Java Pro 7.2

TEL-RAN
by Starta Institute

Аннотации. Паттерны для введения в Spring

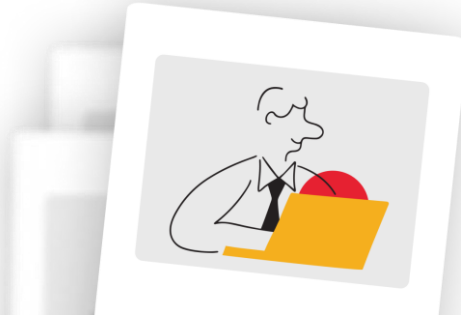
ПРЕПОДАВАТЕЛЬ

TEL-RAN
by Starta Institute

Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

TEL-RAN
by Starta Institute

ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа

TEL-RAN
by Starta Institute

1 ПОВТОРЕНИЕ ИЗУЧЕННОГО

Повторение

TEL-RAN
by Starta Institute

Исправьте ошибку в коде

```
public static void main(String[] args) {
    Class<?> cls = Class.forName("NonExistentClass");
    System.out.println(cls.getName());
}
```

Повторение

TEL-RAN
by Starta Institute

Исправьте ошибку в коде

```
public static void main(String[] args) throws ClassNotFoundException {
    Class<?> cls = Class.forName("NonExistentClass");
    System.out.println(cls.getName());
}
```

Ошибка в строке *Class<?> cls = Class.forName("NonExistentClass");*. Если класс с именем *"NonExistentClass"* не существует, метод *Class.forName* выбросит *ClassNotFoundException*. Необходимо обработать это исключение.

Повторение

TEL-RAN
by Starta Institute

Исправьте три ошибки в коде

```
public class MyClass {
    private String myField;

    public static void main(String[] args) {
        MyClass obj = new MyClass();
        Field field = MyClass.class.getField("myField");
        field.set(obj, "Hello, Reflection!");
        System.out.println(obj.myField);
    }
}
```

Повторение

TEL-RAN
by Starta Institute

Исправьте три ошибки в коде

```
public class MyClass {
    private String myField;

    public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
        MyClass obj = new MyClass();
        Field field = MyClass.class.getDeclaredField("myField");
        field.setAccessible(true);
        field.set(obj, "Hello, Reflection!");
        System.out.println(obj.myField);
    }
}
```

1 Метод *getField()* возвращает только публичные поля. Нужно использовать *getDeclaredField()*. 2 Поле *myField* является приватным. Чтобы получить к нему доступ, нужно вызвать метод *field.setAccessible(true)*. 3 Методы *getDeclaredField()* и *set()* выбрасывают checked exception.

Повторение

TEL-RAN
by Starta Institute

Исправьте ошибку в коде

```
public static void main(String[] args) throws NoSuchMethodException, InvocationTargetException,
InstantiationException, IllegalAccessException {
    Class<?> cls = String.class;
    Constructor<?> constructor = cls.getConstructor(int.class);
    String str = (String) constructor.newInstance(5);
    System.out.println(str);
}
```

Повторение

TEL-RAN
by Starta Institute

Исправьте ошибку в коде

```
public static void main(String[] args) throws NoSuchMethodException, InvocationTargetException,
InstantiationException, IllegalAccessException {
    Class<?> cls = String.class;
    Constructor<?> constructor = cls.getConstructor(char[].class);
    String str = (String) constructor.newInstance(new char[0]);
    System.out.println(str);
}
```

Ошибка в строке *Constructor<?> constructor = cls.getConstructor(int.class);*.

Класс *String* не имеет конструктора, принимающего *int*. Необходимо использовать другой конструктор.

Повторение

TEL-RAN
by Starta Institute

Что будет выведено в консоль?

```
public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
    Class<Byte> clazz = Byte.class;
    Field field = clazz.getField("MAX_VALUE");
    System.out.println(field.get(null));
}
```

Повторение

TEL-RAN
by Starta Institute

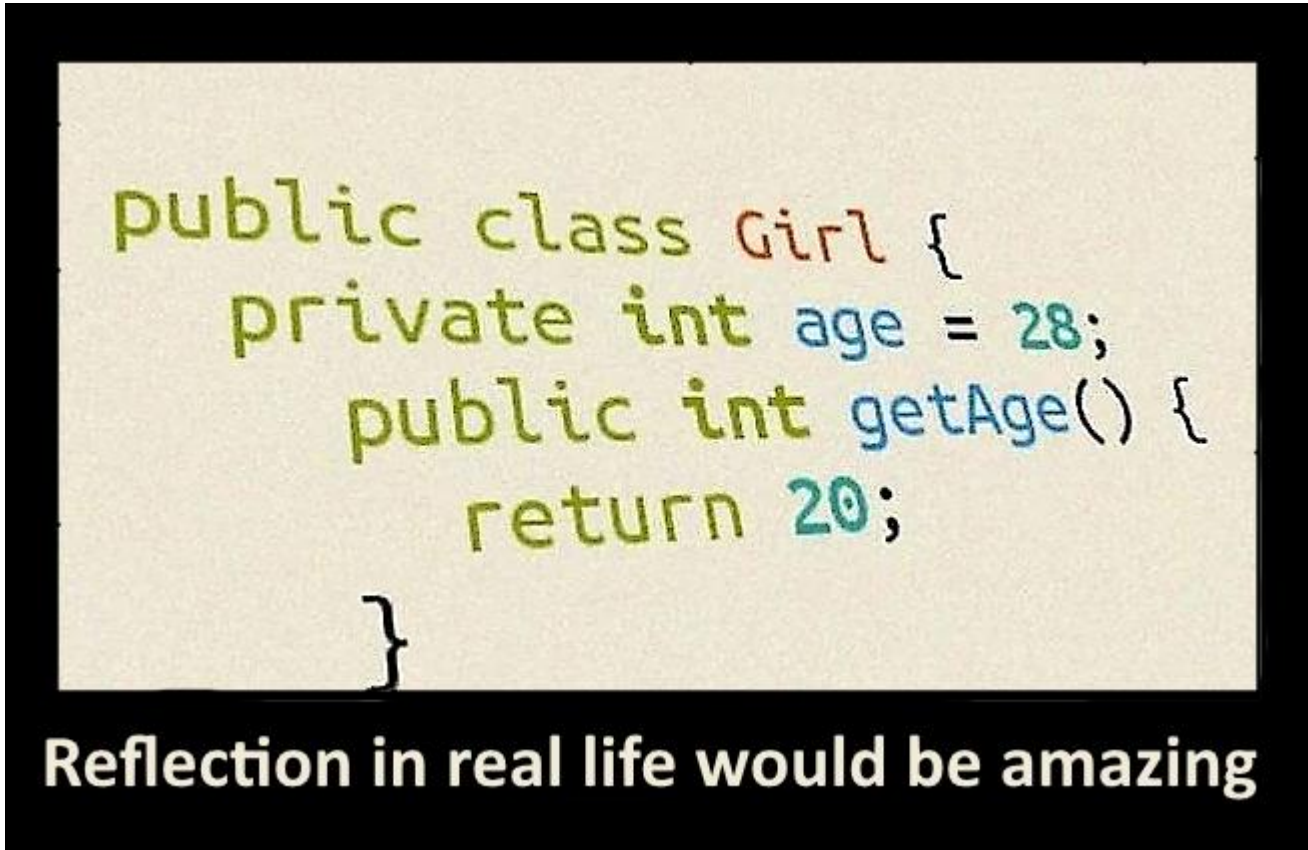
Что будет выведено в консоль?

```
public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
    Class<Byte> clazz = Byte.class;
    Field field = clazz.getField("MAX_VALUE");
    System.out.println(field.get(null));
}
```

