Введение в программирование на Java

Лекция 18. Метапрограммирование. Reflection API.

Виталий Олегович Афанасьев

09 июня 2025

Метапрограммирование (1)

Метапрограммирование — программы работают с программами как с данными.

Метапрограммирование (2)

Средства метапрограммирования:

- Генераторы кода
- Интроспекция / Reflection
- Динамическое выполнение кода (eval)
- ...

```java
package edu.hse.example;

public class Wrapper {
    public final int value;
    private String str;
    public Wrapper(int value, String str) {
        this.value = value;
        this.str = str;
    }
    public String getStr() {
        return str;
    }
    public void setStr(String str) {
        this.str = str;
    }
    public void foo(boolean printFirst) {
        bar(printFirst);
    }
    private void bar(boolean printFirst) {
        System.out.println(printFirst ? value : str);
    }
}
```

Оператор `instanceof`

Оператор `instanceof` позволяет проверить тип объекта **во время выполнения**.

```
1  Object wrapper = new Wrapper(42, "Lorem Ipsum");
2
3
4
5
6
7
8
9
10
11
12
```

Данный оператор нужно применять **с умом**. С ним очень легко перейти от объектно-ориентированного кода к процедурному.

Оператор `instanceof`

Оператор `instanceof` позволяет проверить тип объекта **во время выполнения**.

```
1  Object wrapper = new Wrapper(42, "Lorem Ipsum");
2  Object obj = new Object();
3
4
5
6
7
8
9
10
11
12
```

Данный оператор нужно применять **с умом**. С ним очень легко перейти от объектно-ориентированного кода к процедурному.

Оператор `instanceof`

Оператор `instanceof` позволяет проверить тип объекта **во время выполнения**.

```
1  Object wrapper = new Wrapper(42, "Lorem Ipsum");
2  Object obj = new Object();
3  Wrapper nullWrapper = null;
4
5
6
7
8
9
10
11
12
```

Данный оператор нужно применять **с умом**. С ним очень легко перейти от объектно-ориентированного кода к процедурному.

Оператор `instanceof`

Оператор `instanceof` позволяет проверить тип объекта **во время выполнения**.

```
1  Object wrapper = new Wrapper(42, "Lorem Ipsum");
2  Object obj = new Object();
3  Wrapper nullWrapper = null;
4
5  boolean wrapperIsWrapper = wrapper instanceof Wrapper; // true
6
7
8
9
10
11
12
```

Данный оператор нужно применять **с умом**. С ним очень легко перейти от объектно-ориентированного кода к процедурному.

Оператор `instanceof`

Оператор `instanceof` позволяет проверить тип объекта **во время выполнения**.

```
1  Object wrapper = new Wrapper(42, "Lorem Ipsum");
2  Object obj = new Object();
3  Wrapper nullWrapper = null;
4
5  boolean wrapperIsWrapper = wrapper instanceof Wrapper; // true
6  boolean objIsObject = obj instanceof Object; // true
7
8
9
10
11
12
```

Данный оператор нужно применять **с умом**. С ним очень легко перейти от объектно-ориентированного кода к процедурному.

Оператор `instanceof`

Оператор `instanceof` позволяет проверить тип объекта **во время выполнения**.

```
1  Object wrapper = new Wrapper(42, "Lorem Ipsum");
2  Object obj = new Object();
3  Wrapper nullWrapper = null;
4
5  boolean wrapperIsWrapper = wrapper instanceof Wrapper; // true
6  boolean objIsObject = obj instanceof Object; // true
7
8  boolean objIsWrapper = obj instanceof Wrapper; // false
9
10
11
12
```

Данный оператор нужно применять **с умом**. С ним очень легко перейти от объектно-ориентированного кода к процедурному.

