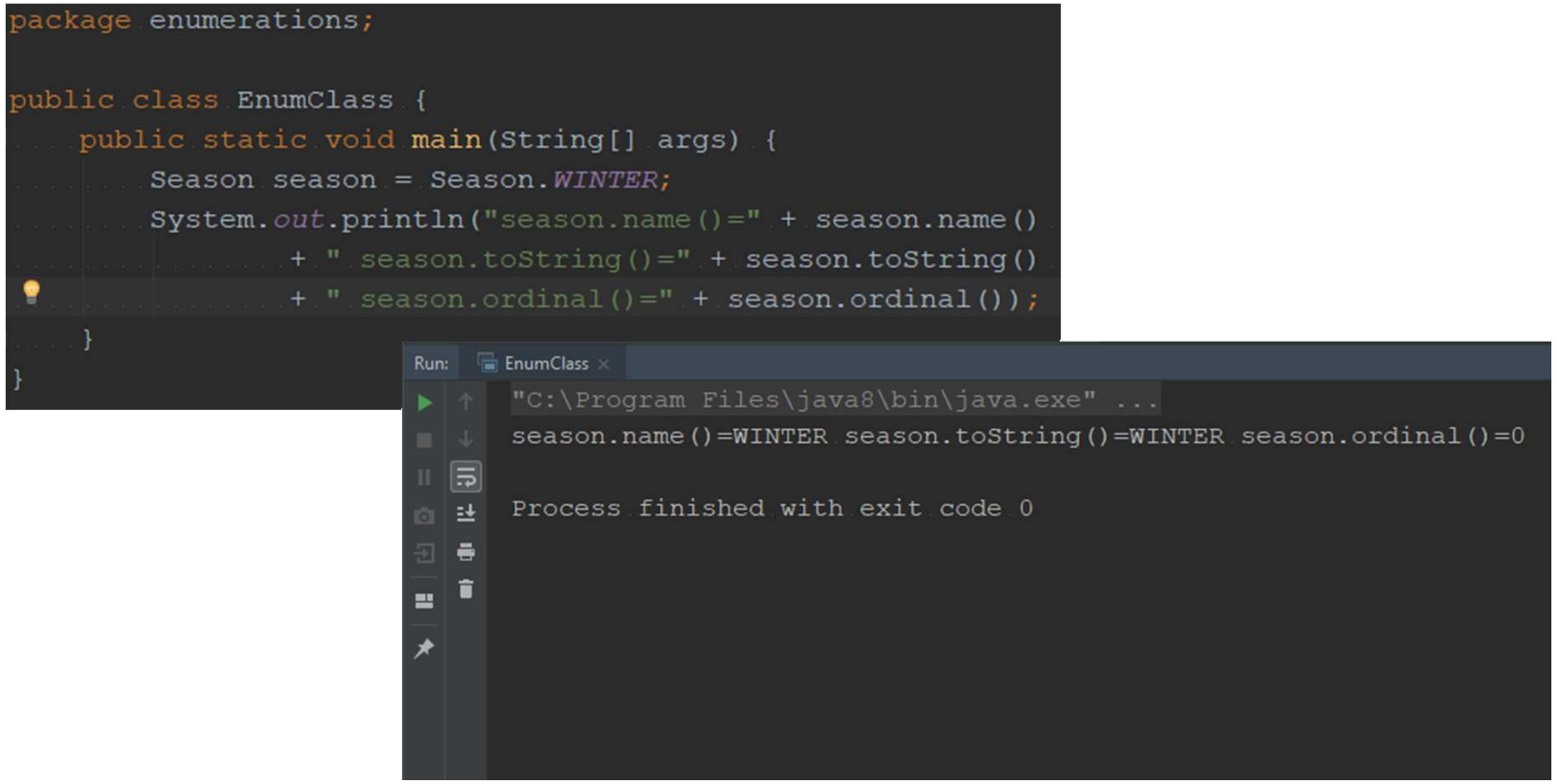
- Элементы enum Season (WINTER, SPRING и т.д.) — это статически доступные экземпляры enum-класса Season.
- Их статическая доступность позволяет выполнять сравнение с помощью оператора сравнения ссылок ==

```
Season season = Season.SUMMER;
if (season == Season.AUTUMN) season = Season.WINTER;
```

Методы перечислений.

- name() – возвращает имя константы, определенной в перечислении.
- toString() - возвращает имя константы, определенной в перечислении, и является наиболее используемым при работе с перечислениями.
- ordinal() - возвращает порядковый номер определенной константы (нумерация начинается с 0).

Перечисления (enum) в Java



```
package enumerations;

public class EnumClass {
    public static void main(String[] args) {
        Season season = Season.WINTER;
        System.out.println("season.name()=" + season.name()
                + " season.toString()=" + season.toString()
                + " season.ordinal()=" + season.ordinal());
    }
}
```

```
Run:  EnumClass
"C:\Program Files\java8\bin\java.exe" ...
season.name()=WINTER season.toString()=WINTER season.ordinal()=0

Process finished with exit code 0
```

Пример использования методов name(), toString() и ordinal().

Перечисления (enum) в Java

Методы перечислений.

- valueOf(String name) – получение элемента enum по строковому представлению его имени.

```
String name = "WINTER";
Season season = Season.valueOf(name);
```

В результате выполнения кода переменная season будет равна Season.WINTER.

- Следует обратить внимание, что если элемент не будет найден, то будет выброшен IllegalArgumentException, а в случае, если name равен null — NullPointerException.
- values() – получение всех элементов перечисления.

```
System.out.println(Arrays.toString(Season.values()));
```

```
[WINTER, SPRING, SUMMER, AUTUMN]
```

Перечисления (enum) в Java

Пользовательские методы в перечислениях.

Существует возможность добавлять собственные методы как в enum-классы, так и в их элементы:

```
package enumerations;

public enum Direction {
    UP, DOWN;

    public Direction opposite(){
        return this ==UP ? DOWN : UP;
    }
}
```

```
package enumerations;

public class EnumClass {
    public static void main(String[] args) {
        Direction direction = Direction.UP;
        System.out.println(direction.opposite());
    }
}
```

Практика

Создайте ENUM со всеми днями недели. Сделайте метод в классе, который выведет все дни недели в консоль с указанием рабочий это день или выходной.

Практика

Напишите класс BaseConverter для конвертации из градусов по Цельсию в Кельвины, Фаренгейты, и так далее. У класса должен быть метод convert, который и делает конвертацию.

Соотношения температур:

- <Температура по Цельсию>
- <Температура по Цельсию> + 273.15 —-> по Кельвину
- 1.8 * <Температура по Цельсию> + 32 —-> по Френгейту