127 (значение статического поля *MAX_VALUE*)

Повторение

В чём прикол мема?



TEL-RAN
by Starta Institute

2 ОСНОВНОЙ БЛОК

Введение

TEL-RAN
by Starta Institute

- Делаем замечания
- Неповторимый
- Одинокий волк

Проблема

Рефлексия помогает программе исследовать используемые классы.

Но как это помогает улучшать работу программы и автоматизировать разработку с помощью фреймворков?

Делаем замечания

Аннотации

TEL-RAN
by Starta Institute

Аннотация (от лат. *annotatio* «замечание») – в языке Java является специальной формой синтаксических метаданных, которая может быть добавлена в исходный код. Используются для анализа кода, компиляции или выполнения. Аннотируемы пакеты, классы, методы, переменные и параметры.

Выглядит как *@ИмяАннотации*.

Основные функции аннотаций:

- даёт необходимую информацию для компилятора / интерпретатора;
- даёт информацию различным инструментам для генерации кода, конфигураций и т. д.;
- может использоваться во время выполнения для получения данных через *reflection API*.

Аннотации появились в Java 1.5 и были интегрированы в javac в Java 1.6.

Делаем замечания

Виды аннотаций

TEL-RAN
by Starta Institute

Аннотации, применяемые к исходному коду:

@Override – аннотация-маркер, которая может применяется к методам. Показывает компилятору, что метод переопределяет метод класса-предка. Вызывает ошибку компиляции, если метод не найден в родительском классе или интерфейсе;

@Deprecated – отмечает, что метод устарел и не рекомендуется к использованию. Этот метод пока оставлен, но будет удалён в будущих версиях. Вызывает предупреждение компиляции, если метод используется;

@SuppressWarnings – указывает компилятору подавить предупреждения компиляции, определённые в параметрах аннотации;

Делаем замечания

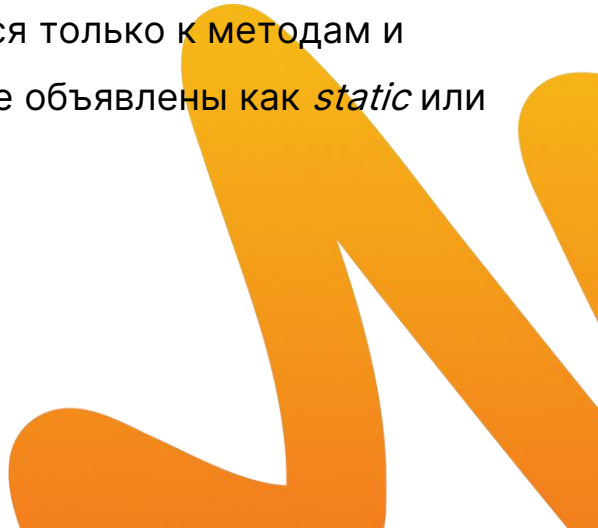
Виды аннотаций

TEL-RAN
by Starta Institute

Аннотации, применяемые к исходному коду:

@FunctionalInterface - аннотация, представленная в Java 8 и указывающая, что тип предлагается к использованию как функциональный интерфейс.

@SafeVarargs – указывает, что никакие небезопасные действия, связанные с параметром переменного количества аргументов, недопустимы. Применяется только к методам и конструкторам с переменным количеством аргументов, которые объявлены как *static* или *final*.



Делаем замечания

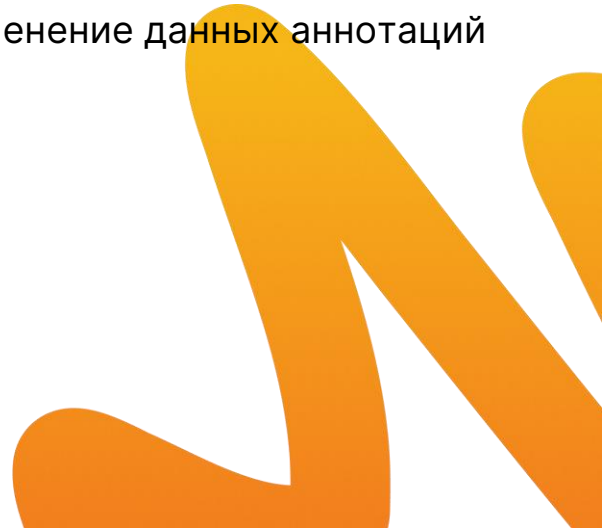
Виды аннотаций

TEL-RAN
by Starta Institute

Аннотации, применяемые к исходному коду:

Аннотации типов предназначены для улучшенного анализа программ и более строгой проверки типов. Например, *@NonNull, @NotEmpty, @Nullable* и т.д.