Оператор `instanceof`

Оператор `instanceof` позволяет проверить тип объекта **во время выполнения**.

```
1  Object wrapper = new Wrapper(42, "Lorem Ipsum");
2  Object obj = new Object();
3  Wrapper nullWrapper = null;
4
5  boolean wrapperIsWrapper = wrapper instanceof Wrapper; // true
6  boolean objIsObject = obj instanceof Object; // true
7
8  boolean objIsWrapper = obj instanceof Wrapper; // false
9  boolean wrapperIsObject = wrapper instanceof Object; // true
10
11
12
```

Данный оператор нужно применять **с умом**. С ним очень легко перейти от объектно-ориентированного кода к процедурному.

Оператор `instanceof`

Оператор `instanceof` позволяет проверить тип объекта **во время выполнения**.

```
1  Object wrapper = new Wrapper(42, "Lorem Ipsum");
2  Object obj = new Object();
3  Wrapper nullWrapper = null;
4
5  boolean wrapperIsWrapper = wrapper instanceof Wrapper; // true
6  boolean objIsObject = obj instanceof Object; // true
7
8  boolean objIsWrapper = obj instanceof Wrapper; // false
9  boolean wrapperIsObject = wrapper instanceof Object; // true
10
11 boolean nullIsWrapper = nullWrapper instanceof Wrapper; // false
12
```

Данный оператор нужно применять **с умом**. С ним очень легко перейти от объектно-ориентированного кода к процедурному.

Оператор `instanceof`

Оператор `instanceof` позволяет проверить тип объекта **во время выполнения**.

```
1  Object wrapper = new Wrapper(42, "Lorem Ipsum");
2  Object obj = new Object();
3  Wrapper nullWrapper = null;
4
5  boolean wrapperIsWrapper = wrapper instanceof Wrapper; // true
6  boolean objIsObject = obj instanceof Object; // true
7
8  boolean objIsWrapper = obj instanceof Wrapper; // false
9  boolean wrapperIsObject = wrapper instanceof Object; // true
10
11 boolean nullIsWrapper = nullWrapper instanceof Wrapper; // false
12 boolean nullIsObject = nullWrapper instanceof Object; // false
```

Данный оператор нужно применять **с умом**. С ним очень легко перейти от объектно-ориентированного кода к процедурному.

Оператор `switch` для проверки типов

Аналогичную проверку можно сделать и при помощи оператора `switch`.

```
1 Object obj = ...;
2 switch (obj) {
3     case Integer i -> System.out.println("Is integer " + i);
4     case String s -> System.out.println("Is string '" + s + "'");
5     case Wrapper w -> System.out.println("Is wrapper");
6     case null, default -> System.out.println("Something else");
7 }
```

Оператор **должен** покрывать все возможные варианты (либо иметь ветвь `default`).

Reflection API:
Получение информации о классе

```
1  Class<Wrapper> wrapperClass = Wrapper.class;
2
3
4
5
6
7
8
9
10
11
12
```

Reflection API:
Получение информации о классе

```
1  Class<Wrapper> wrapperClass = Wrapper.class;
2
3  System.out.println(wrapperClass.getName()); // edu.hse.example.Wrapper
4
5
6
7
8
9
10
11
12
```

Reflection API:
Получение информации о классе

```
1  Class<Wrapper> wrapperClass = Wrapper.class;
2
3  System.out.println(wrapperClass.getName()); // edu.hse.example.Wrapper
4  System.out.println(wrapperClass.getSimpleName()); // Wrapper
5
6
7
8
9
10
11
12
```

Reflection API:
Получение информации о классе

```
1  Class<Wrapper> wrapperClass = Wrapper.class;
2
3  System.out.println(wrapperClass.getName()); // edu.hse.example.Wrapper
4  System.out.println(wrapperClass.getSimpleName()); // Wrapper
5
6  Field[] publicFields = wrapperClass.getFields(); // [ value ]
7
8
9
10
11
12
```

Reflection API:
Получение информации о классе

```
1  Class<Wrapper> wrapperClass = Wrapper.class;
2
3  System.out.println(wrapperClass.getName()); // edu.hse.example.Wrapper
4  System.out.println(wrapperClass.getSimpleName()); // Wrapper
5
6  Field[] publicFields = wrapperClass.getFields(); // [ value ]
7  Field[] allFields = wrapperClass.getDeclaredFields(); // [ value, str ]
8
9
10
11
12
```

