образование ПОЛИТЕХ одноклассники экосистема

Reflection. Unit тестирование

Галкин Александр Сергеевич

2022 г.

Оглавление

Рефлексия

Рефлексия

- Механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов
- Позволяет исследовать информацию о полях, методах и конструкторах классов и выполнять операции над ними
- По сути - некоторый аналог динамической типизации

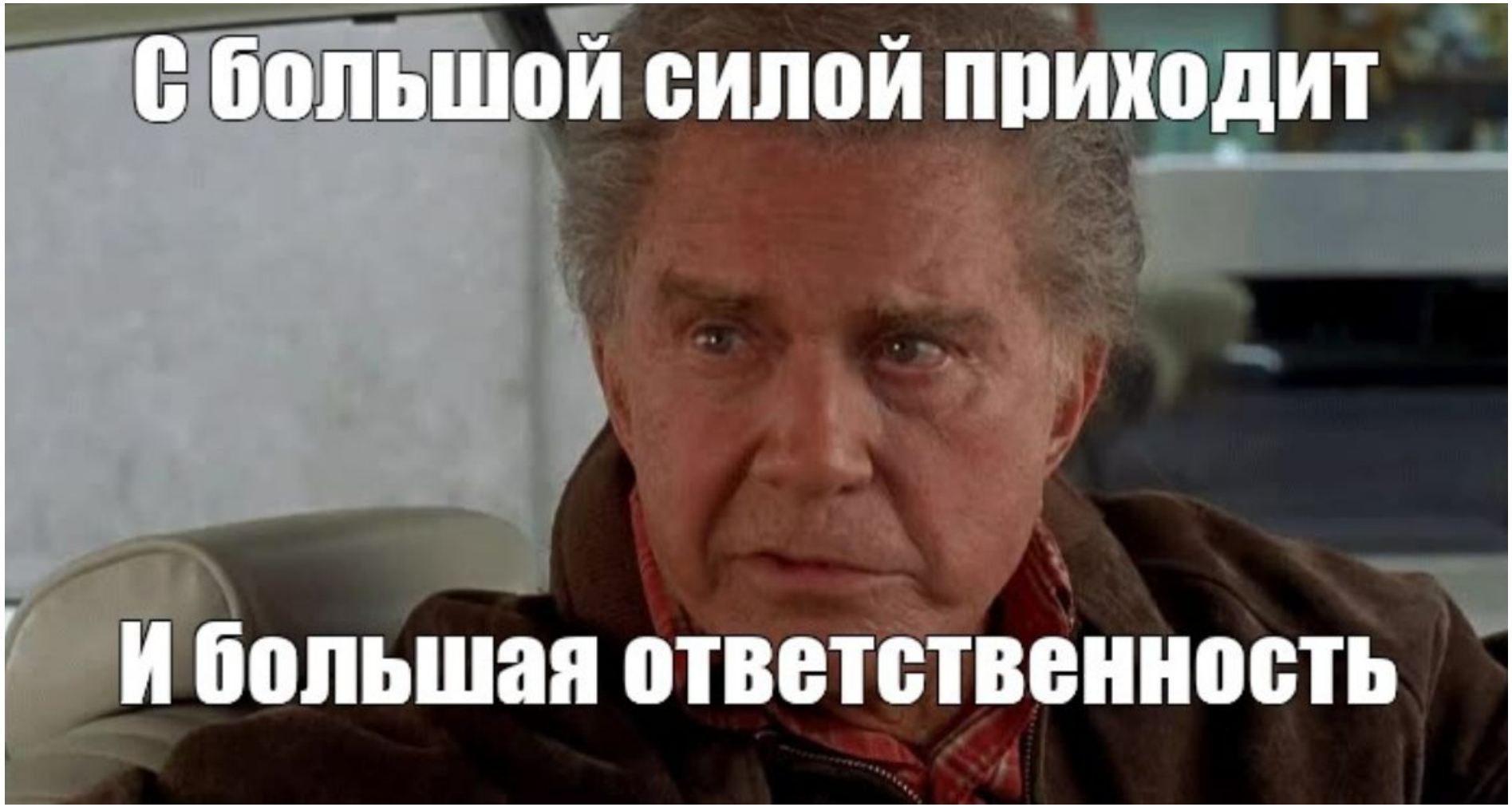
Рефлексия. Возможности

- Определить класс объекта
- Получить информацию о модификаторах класса, полях, методах, конструкторах и суперклассах
- Выяснить, какие константы и методы принадлежат интерфейсу
- Создать экземпляр класса
- Вызвать метод объекта
- Создать новый массив, размер и тип компонентов которого неизвестны

Рефлексия. Примеры использования

- Создание экземпляров пользовательских классов
- Среда разработки
- Отладчик
- Тестирование

Рефлексия. Предупреждение



Рефлексия. Недостатки

- Накладные расходы (динамическая типизация, не используются оптимизаторы JVM)
- Ограничение безопасности (требует доступ к runtime)
- Взаимодействие с внутренними данными
- Нарушает абстракции