Java 8 определяет аннотации типов, но не реализует их. Вместо этого предлагается использовать сторонние фреймворки, реализующие их. Т.е. при енение данных аннотаций без фреймворка не имеет смысла.



Делаем замечания

Виды аннотаций

TEL-RAN
by Starta Institute

Аннотации, применяемые к другим аннотациям:

@Retention – определяет стадию, до которой "доживает" аннотация внутри класса: будет она присутствовать только в исходном коде, в скомпилированном файле, или она будет также видна и в процессе выполнения. Каждая аннотация имеет только один из возможных "типов хранения" указанный в классе *RetentionPolicy*:

SOURCE – аннотация используется только при написании кода и игнорируется компилятором, т.е. не сохраняется после компиляции (генерация кода).

CLASS – аннотация сохраняется после компиляции, однако игнорируется JVM, т.е. не может быть использована во время выполнения (*plug-in* приложения).

RUNTIME – аннотация, которая сохраняется после компиляции и подгружается JVM (т.е. может использоваться во время выполнения самой программы). Используется в качестве меток в коде, которые напрямую влияют на ход выполнения программы (*рефлексия*)

Делаем замечания

Виды аннотаций

TEL-RAN
by Starta Institute

Аннотации, применяемые к другим аннотациям:

@Target – указывает, что именно можно пометить этой аннотацией: поле, метод, тип и т. д.

ANNOTATION_TYPE - другая аннотация

CONSTRUCTOR - конструктор класса

FIELD - поле класса

LOCAL_VARIABLE - локальная переменная

METHOD - метод класса

PACKAGE - описание пакета package

PARAMETER - параметр метода

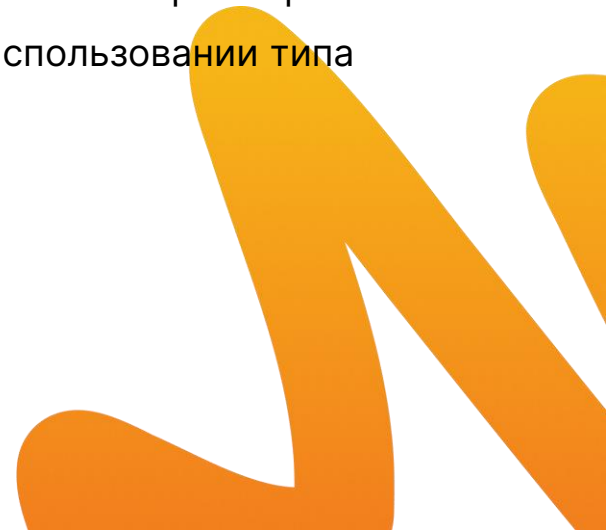
TYPE - указывается над классом, перечислением или записью

MODULE – модуль проекта

RECORD_COMPONENT - запись

TYPE_PARAMETER – тип параметра

TYPE_USE – при использовании типа



Делаем замечания

Виды аннотаций

TEL-RAN
by Starta Institute

Аннотации, применяемые к другим аннотациям:

@Documented – указывает, что помеченная таким образом аннотация должна быть добавлена в *javadoc* поля/метода и так далее.

@Inherited – отмечает, что аннотация может быть расширена подклассами аннотируемого класса, т.е. помечает аннотацию, которая будет унаследована потомком класса, отмеченного такой аннотацией.

@Repeatable - аннотация, появившаяся в Java 8 и определяющая, что помеченная аннотация может быть применена более одного раза к одному и тому же объявлению класса или его использованию. Применяется, когда надо написать аннотацию, запускающую метод в заданное время или по определенному расписанию.

Делаем замечания

Собственные аннотации

TEL-RAN
by Starta Institute

Объявление аннотации похоже на объявление интерфейса с использованием знака @ перед ключевым словом *interface*:

```
@Edible
Food food = new Food();

public @interface Edible{
    //создание собственной аннотации
}
```

Делаем замечания **Собственные аннотации**

Пользовательские аннотации могут включать в себя различные значения, которые описываются как методы аннотации. Каждое объявление метода определяет элемент аннотации. Объявление метода не должно включать в себя каких-либо аргументов или инструкции *throws*. Возвращаемый тип обязан быть примитивным типом, строкой, классом, перечислением или массивом одного из указанных типов. Методы могут иметь значения по умолчанию.

```
@Edible(true) // присваивание true для edible
Food food = new Food();

public @interface Edible {
    // по умолчанию edible будет false
    boolean edible() default false;
}
```

```
public @interface Author {
    String first_name();
    String last_name();
}

@Author(first_name="James", last_name="Gosling")
Book book = new Book();
```

```
@Target({ElementType.METHOD})
public @interface SomeAnnotation{}

public class SomeClass{
    @SomeAnnotation
    private void doSomething(){}
}
```

Делаем замечания

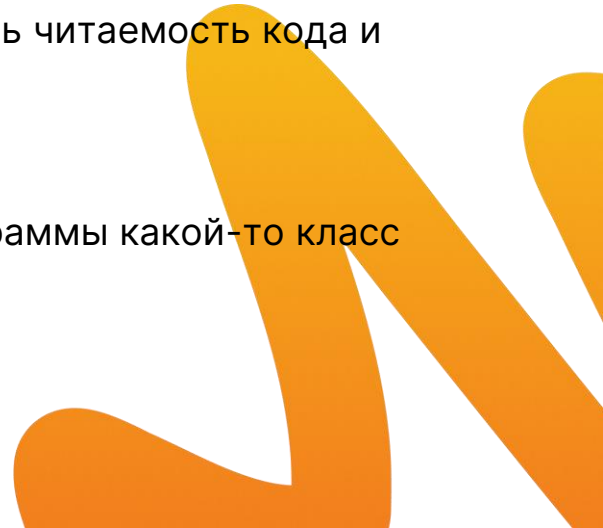
Собственные аннотации

TEL-RAN
by Starta Institute

Аннотации являются характеристиками членов класса, над которыми они указаны. Их можно получать с помощью рефлексии методами *getDeclaredAnnotation(Class c)*, *getAnnotation(Class c)*, *getAnnotationsByType(Class c)* или *getDeclaredAnnotationsByType(Class c)*.