Reflection API:
Получение информации о классе

```
1  Class<Wrapper> wrapperClass = Wrapper.class;
2
3  System.out.println(wrapperClass.getName()); // edu.hse.example.Wrapper
4  System.out.println(wrapperClass.getSimpleName()); // Wrapper
5
6  Field[] publicFields = wrapperClass.getFields(); // [ value ]
7  Field[] allFields = wrapperClass.getDeclaredFields(); // [ value, str ]
8
9  Method[] publicMethods = wrapperClass.getMethods();
10
11
12
```

Reflection API:
Получение информации о классе

```
1  Class<Wrapper> wrapperClass = Wrapper.class;
2
3  System.out.println(wrapperClass.getName()); // edu.hse.example.Wrapper
4  System.out.println(wrapperClass.getSimpleName()); // Wrapper
5
6  Field[] publicFields = wrapperClass.getFields(); // [ value ]
7  Field[] allFields = wrapperClass.getDeclaredFields(); // [ value, str ]
8
9  Method[] publicMethods = wrapperClass.getMethods();
10      // [ getStr, setStr, foo, equals, hashCode, toString, ... ]
11
12
```

Reflection API:
Получение информации о классе

```
1  Class<Wrapper> wrapperClass = Wrapper.class;
2
3  System.out.println(wrapperClass.getName()); // edu.hse.example.Wrapper
4  System.out.println(wrapperClass.getSimpleName()); // Wrapper
5
6  Field[] publicFields = wrapperClass.getFields(); // [ value ]
7  Field[] allFields = wrapperClass.getDeclaredFields(); // [ value, str ]
8
9  Method[] publicMethods = wrapperClass.getMethods();
10      // [ getStr, setStr, foo, equals, hashCode, toString, ... ]
11 Method[] allMethods = wrapperClass.getDeclaredMethods();
12
```

Reflection API:
Получение информации о классе

```
1  Class<Wrapper> wrapperClass = Wrapper.class;
2
3  System.out.println(wrapperClass.getName()); // edu.hse.example.Wrapper
4  System.out.println(wrapperClass.getSimpleName()); // Wrapper
5
6  Field[] publicFields = wrapperClass.getFields(); // [ value ]
7  Field[] allFields = wrapperClass.getDeclaredFields(); // [ value, str ]
8
9  Method[] publicMethods = wrapperClass.getMethods();
10      // [ getStr, setStr, foo, equals, hashCode, toString, ... ]
11 Method[] allMethods = wrapperClass.getDeclaredMethods();
12      // [ getStr, setStr, foo, bar, equals, hashCode, toString, ... ]
```

Reflection API:
Получение информации о поле

```
1 try {
2     Field valueField = Wrapper.class.getField("value");
3     System.out.println(valueField.getName()); // value
4     System.out.println(valueField.getType().getName()); // int
5 } catch (NoSuchFieldException e) {
6     ...
7 }
```

Reflection API:
Получение значения поля

```
1 Wrapper wrapper = new Wrapper(42, "Lorem Ipsum");
2 try {
3     Field valueField = Wrapper.class.getField("value");
4     Object value = valueField.get(wrapper);
5     System.out.println(value); // 42
6 } catch (NoSuchFieldException | IllegalAccessException e) {
7     ...
8 }
```

Reflection API:
Получение значения приватного поля (1)

```
1 Wrapper wrapper = new Wrapper(42, "Lorem Ipsum");
2 try {
3     Field strField = Wrapper.class.getDeclaredField("str");
4     String str = strField.get(wrapper);
5             // IllegalAccessException: 'str' is private
6     System.out.println(str);
7 } catch (NoSuchFieldException | IllegalAccessException e) {
8     ...
9 }
```

