16.
«Сделай пометку и посмотри на себя»

Информатика, ИТИС, 1 курс
М.М.Абрамский
2017

META

Слово «Мета»

Греческое слово

μετά

«между, через, после, за, следующее»

В: Что такое метаданные?

Слово «Мета»

Греческое слово

μετά

«между, через, после, за, следующее»

В: Что такое метаданные?

О: Данные о данных.

Метаданные в программах

- Не влияют на непосредственную работу программы,
- Но могут быть выявлены другими программами на этапе компилирования или разработки, которые при этом скорректируют свою работу.

```
class MyThread extends Thread {

    public void run(boolean alive) {
        System.out.println("THREAD IS COMING! ");
    }
    public static void main(String[] args) {
        (new MyThread()).start();
    }
}
```

```
class MyThread2 extends Thread {

    public void run() {
        System.out.println("THREAD IS COMING! ");
    }
    public static void main(String[] args) {
        (new MyThread2()).start();
    }
}
```

Чем различаются эти случаи? В чем возможная ошибка?

С MyThread все было бы в порядке, если бы применили..

```
class MyThread extends Thread {

    @Override
    public void run(boolean alive) {
        System.out.println("THREAD IS COMING! ");
    }
    public static void main(String[] args) {
        (new MyThread()).start();
    }
}
```

- Компилятор бы просто не скомпилировал эту программу, т.к. метод, над которым написано @Override, не является переопределением.
- Увидев ошибку компилятора, мы бы исправили сигнатуру

Заметка про Override

- Нужда для программиста, а не для программы
- Запрещает компилирование, но при этом никак не влияет на выполнение метода (при правильном случае что она есть, что ее нет)
- Override – аннотация.
  - А аннотации – это и есть метаданные.

Про аннотации

- Не влияют напрямую на работу кода, но могут быть обнаружены другими средствами
- Могут быть аннотированы класс, метод, параметр, атрибут и т.д.
- Другие примеры аннотаций?
  - @Deprecated
  - @SuppressWarnings

Создание собственных аннотаций

Самая простая

```
@interface MyAnno {}
```

Использование:

```
@MyAnno
class MyClass {
    // ...
}
```

Методы-члены аннотации

- Объявляются как методы:

```
@interface Author {
    String name();
    int year();
}
```

- Но используются как поля:

```
@Author(name="Tony Stark", year=1996)
class MyClass {
    // ...
}
```

Значения по умолчанию

- *Внимание на year:*

```
@interface Author {
    String name();
    int year() default 2000;
}
```

- Теперь можно делать и так,

```
@Author(name="Tony Stark", year=1996)
class MyClass { …
```

- И так:

```
@Author(name="Tony Stark")
class MyClass { …
```

Аннотации, аннотирующие аннотации (лежат в java.lang.annotation)

@Retention – политика удержания аннотации (*по-деревенски: до какого этапа компилирования или выполнения аннотация видна)*

Значения лежат в перечислении RetentionPolicy:

- SOURCE – отбрасываются при компиляции
- CLASS – сохраняются в байт-коде, но недоступны во время работы
- RUNTIME – сохраняются в байт-коде и доступны во время выполнения

? Какой Retention у Override?

Аннотации, аннотирующие аннотации (лежат в java.lang.annotation)

@Target – к чему может быть применена аннотация? Значения – из перечисления **ElementType** (из того же пакета)**:**

- FIELD – поле
- METHOD – метод
- TYPE – класс, интерфейс, перечисление
- …

- Может применяться к нескольким:

```
@Target({ElementType.TYPE, ElementType.METHOD})
```

Аннотации, аннотирующие аннотации

Чтобы наш Author был доступен во время работы и применялся к объявлениям класса, интерфейса:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Author {
    String name();
    int year() default 2000;
}
```

ВСПОМНИМ ООП

Вспомним ООП.
Что есть у каждого класса

- Название класса
- Название пакета
- Атрибуты
- Методы
- ?...

Еще раз

Класс:

- Имя
- Имя пакета
- Набор атрибутов
- Набор методов
- ...

In English, please

Class:

- name
- package name
- List of attributes
- List of methods
- ...

Со шрифтом “Courier New” выглядит «по-программистски»

```
Class:
    name
    packageName
    List attributes
    List methods
    …
```

Wait, what?

```
class Class {
    String name;
    String packageName;
    List<Attribute> attributes;
    List<Method> methods;
    …
}
```

- Класс (**Class**) – тоже сущность (а сущность – это **класс**);

- А **все** конкретные **реализованные классы** *(String, User, List, Thread – ДА ВСЕ)* – **экземпляры класса Class**.

- Значит, все инструменты ООП мы можем применить к самим классам как к сущностям.
    - Это и называется рефлексией!

Класс Class

- Служебный класс, экземпляры которого хранят конкретную информацию о конкретном классе.
  - Объект класса Class для String, объект класса Class для Thread и т.п.
- Уже реализован в Java (Reflection API)

Как узнать свой класс?

- Объекту

(*пусть **obj** – экземпляр класса **MyClass***):

```
Class c = obj.getClass();
```

- Классу

(*пусть это **MyClass***):

```
Class c = MyClass.class;
```

- Названию класса

(*пусть полное имя класса: **org.kpfu.UseClass***):

```
Class c = Class.forName("org.kpfu.UseClass");
```

О-па!

- Экземпляры класса, представимого объектом класса Class, можно создавать с помощью getInstance

- `String type = scanner.next();`
- `Class c = Class.forName(type);`
- `Object o = c.newInstance();`

Параметризация

- Вообще говоря, Class параметризован
  - Не Class, а Class<T>

- Но если знать тип заранее, весь кайф от зависимости типа данных от входа пропадает.

Параметризация

```
Class<String> c =
Class.forName(интересно_какой_же_сюда_
мы_можем_вставить_класс_неужели_String
_вот_это_неожиданность);
String s = c.newInstance();
```

– бред, чего сразу String не использовал?

Параметризация

А вот так – больше возможностей:

```
String type = scanner.next();
Class c = Class.forName(type);
Object o = c.newInstance();
//тип неизвестен заранее
```

Да, экземпляры с будут Object, но мы можем в принципе вызвать instanceof – и все будет ОК.