Собственная аннотация может заменить комментарии, повысить читаемость кода и привнести новый функционал.

Для анализа аннотаций и реакции на них во время работы программы какой-то класс должен обрабатывать объекты, получая их аннотации.



Делаем замечания

Повторяющиеся аннотации

TEL-RAN by Starta Institute

Повторяющиеся аннотации можно использовать несколько раз. Например, есть аннотация *@Game* для проведения игр в разные дни. Перед определением класса *Main* можно применить несколько раз аннотацию *@Game*

```
import java.lang.annotation.Repeatable;

@Repeatable(Games.class)
@interface Game {
    String name() default "Что-то под вопросом";
    String day();
}
```

```
Крикет в воскресенье
Что-то под вопросом в четверг
Хоккей в понедельник
```

```
@Game(name = "Крикет",  day = "воскресенье")
@Game(day = "четверг")
@Game(name = "Хоккей",   day = "понедельник")
public class Main {
    public static void main(String[] args) {
        Games games = Main.class.getAnnotation(Games.class);

        for (Game game : games.value()) {
            System.out.println(game.name() + " в " + game.day());
        }
    }
}
```

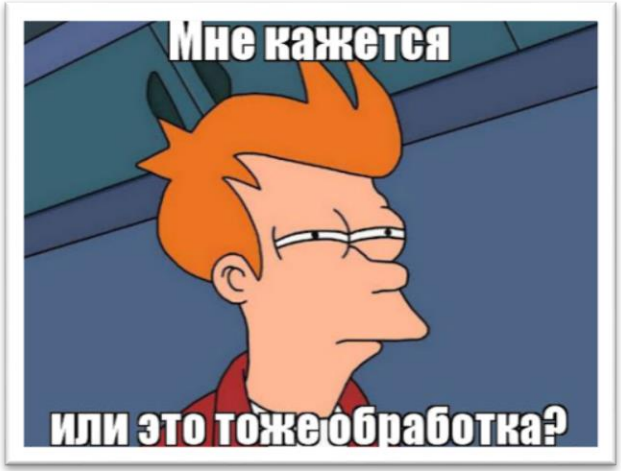
Делаем замечания

Обработка аннотаций

TEL-RAN
by Starta Institute

Обработка аннотаций может выполняться:

1. Во время выполнения программы с помощью рефлексии (например, *Spring* стал использовать аннотации в дополнение к *xml*-конфигурации);
2. Во время компиляции с помощью обработчиков аннотаций – классов, которые нужно зарегистрировать в качестве обработчиков (*annotationProcessor*).



Делаем замечания

Обработка аннотаций

Обработка во время компиляции с *Maven* является вопросом настройки плагина компилятора.

IntelliJ IDEA позволяет использовать обработчики аннотаций, указанные в настройках проекта:
File -> Settings -> Build, Execution, Deployment -> Compiler -> Annotation Processors

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-compiler-plugin</artifactId>
         <version>3.8.1</version>
         <configuration>
            <annotationProcessors>
               <annotationProcessor>
                  ch.frankel.blog.SampleProcessor
               </annotationProcessor>
            </annotationProcessors>
         </configuration>
      </plugin>
   </plugins>
</build>
```

Делаем замечания
Обработка аннотаций

Сам обработчик должен реализовывать *Processor*, но абстрактный класс *AbstractProcessor* самостоятельно реализует большую часть своих методов, кроме метода *process()*: на практике достаточно наследоваться от *AbstractProcessor*.

```
@SupportedAnnotationTypes("ch.frankel.blog.*")              // (1)
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SampleProcessor extends AbstractProcessor {

    @Override
    public boolean process(                                 // (2)
        Set<? extends TypeElement> annotations,
        RoundEnvironment env
    ) {
        annotations.forEach(annotation -> {                 // (3)
            Set<? extends Element> elements =
env.getElementsAnnotatedWith(annotation);                   // (4)
            elements.stream()
                    .filter(TypeElement.class::isInstance)  // (5)
                    .map(TypeElement.class::cast)           // (6)
                    .map(TypeElement::getQualifiedName)     // (7)
                    .map(name -> "Class " + name + " is annotated with " +
annotation.getQualifiedName())
                    .forEach(System.out::println);
        });
        return true;
    }
}
```

Делаем замечания

Обработка аннотаций

1) Обработчик будет вызываться для каждой аннотации, принадлежащей пакету *ch.frankel.blog*

(2) *process()*: основной метод, подлежащий переопределению

(3) Цикл вызывается для каждой аннотации

```
@SupportedAnnotationTypes("ch.frankel.blog.*")              // (1)
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SampleProcessor extends AbstractProcessor {

    @Override
    public boolean process(                                 // (2)
        Set<? extends TypeElement> annotations,
        RoundEnvironment env
    ) {
        annotations.forEach(annotation -> {                 // (3)
            Set<? extends Element> elements =
env.getElementsAnnotatedWith(annotation);                   // (4)
            elements.stream()
                    .filter(TypeElement.class::isInstance)  // (5)
                    .map(TypeElement.class::cast)           // (6)
                    .map(TypeElement::getQualifiedName)     // (7)
                    .map(name -> "Class " + name + " is annotated with " +
annotation.getQualifiedName())
                    .forEach(System.out::println);
        });
        return true;
    }
}
```

Делаем замечания

Обработка аннотаций

(4) Аннотация не так интересна, как аннотированный ею элемент. Это способ получить аннотированный элемент