Reflection API:
Получение значения приватного поля (2)

```
1 Wrapper wrapper = new Wrapper(42, "Lorem Ipsum");
2 try {
3     Field strField = Wrapper.class.getDeclaredField("str");
4     strField.setAccessible(true);
5     String str = strField.get(wrapper);
6     System.out.println(str); // Lorem Ipsum
7 } catch (NoSuchFieldException | IllegalAccessException e) {
8     ...
9 }
```

Reflection API:
Получение информации о методе

```
1 try {
2     Method method = Wrapper.class.getMethod("foo", boolean.class);
3     System.out.println(method.getName()); // foo
4     System.out.println(method.getReturnType().getName()); // void
5     Class<?>[] paramTypes = method.getParameterTypes(); // [ boolean ]
6 } catch (NoSuchMethodException e) {
7     ...
8 }
```

Reflection API: Вызов метода

```
1 Wrapper wrapper = new Wrapper(42, "Lorem ipsum");
2 try {
3     Method method = Wrapper.class.getMethod("foo", boolean.class);
4     method.invoke(wrapper, true); // 42
5 } catch (NoSuchMethodException | IllegalAccessException |
       InvocationTargetException e) {
6     ...
7 }
```

Reflection API: Вызов приватного метода (1)

```
1 Wrapper wrapper = new Wrapper(42, "Lorem ipsum");
2 try {
3   Method method = Wrapper.class.getDeclaredMethod("bar", boolean.class);
4   method.invoke(wrapper, true);
5       // IllegalAccessException: 'bar' is private
6 } catch (NoSuchMethodException | IllegalAccessException |
      InvocationTargetException e) {
7     ...
8 }
```

Reflection API:
Вызов приватного метода (2)

```
1 Wrapper wrapper = new Wrapper(42, "Lorem ipsum");
2 try {
3   Method method = Wrapper.class.getDeclaredMethod("bar", boolean.class);
4   method.setAccessible(true);
5   method.invoke(wrapper, true); // 42
6 } catch (NoSuchMethodException | IllegalAccessException |
      InvocationTargetException e) {
7     ...
8 }
```

Reflection API: Generics (1)

```
1 public class Generic<T, U extends Wrapper> {
2     public T first;
3     public U second;
4     public List<Integer> third;
5 }
```

Reflection API: Generics (2)

Информация о дженериках недоступна через Reflection API, т.к. происходит type erasure.

```
1 Field[] publicFields = Generic.class.getFields();
2 for (Field publicField : publicFields) {
3     String type = publicField.getType().getSimpleName();
4     String name = publicField.getName();
5     System.out.println(type + " " + name);
6 }
```

Вывод:

```
Object first
Wrapper second
List third
```

Аннотации

Java позволяет создавать собственные аннотации:

```
1 public @interface SampleAnnot {
2 }
```

Аннотациями можно помечать различные сущности в коде:

```
1  @SampleAnnot
2  public class SampleClass {
3      @SampleAnnot
4      private final int field;
5
6      @SampleAnnot
7      public void foo(@SampleAnnot int param) {
8          ...
9      }
10 }
```

Аннотации: Параметры (1)

Аннотации могут иметь параметры (задаются как методы интерфейса).

При помощи ключевого слова `default` можно задать параметрам значение по умолчанию.

```
1 public @interface SampleAnnot {
2     String str();
3     int val() default 42;
4 }
```

```
1 @SampleAnnot(str = "Lorem ipsum", val = 123)
2 public void foo() {
3     ...
4 }
5 @SampleAnnot(str = "Lorem ipsum")
6 public void bar() {
7     ...
8 }
```

Аннотации: Параметры (2)

Параметры могут быть заданы только следующими значениями:

- Константами примитивных типов
- Строковыми константами
- Объектами типа `Class` (в формате `Smth.class`)
- Элементами `enum`
- Другими аннотациями
- Константными массивами из всего перечисленного

Аннотации: Параметры (3)

Если у аннотации задан только один параметр с именем `value`, то для его задания можно применять более короткую запись.

```
1 public @interface SampleAnnot {
2     String value();
3 }
```

```
1 @SampleAnnot("Lorem ipsum")
2 public void foo() {
3     ...
4 }
```