Class

Class. Получение

- *Class<? extends String[]> clazz1 = args.getClass();*
- *Class<? extends String[]> clazz2 = String[].class;*
- *Class<?> clazz3 = Class.forName("[Ljava.lang.String;");*
- *Class<?> clazz4 = clazz1.getSuperclass();*
- *Class<?>[] clazz5 = Character.class.getClasses();*
- *Class<?>[] clazz6 = Character.class.getDeclaredClasses();*

Class. Модификаторы

- Доступ
- Требования переопределения
- Возможность модификации
- Аннотации
- *java.lang.reflect.Modifier*
- *clazz.getModifiers();*

Class. Что можно найти

- Поля
- Методы
- Конструкторы

- Перечисление всех или поиск конкретного
- Только в классе или во всех предках

Member

- Интерфейс
- *java.lang.reflect.Member*
- Наследники
  - *java.lang.reflect.Field*
  - *java.lang.reflect.Method*
  - *java.lang.reflect.Constructor*
- Только в классе или во всех предках

Field

Fields. Получение

Class Methods for Locating Fields

Class API	List of members?	Inherited members?	Private members?
getDeclaredField()	no	no	yes
getField()	no	yes	no
getDeclaredFields()	yes	no	yes
getFields()	yes	yes	no

Fields. Модификаторы

- Доступ
- Специфические для поля отвечающие за поведение в runtime
- Возможность модификации
- Аннотации

Fields. Особенности

- Поля могут быть объектами или примитивами
- Можно получить значение
- Можно установить новое значение (даже на ***final***)
- *bool.setAccessible(true);* - работает не всегда

Methods

Methods. Получение

Class Methods for Locating Methods

Class API	List of members?	Inherited members?	Private members?
getDeclaredMethod()	no	no	yes
getMethod()	no	yes	no
getDeclaredMethods()	yes	no	yes
getMethods()	yes	yes	no

Constructors. Получение

Class Methods for Locating Constructors

Class API	List of members?	Inherited members?	Private members?
getDeclaredConstructor()	no	N/A[1]	yes
getConstructor()	no	N/A[1]	no
getDeclaredConstructors()	yes	N/A[1]	yes
getConstructors()	yes	N/A[1]	no

Methods. Состав

- Модификаторы
- Название
- Параметры
- Возвращаемые значения
- Список исключений
- Аннотации

Annotations

Annotations

Annotations. Определение

- Специальные плашки для классов, методов, полей, которые начинаются с @
- Содержат в себе некоторую метаинформацию о программе
- Прямо не влияют на работу кода, который они аннотируют

- Нужны для
  - Информация для компилятора
  - Обработка во время компиляции и во время развертывания
  - Обработка во время выполнения

Пример определения

```java
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface AnnotationWithParams {
    String value();
    int number() default 0;
    double[] numbers();
    Gender gender() default Gender.FEMALE;
    Class<? extends Number> clazz() default Integer.class;
}
```

Пример использования

```
@AnnotationWithParams(value = "Hello",
        numbers = {1.0, 2.0},
        clazz = Double.class)
class MyClass {
```

Популярные аннотации

- *@Deprecated* — Устаревший код
- *@Override* — Переопределенный метод
- *@SuppressWarnings* — Подавление предупреждений
- *@FunctionalInterface*

Аннотации для других аннотаций

- *@Retention* — Время жизни аннотации (*SOURCE*, *CLASS*, *RUNTIME*)
- *@Documented* — Информация из аннотаций попадает в документацию
- *@Target* — Область применения аннотации (см класс *ElementType*)

Аннотации для других аннотаций

Пример работы с аннотациями

```
for (Method method : clazz.getMethods()) {
    if (method.isAnnotationPresent(MyTest.class)) {
        try {
            method.invoke( obj: null);
        } catch (Exception e) {
            System.out.println("Test " + method +" failed: " + e);
        }
    }
}
```

Дополнительные материалы

https://docs.oracle.com/javase/tutorial/reflect/index.html

Тестирование

TTD

1. Есть ожидаемый функционал
2. Пишется тест
3. Подгоняется API
4. Пишется код для прохождения теста
5. Код подгоняется под стандарты

JUnit. Введение

- 1 класс — 1 тестовый класс
- 1 тест — 1 метод с аннотацией `@Test`
- Для выполнения каждого теста создается уникальный экземпляр тестового класса
- В идеале покрывать каждую строчку кода
- Обычно покрывают каждое ветвление
- Минимальное покрытие — особые случаи и самый частый случай
- Можно проверять результаты, исключения, время выполнения