```
@SupportedAnnotationTypes("ch.frankel.blog.*")                    // (1)
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SampleProcessor extends AbstractProcessor {

    @Override
    public boolean process(                                       // (2)
        Set<? extends TypeElement> annotations,
        RoundEnvironment env
    ) {
        annotations.forEach(annotation -> {                       // (3)
            Set<? extends Element> elements =
env.getElementsAnnotatedWith(annotation);                         // (4)
            elements.stream()
                    .filter(TypeElement.class::isInstance)        // (5)
                    .map(TypeElement.class::cast)                 // (6)
                    .map(TypeElement::getQualifiedName)           // (7)
                    .map(name -> "Class " + name + " is annotated with " +
annotation.getQualifiedName())
                    .forEach(System.out::println);
        });
        return true;
    }
}
```

Делаем замечания

Обработка аннотаций

(5) В зависимости от того, какой элемент аннотирован, его необходимо привести к правильному дочернему интерфейсу Element.

```
@SupportedAnnotationTypes("ch.frankel.blog.*")              // (1)
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SampleProcessor extends AbstractProcessor {

    @Override
    public boolean process(                                 // (2)
        Set<? extends TypeElement> annotations,
        RoundEnvironment env
    ) {
        annotations.forEach(annotation -> {                 // (3)
            Set<? extends Element> elements =
env.getElementsAnnotatedWith(annotation);                   // (4)
            elements.stream()
                    .filter(TypeElement.class::isInstance)  // (5)
                    .map(TypeElement.class::cast)           // (6)
                    .map(TypeElement::getQualifiedName)     // (7)
                    .map(name -> "Class " + name + " is annotated with " +
annotation.getQualifiedName())
                    .forEach(System.out::println);
        });
        return true;
    }
}
```

Делаем замечания

Обработка аннотаций

(6) Нам нужно полное имя класса, для которого установлена аннотация, поэтому необходимо привести его к типу, который делает этот конкретный атрибут доступным

(7) получаем полное имя от TypeElement

```
@SupportedAnnotationTypes("ch.frankel.blog.*")              // (1)
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SampleProcessor extends AbstractProcessor {