Аннотации: Target

При помощи аннотации Target (пакет java.lang.annotation) можно задать сущности, которые можно аннотировать конкретной аннотацией.

```
1 @Target({ElementType.FIELD, ElementType.METHOD})
2 public @interface SampleAnnot {
3 }
```

Возможные значения:

- TYPE
- FIELD
- METHOD
- PARAMETER
- CONSTRUCTOR
- LOCAL_VARIABLE
- ANNOTATION_TYPE
- PACKAGE

Аннотации: Retention

Аннотация `Retention` позволяет указать, как долго будет сохраняться информация об аннотации на сущностях.

```
1 // @SampleAnnot удаляется после компиляции
2 @Retention(RetentionPolicy.SOURCE)
3 public @interface SampleAnnot {
4 }
```

```
1 // @SampleAnnot сохраняется после компиляции,
2 //   но не доступна через Reflection API
3 @Retention(RetentionPolicy.CLASS)
4 public @interface SampleAnnot {
5 }
```

```
1 // @SampleAnnot сохраняется после компиляции
2 //   и доступна через Reflection API
3 @Retention(RetentionPolicy.RUNTIME)
4 public @interface SampleAnnot {
5 }
```

Reflection API:
Чтение информации об аннотациях

При `RetentionPolicy.RUNTIME` информацию об аннотациях можно получить при помощи Reflection API во время работы программы.

```java
public class Annotated {
    @SampleAnnot("Some text")
    public int first;

    public int second;
}
```

```java
try {
  Field firstField = Annotated.class.getField("first");
  Field secondField = Annotated.class.getField("second");
  Annotation[] annots1 = firstField.getAnnotations();// [ @SampleAnnot ]
  Annotation[] annots2 = secondField.getAnnotations(); // []
  SampleAnnot sample = firstField.getAnnotation(SampleAnnot.class);
  String value = sample.value(); // Some text
} catch (NoSuchFieldException e) {
    ...
}
```

Применение метапрограммирования: Сериализация (1)

```
1  private static void serialize(PrintStream out, Object o) {
2      Class<?> clazz = o.getClass();
3      out.println(clazz.getName());
4      Field[] fields = clazz.getDeclaredFields();
5      for (Field field : fields) {
6          try {
7              field.setAccessible(true);
8              String name = field.getName();
9              Object value = field.get(o);
10             out.println("\t" + name + "=" + value);
11         } catch (IllegalAccessException e) {
12             throw new IllegalStateException(e);
13         }
14     }
15 }
```

Применение метапрограммирования: Сериализация (2)

```
1 serialize(System.out, new Wrapper(42, "Lorem Ipsum"));
```

Вывод:

```
edu.hse.example.Wrapper
    value=42
    str=Lorem Ipsum
```

Применение метапрограммирования: JUnit

```
1 public class MathTest {
2     @Test
3     public testAbs() {
4         assertEquals(10, Math.abs(10));
5         assertEquals(42, Math.abs(-42));
6         assertEquals(0, Math.abs(0));
7     }
8 }
```

Применение метапрограммирования: Spring Framework

```java
@RestController
@RequestMapping("/api/books")
public class BookController {
    @Autowired
    private BookRepository bookRepository;

    @GetMapping
    public List<Book> find(@RequestParam String title) {
        return bookRepository.findByTitle(title);
    }

    @GetMapping("/{id}")
    public Book findOne(@PathVariable Long id) {
        return bookRepository.findById(id)
          .orElseThrow(BookNotFoundException::new);
    }
}
```

```
http://127.0.0.1:8080/api/books?title=Солярис
http://127.0.0.1:8080/api/books/100500
```

Применение метапрограммирования: Lombok

```
1  @Getter
2  @Setter
3  @EqualsAndHashCode
4  @AllArgsConstructor
5  @ToString(includeFieldNames=true)
6  public class User {
7      private int id;
8      private String firstName;
9      private String lastName;
10 }
```