

## Модуль 07 - Бассейн Java

### Reflection

Резюме: Сегодня вы будете разрабатывать собственные фреймворки, использующие механизм отражения.

Содержание

I Предисловие

II Инструкции

III Упражнение 00: Работа с классами

IV Упражнение 01: Аннотации – ИСТОЧНИК

V Упражнение 02: ORM

## Предисловие

Reflection — это мощный механизм, обеспечивающий работу фреймворков (таких как Spring или Hibernate). Знание принципов работы Java Reflection API гарантирует правильное использование различных технологий для реализации корпоративных систем.


Инструмент Reflection позволяет гибко использовать информацию о классе во время выполнения, а также динамически изменять состояние объектов, не используя эту информацию при написании исходного кода. Одной из возможностей отражения является изменение значений приватных полей извне. Тогда мы можем спросить, противоречит ли это принципу инкапсуляции, и ответ будет отрицательным :)





## Глава III

### Упражнение 00. Работа с классами

	Exercise 00
Work with Classes	
Turn-in directory : <i>ex00/</i>	
Files to turn in : Reflection-folder	
Allowed functions : All	

Теперь вам нужно реализовать проект Maven, который взаимодействует с классами вашего приложения.

Нам нужно создать как минимум два класса, каждый из которых имеет:

- private fields (supported types are String, Integer, Double, Boolean, Long)
- public methods
- an empty constructor
- a constructor with a parameter
- toString() method

В этой задаче вам не нужно реализовывать методы get/set.

Вновь созданные классы должны находиться в отдельном пакете классов (этот пакет может находиться в других пакетах).

Предположим, что в приложении есть классы User и Car.

Класс пользователя описан ниже:

```
public class User {
    private String firstName;
    private String lastName;
    private int height;
    public User() {
        this . firstName = "Default first name";
        this . lastName = "Default last name";
        this . height = 0;
    }
    public User(String firstName, String lastName, int height) {
        this . firstName = firstName;
        this . lastName = lastName;
        this . height = height;
    }
    public int grow(int value) {
        this . height += value;
        return height;
    }
    @Override
    public String toString() {
```

```

        return new StringJoiner(", ", User.class . getSimpleName() + "[, "]
        .add("firstName=" + firstName + "")
        .add("lastName=" + lastName + "")
        .add("height=" + height)
        .toString();
    }
}

```

Реализованное приложение должно работать следующим образом:

- Предоставьте информацию о классе в пакете классов.
- Разрешить пользователю создавать объекты указанного класса с определенными значениями полей.
- Показать информацию о созданном объекте класса.
- Вызывать методы класса.

Пример работы программы:

```

Classes:
User
Car
-----
Enter class name:
-> User
-----
fields :
    String firstName
    String lastName
    int height
methods:
    int grow(int)
-----
Let's create an object.
firstName:
-> UserName
lastName:
-> UserSurname
height:
-> 185
Object created: User[firstName='UserName', lastName='UserSurname', height=185]
-----
Enter name of the field for changing:
-> firstName
Enter String value:
-> Name
Object updated: User[firstName='Name', lastName='UserSurname', height=185]
-----
Enter name of the method for call:
-> grow(int)
Enter int value:
-> 10
Method returned:
195


```

- Если метод содержит более одного параметра, необходимо задать значения для каждого из них.

- Если метод имеет `тип void`, строка с информацией о возвращаемом значении не отображается.
- В сеансе программы возможно взаимодействие только с одним классом; одно поле его объекта может быть изменено, и один метод может быть вызван
- Вы можете использовать оператор `throws`.

## Глава IV

### Упражнение 01. Аннотации — ИСТОЧНИК

	Exercise 01
Annotations – SOURCE	
Turn-in directory : <i>ex01/</i>	
Files to turn in : Annotations-folder	
Allowed functions : All	

Аннотации позволяют хранить метаданные непосредственно в коде программы. Теперь ваша цель — реализовать класс `HtmlProcessor` (производный от `AbstractProcessor`), который обрабатывает классы со специальными аннотациями `@HtmlForm` и `@HtmlInput` и генерирует код HTML-формы внутри папки `target/classes` после выполнения команды `mvn clean compile`. Предположим, у нас есть класс `UserForm`:

```
@HtmlForm(fileName = "user_form.html", action = "/users", method = "post")
public class UserForm {
    @HtmlInput(type = "text", name = "first_name", placeholder = "Enter First Name")
    private String firstName;

    @HtmlInput(type = "text", name = "last_name", placeholder = "Enter Last Name")
    private String lastName;

    @HtmlInput(type = "password", name = "password", placeholder = "Enter Password")
    private String password;
}
```

Затем он должен быть использован в качестве основы для создания файла «`user_form.html`» со следующим содержанием:


```
<form action = "/users" method = "post">
<input type = "text" name = "first_name" placeholder = "Enter First Name">
<input type = "text" name = "last_name" placeholder = "Enter Last Name">
<input type = "password" name = "password" placeholder = "Enter Password">
<input type = "submit" value = "Send">
</form>
```

- Аннотации `@HtmlForm` и `@HtmlInput` должны быть доступны только во время компиляции.
- Структура проекта на усмотрение разработчика
- Для корректной обработки аннотаций рекомендуется использовать специальные настройки плагина `mavencompiler-plugin` и зависимость автосервиса от `com.google.auto.service`.



## Глава V

### Упражнение 02: ORM

	Exercise 02
	ORM
	Turn-in directory : <i>ex02/</i>
	Files to turn in : ORM-folder
	Allowed functions : All

Ранее мы упоминали, что структура Hibernate ORM для баз данных основана на отражении.

Концепция ORM позволяет автоматически преобразовывать реляционные ссылки в объектно-ориентированные. Такой подход делает приложение полностью независимым от СУБД. Вам нужно реализовать тривиальную версию такой структуры ORM.

Предположим, у нас есть набор классов моделей. Каждый класс не содержит зависимостей от других классов, а его поля могут принимать только следующие типы значений: String, Integer, Double, Boolean, Long. Укажем некий набор аннотаций для класса и его членов, например, класс пользователя:

```
@OrmEntity(table = "simple_user")
public class User {
    @OrmColumnId
    private Long id;
    @OrmColumn(name = "first_name", length = 10)
    private String firstName;
    @OrmColumn(name = "first_name", length = 10)
    private String lastName;
    @OrmColumn(name = "age")
    private Integer age;

    // setters /getters
}
```

Разработанный вами класс `OrmManager` должен генерировать и выполнять соответствующий код SQL при инициализации всех классов, помеченных аннотацией `@OrmEntity`. Этот код будет содержать команду `CREATE TABLE` для создания таблицы с именем, указанным в аннотации. Каждое поле класса, помеченное аннотацией `@OrmColumn`, становится столбцом в этой таблице. Поле, помеченное аннотацией `@OrmColumnId`, указывает на необходимость создания идентификатора автоинкремента. `OrmManager` также должен поддерживать следующие набор операций (для каждой из них также генерируется соответствующий код SQL в Runtime):

```
public void save(Object entity)
public void update(Object entity)
public <T> T findById(Long id, Class<T> aClass)
```

`OrmManager` должен обеспечивать вывод сгенерированного SQL на консоль во время выполнения.

- При инициализации `OrmManager` должен удалить созданные таблицы.
- Метод обновления должен заменять значения в столбцах, указанных в объекте, даже если значение поля объекта равно нулю.