    @Override
    public boolean process(                                 // (2)
        Set<? extends TypeElement> annotations,
        RoundEnvironment env
    ) {
        annotations.forEach(annotation -> {                 // (3)
            Set<? extends Element> elements =
env.getElementsAnnotatedWith(annotation);                   // (4)
            elements.stream()
                    .filter(TypeElement.class::isInstance)  // (5)
                    .map(TypeElement.class::cast)           // (6)
                    .map(TypeElement::getQualifiedName)     // (7)
                    .map(name -> "Class " + name + " is annotated with " +
annotation.getQualifiedName())
                    .forEach(System.out::println);
        });
        return true;
    }
}
```

Задание

TEL-RAN
by Starta Institute

Создайте классы Human, Robot, Animal в пакете subject. Создайте собственную аннотацию @Lifeforms и примените ее к классам Human и Animal. Затем напишите класс ReflectionHelper, использующий Reflection API, который по названию пакета получит все классы, отмеченные аннотацией @Lifeforms. В методе main выведите названия всех найденных классов. Для каждого класса создайте по одному объекту, используя рефлексию и конструктор по умолчанию.

Добавьте класс BioRobot, который расширяет класс Robot. Укажите аннотацию @Lifeforms над классом BioRobot. Проверьте, что новый класс находится

AnnotationFindTask.zip

Проблема

TEL-RAN
by Starta Institute

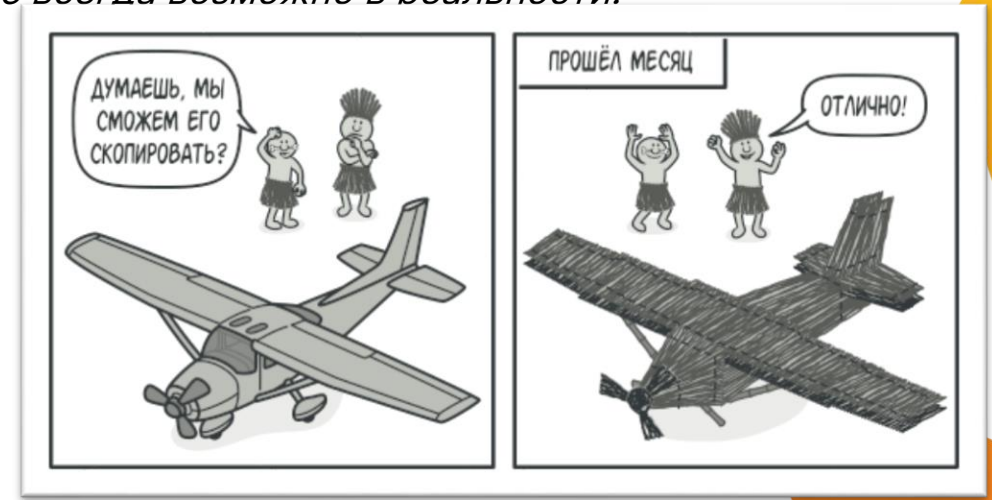
Прежде чем перейти к изучению *Spring* нужно подробнее познакомиться с некоторыми паттернами проектирования программы, которые применяются в *Spring*.

Проблема

TEL-RAN
by Starta Institute

У вас есть объект, который нужно скопировать. Как это сделать? Нужно создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый. Прекрасно! Но есть нюанс. Не каждый объект удастся скопировать таким образом, ведь часть его состояния может быть приватной, а значит – недоступной для остального кода программы.

Копирование «извне» не всегда возможно в реальности.

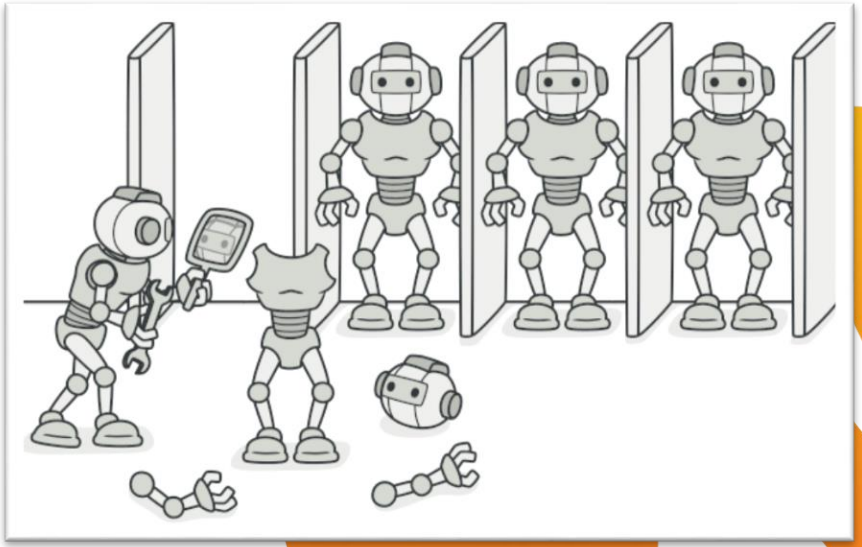


Неповторимый
Прототип

TEL-RAN
by Starta Institute

Прототип (prototype) – это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации. Паттерн Прототип поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Обычно такой интерфейс имеет всего один метод *clone()*.

Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта. Так получится скопировать даже приватные поля, так как большинство языков программирования разрешает доступ к приватным полям любого объекта текущего класса.



Неповторимый

Прототип

TEL-RAN
by Starta Institute

Объект, который копируют, называется *прототипом* (откуда и название паттерна). Когда объекты программы содержат сотни полей и тысячи возможных конфигураций, прототипы могут служить своеобразной альтернативой созданию подклассов.

В этом случае все возможные прототипы заготавливаются и настраиваются на этапе инициализации программы. Потом, когда программе нужен новый объект, она создаёт копию из приготовленного прототипа.

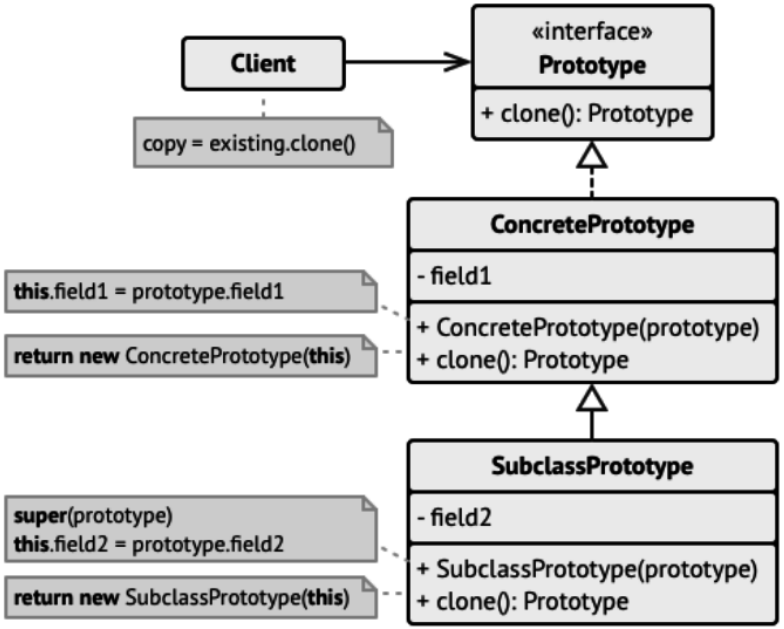
Неповторимый

Базовая реализация прототипа

TEL-RAN
by Starta Institute

3 **Клиент** создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

1 **Интерфейс прототипов** описывает операции клонирования. В большинстве случаев — это единственный метод `clone`.

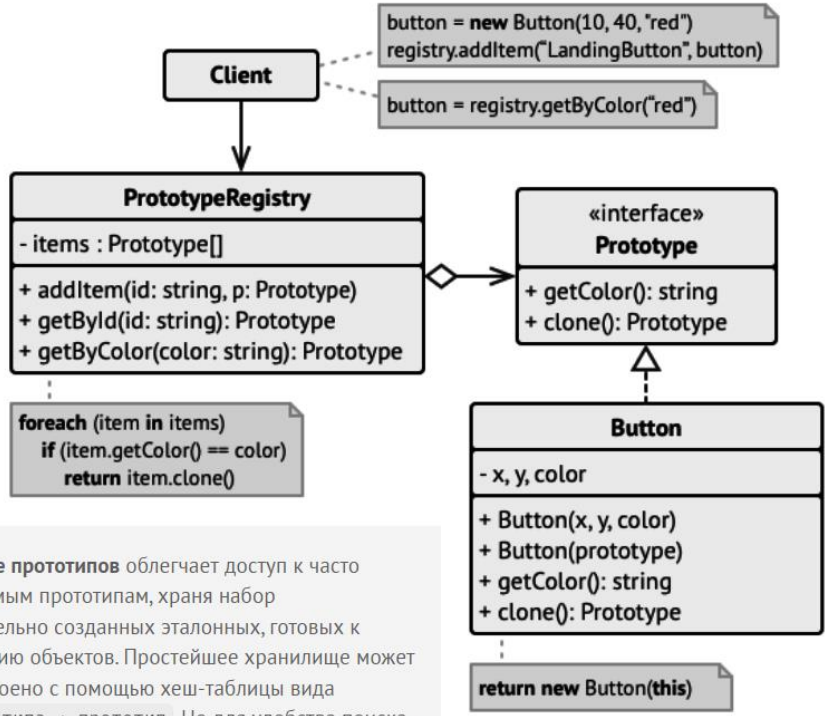


2 **Конкретный прототип** реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту. Например, клонирование связанных объектов, распутывание рекурсивных зависимостей и прочее.

Неповторимый

Реализация с общим хранилищем

TEL-RAN
by Starta Institute



1 **Хранилище прототипов** облегчает доступ к часто используемым прототипам, храня набор предварительно созданных эталонных, готовых к копированию объектов. Простейшее хранилище может быть построено с помощью хеш-таблицы вида `имя-прототипа → прототип`. Но для удобства поиска прототипы можно маркировать и другими критериями, а не только условным именем.

Неповторимый

Прототип в Java

TEL-RAN
by Starta Institute

В Java паттерн *Прототип* заложен на уровне класса *Object*.