JUnit. Основные аннотации

- `@Before` (`BeforeEach`) — выполняется перед каждым тестом
- `@BeforeClass` (`BeforeAll`) — выполняется один раз перед загрузкой тестового класса
- `@After` (`AfterEach`) — выполняется после каждого теста
- `@AfterClass` (`AfterAll`) — выполняется один раз после выполнения всех тестов
- `@Test(expected = NullPointerException.class, timeout = 1000)` — можно задать ожидаемую ошибку и время исполнения

JUnit. Правила

- `@Rule` — позволяют дополнительно задать некоторые правила для тестов
- `TemporaryFolder` — можно создавать временные файлы, которые будут удаляться
- `Timeout` — позволяет создать глобальный таймаут
- `ExpectedException` — удобно сравнивать исключения

JUnit. Запуск

- `@RunWith(Enclosed.class)` — запускает последовательно внутренние классы
- `@Category(Unit.class)` — позволяет разделять запуски по категориям (`@Categories.IncludeCategory(Unit.class)`)
- `@RunWith(Parameterized.class)` — запуск с параметрами через конструктор

JUnit 5

- **JUnit Platform** — фундаментальная основа для запуска на **JVM** фреймворков для тестирования
- **JUnit Jupiter** — проект предоставляет новые возможности для написания тестов и создания собственных расширений. Умеет запускать тесты на платформе
- **JUnit Vintage** — поддержка легаси для *JUnit 3* и *JUnit 4*

JUnit 5. Особенности

- Тесты теперь не обязательно `public`
- Группировка нескольких ассертов с помощью assertAll
- Работа с исключениями assertThrows
- `@Test` — исключительно маркер
- `@Nested` — вместо `@RunWith(Enclosed.class)`
- `@TestInstance(TestInstance.Lifecycle.PER_METHOD)` — как часто надо создавать экземпляры класса
- `@ParameterizedTest` — запуск теста с параметрами `@MethodSource` `@ValueSource`
- https://junit.org/junit5/

Заглушки. Тестирование логики

- Код существует не в вакууме
- Методы в классах работают с другими сложными объектами
- Для тестирования функционала нам нужно простое поведение сложных объектов
- Заглушка (**stub**) — эмулирование сложного объекта, чтобы он вел себя как нам нужно
- Можно самостоятельно плодить наследников для эмулирования простого поведения
- Можно воспользоваться фрэймворками

Mockito

- Библиотека по созданию заглушек в 1 строку
- Можно задать поведение каждого метода в зависимости от аргументов
- Можно создать моки на синглтоны и прочие классы без конструкторов
- Основной класс для работы **Mockito**

Mockito. Создание заглушек

- `mock(DragonSlayingStrategy.class);`
- `@Mock private StealingMethod method`
- Любой метод моков не окажет никакого влияния на саму заглушку
- Таким образом нельзя делать заглушки для **Enum**, **final** классов и переопределять **final** методы
- Любые методы по-умолчанию возвращают **null** или пустые коллекции

Mockito. Шпионы

- Заглушка + реальный объект
- Мы хотим протестировать логику метода вместе с “важными” и “неважными” методами этого класса
- `spy(new HalflingThief(method));`
- Методы шпиона могут влиять на внутреннее состояние
- По-умолчанию, метод пытается исполнить свое реальное тело

Mockito. Эмуляция

- `when(dataService.getAllData()).thenReturn(data);` — сначала вызывается метод, потом идет подмена. Предпочтительнее, так как есть проверка по типам
- `doReturn(data).when(dataService).getAllData();` — Нет проверки по типам, но часто используется для шпионов, когда нам не нужен реальный вызов метода. Также используется при void методах

Mockito. Эмуляция при заданных параметров

- `when(dataService.getDataById(anyInt()))`
  `.thenReturn("dataItem");` — любой аргумент
- `when(dataService.getDataById(eq(1)))`
  `.thenReturn("dataItem");` — заданный аргумент
- `when(dataService.getDataByString(`
  `argThat(arg -> arg == null || arg.length() < 2)))`
  `.thenReturn("dataItem");`

— фильтрация аргумента

Mockito. Сложные эмуляции

- `when(dataService.getDataByString(any()))`
  `.thenThrow(NumberFormatException.class);` — исключения
- `when(dataService.getDataByStrings(any()))`
  `.thenAnswer(invocation ->`
  `invocation.<List<String>>getArgument(0).stream()` —
  получение параметров, обработка, возвращение результатов
- `when(dataService.getDataByString("a"))`
  `.thenReturn("valueA1","valueA2")`
  `.thenThrow(NumberFormatException.class)` —
  последовательное выполнение

Mockito. Слежение за вызовами

- `verify(mockedVisitor).visitCommander(eq(unit));` — однократный вызов
- `verify(mockedVisitor, times(2)).visitCommander(eq(unit));` — заданное количество вызовов
- `verify(mockedVisitor, never()).visitCommander(eq(unit));` — ни одного вызова
- `inOrder(mockedVisitor).verify(mockedVisitor).visitCommander(eq(unit));` — когда важен порядок

Спасибо!

образование

ПОЛИТЕХ

одноклассники
экосистема