```
protected native Object clone() throws CloneNotSupportedException;
```

Однако, вызов метода *clone()* у объекта Вашего класса приведёт к выбрасыванию *CloneNotSupportedException*. Чтобы такого не происходило, нужно, чтобы класс имплементировал маркерный интерфейс *Cloneable*.

```
package java.lang;
public interface Cloneable { }
```

Имплементация *Cloneable* подсказывает JVM, что можно клонировать объект простым копированием полей. Другой подход – переопределить метод *clone()* в Вашем классе.

Если не требуется клонировать разнотипные объекты в цикле, то лучше использовать *клонирующий конструктор*, т.к. это не требует обработки дополнительных исключений.

Задание

TEL-RAN
by Starta Institute

Создайте класс Color с полями red, green, blue и alpha (прозрачность), поля должны иметь геттеры и сеттеры, а также значения в диапазоне от 0 до 255. Создайте возможность клонирования цвета. Создайте класс ColorCache, в котором при создании будут храниться все цвета радуги и их названия. В ColorCache можно добавлять новые цвета, передавая новый цвет и его название явно или вызывая методы plusRed(), minusRed(), plusGreen(), minusGreen(), plusBlue(), minusBlue(), plusAlpha(), minusAlpha(). Эти метод принимают название оригинального цвета, который нужно изменить для получения нового, а также название результирующего цвета.

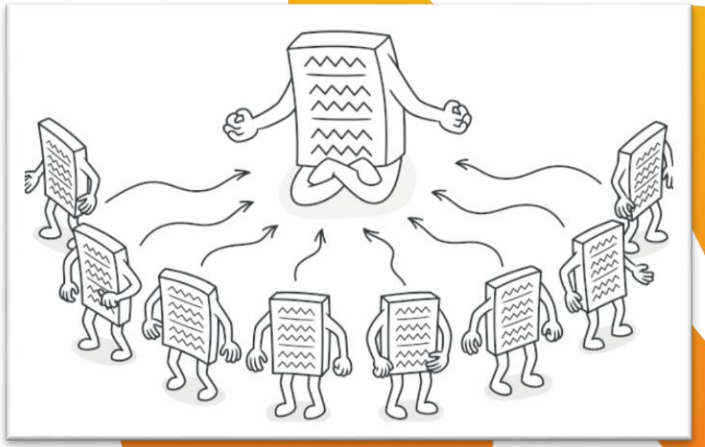
Добавьте в ColorCache несколько цветов и выведите содержимое кэша на экран.

Проблема

TEL-RAN
by Starta Institute

Предположим, Ваша программа принимает некоторые данные и сохраняет их в базу данных (БД) для длительного хранения. Данные могут поступать по разным каналам: от сетевых сервисов, от клиентских приложений, парсится при сканировании сети и т.д. За каждый канал получения данных отвечает отдельный класс, но каждый из этих классов вынужден передавать данные в БД. Подключение к БД – это длительный процесс. Если каждый класс будет устанавливать своё соединение с БД, то программа станет жутко тормозить.

Чтобы избежать этого, достаточно установить соединение с БД один раз и переиспользовать его для всех заинтересованных классов.



Но как классы поймут, что соединение уже установлено одним из них?

Одинокий волк

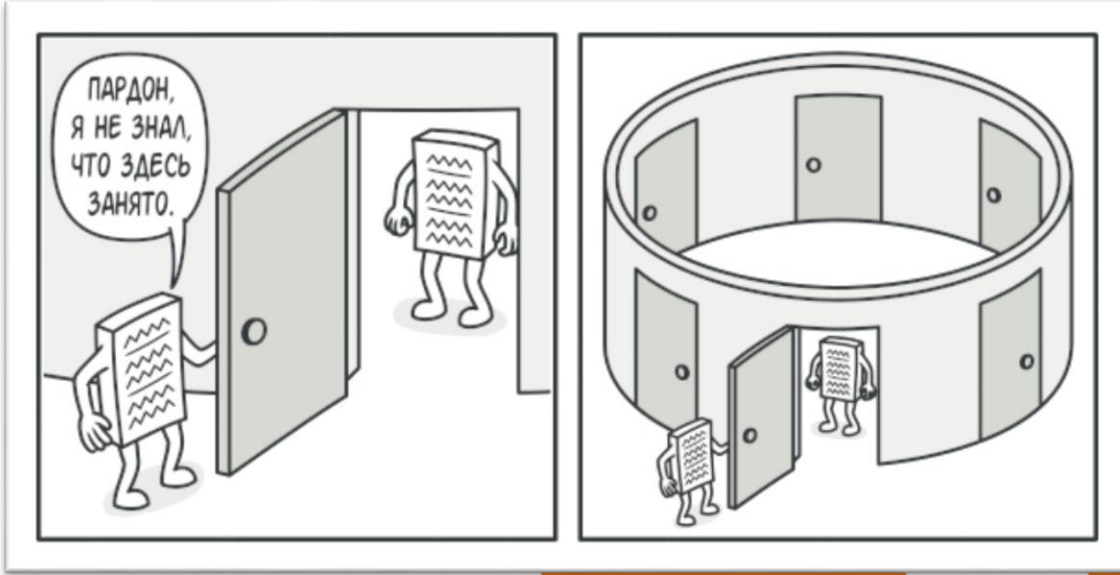
Одиночка

Одиночка (singleton) – это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Представьте, что вы создали объект, а через некоторое время пробуете создать ещё один.

В этом случае хотелось бы получить старый объект, вместо создания нового.

Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса всегда возвращает новый объект.



Одинокий волк

Одиночка в Java

TEL-RAN
by Starta Institute

Для предоставления доступа к общим объектам в других языках (например, C++) есть глобальные переменные, которые доступны из любой точки программы.

В Java для строгого выполнения принципа *инкапсуляции* умышленно отказались от глобальных переменных. Поэтому паттерн Одиночка только гарантирует создание одного экземпляра класса. Если у вас есть доступ к классу одиночки, значит, будет доступ и к статическому методу создания экземпляра. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

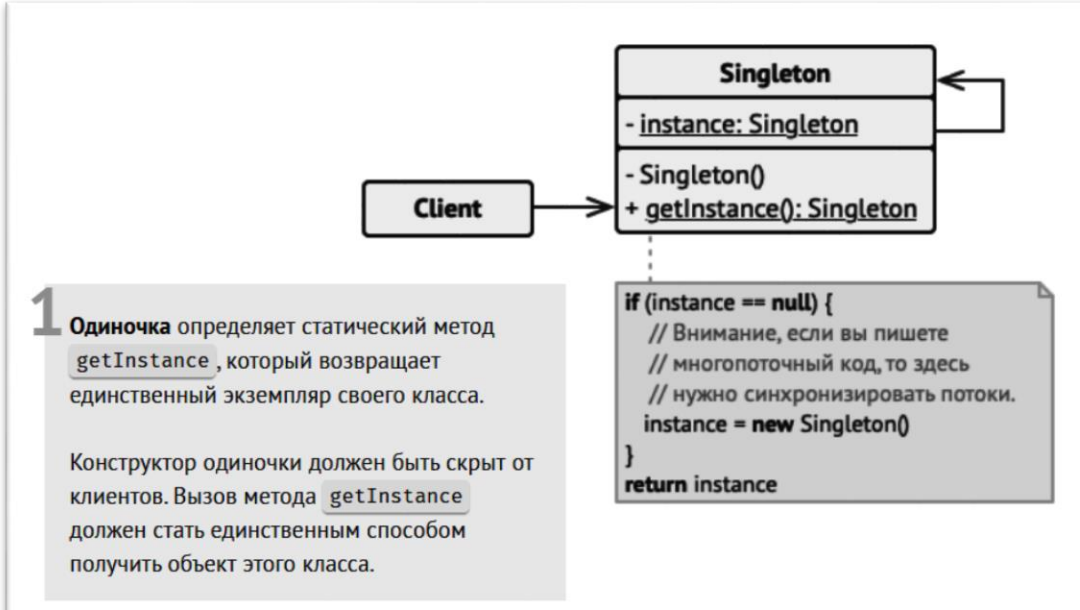
Одинокий волк

Реализация одиночки

TEL-RAN
by Starta Institute

Все реализации одиночки сводятся к тому, чтобы

1. Объявите статический создающий метод, который будет использоваться для получения одиночки.
2. Добавьте «ленивую инициализацию» (создание объекта при первом вызове метода) в создающий метод одиночки.
4. Сделайте конструктор класса приватным.
5. В клиентском коде замените вызовы конструктора одиночка вызовами его создающего метода.



Одинокий волк

Пример реализации одиночки

TEL-RAN
by Starta Institute

PatternSingletonExample.zip

Задание

TEL-RAN
by Starta Institute

Предположим, у вас есть приложение, и вы хотите, чтобы управление его настройками было централизованным. Создайте класс SettingsManager, использующий паттерн Singleton, который будет хранить и управлять всеми настройками приложения, такими как язык, тема оформления, уровень звука и другие параметры. Задача: Реализовать SettingsManager, который обеспечивает глобальный доступ к настройкам приложения и предотвращает создание более одного экземпляра.

TEL-RAN
by Starta Institute

3 Домашнее задание

Домашнее задание

TEL-RAN
by Starta Institute

1 У Вас есть небольшой фреймворк для обработки событий в игре. Вам нужно использовать Reflection API для динамического нахождения всех классов-обработчиков событий, отмеченных вашей аннотацией @EventHandler. Затем необходимо создать экземпляры этих классов и зарегистрируете их в системе обработки событий:

1.1 Создайте аннотацию @EventHandler.

1.2 Создайте несколько классов-обработчиков в пакете handlers и отметьте их аннотацией @EventHandler.

1.3 Напишите класс EventProcessor, который будет использовать Reflection API для нахождения и создания обработчиков событий.

1.4 Используйте вспомогательный класс ReflectionHelper из задания, выполненного на лекции, для поиска классов с аннотацией @EventHandler.

1.5 После запуска Вашей программы выведите объекты всех обработчиков в консоль.

Домашнее задание

TEL-RAN
by Starta Institute

2 Создайте класс Connection с полями id (уникальный), host, port и protocol и двух наследников: FastConnection и SlowConnection. FastConnection лёгкий, поэтому каждому из классов, запрашивающих экземпляр FastConnection, выдаётся новый экземпляр, имеющий те же характеристики, что и ранее созданный (паттерн prototype). Класс SlowConnection тяжёлый, поэтому он создаётся в единственном экземпляре и следует паттерну singleton.

2.1 Создайте класс Connector с методом getConnection. В методе реализуйте логику: если попытка соединиться занимает меньше 300 мс, то метод возвращает экземпляр FastConnection, в противном случае – экземпляр SlowConnection. Время ожидания подключения в классе просто сгенерируйте случайным образом один раз и сохраните в статическом поле. К getConnection могут обращаться несколько потоков.

2.2 Создайте класс Exchanger и 3 объекта: videoExchanger, audioExchanger и gameExchanger, использующие классы соединений. Каждый Exchanger работает с тем же хостом, что и остальные, но в своём потоке.

2.3 При запуске программы каждый Exchanger пытается получить соединение от класса Connector. Выведите id всех соединений в консоль поле получения.

Полезные ссылки

TEL-RAN
by Starta Institute

- Подробнее о паттернах проектирования https://refactoring.guru/ru/design-patterns/catalog

ЗАКЛЮЧЕНИЕ

TEL-RAN
by Starta Institute