```java
@Author(name="Smart Programmer", year=2015)
class Vector2D {
    private double x, y;

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }

    public Vector2D(){
        x = 0;
        y = 0;
    }
    public Vector2D(double x, double y){
        this.x = x;
        this.y = y;
    }

    public Vector2D add(Vector2D v) {
        return new Vector2D(x + v.getX(), y + v.getY());
    }
}
```

Это тоже классы!

- Method
- Field
- Constructor
- Annotation
- Type
- Package

если Class – сущность (класс), то почему они не могут быть классами?

? Какие атрибуты Field? Method?

Получить все методы

```
Class cv = Vector2D.class;
Method[] methods = cv.getMethods();
for (Method method : methods) {
    System.out.println(method.getName());
    System.out.println(method.getReturnType());
    System.out.println(
          Arrays.toString(
               method.getParameterTypes()
          )
    );
}
```

Получить все методы

add
class Vector2D
[class Vector2D]
setY
void
[double]
setX
void
[double]
getX
double
[]
getY
double
[]
hashCode
int
[]
getClass
class java.lang.Class
[]
equals
boolean
[class java.lang.Object]
toString
class java.lang.String
[]
wait
void
[long, int]
wait
void
[long]
wait
void
[]
notify
void
[]
notifyAll
void
[]

Получить все поля

```
Class cv = Vector2D.class;
Field[] fields = cv.getFields();
for (Field field : fields) {
    System.out.println(field.getName());
    System.out.println(field.getType());
}
```

Получить все поля

```
Class cv = Vector2D.class;
Field[] fields = cv.getFields();
for (Field field : fields) {
    System.out.println(field.getName());
    System.out.println(field.getType());
}
```

Кстати, тут ничего не выведется.

Declared

- Рефлексия учитывает инкапсуляцию, хотя может и игнорировать ее
  - getDeclaredMethod(), getDeclaredMethods(), getDeclaredFields() и др. методы с Declared в названии возвращают все соответствующие сущности, вне зависимости от модификатора,
  - Аналогичными методами без Declared будут возвращаться только public-сущности.

Получить все поля

```
Class cv = Vector2D.class;
Field[] fields = cv.getDeclaredFields();
for (Field field : fields) {
    System.out.println(field.getName());
    System.out.println(field.getType());
}
```

Вывод:

x
double
y
double

Да, кстати, проверка Аннотаций

```
Class cv = Vector2D.class;
Annotation[] annotations = cv.getAnnotations();
for(Annotation annotation : annotations){
    if(annotation instanceof Author){
        ...
    }
}
```

Проверяем, что Vector2D аннотирован @Author

Самый экшн

у Класса:

- getMethod(…) – возврат метода по **сигнатуре**;
  - ✓ String.class.getMethod(“charAt”, int.class);
  - ✓ String.class.getMethod(“subString”, new Class[]{int.class, int.class});
- getConstructor(…) – возврат конструктора по **сигнатуре**
  - – По аналогии

у Метода:

- invoke() – вызов метода

Сигнатура в терминах рефлексии

- “Имя и набор типов параметров”
- String и массив объектов класса Class

```
Class cs = String.class;
Method m = cs.getMethod(
      "indexOf",
      new Class[]{String.class, int.class}
);
// или
Method m = cs.getMethod(
      "indexOf", String.class, int.class);
```

varargs

```
// varargs - параметры все передадутся в массив objects
public static void printAll(Object... objects) {
    for (Object o: objects) {
        System.out.println(o);
    }
}
public static void main(String[] args) {
    // можно вызывать на произвольном количестве аргументов
    printAll("hi", 100500);
    printAll("hi", "itis", 2017, new int[]{1,2,3,4});
}
```

Reflection in action!

```
Scanner scanner = new Scanner(System.in);

Class cv = Class.forName(scanner.next());
Class cv2 = Class.forName(scanner.next());

String methodName = scanner.next();
Method m = cv.getMethod(methodName, cv2);

Object o1 = cv.newInstance();
Object o2 = cv2.newInstance();

// вызываю у o1 метод m (с именем methodName)
// на объекте o2
System.out.println(m.invoke(o1, o2));
```

```
Scanner scanner = new Scanner(System.in);
Class cv = Class.forName(scanner.next());
Class cv2 = Class.forName(scanner.next());
String methodName = scanner.next();
Method m = cv.getMethod(methodName, cv2);
Object o1 = cv.newInstance();
Object o2 = cv.newInstance();
System.out.println(m.invoke(o1, o2));
```

- Работает, если я подам на вход:
  - **Vector2D Vector2D add**

    т.к. в Vector2D есть add(Vector2D)
  - **java.util.HashSet int add**

    т.к. в HashSet есть add(Object)
  - **java.lang.Thread java.lang.String setName**

    т.к. в java.lang.Thread есть setName(String)

IMPORTANT!

Я могу управлять работой программ гибко, на разных классах, не переписывая их и не компилируя каждый раз заново!”

Это легло в основу многих java-фреймворков

Рефлексия в других языках

- В Java обычный класс и объект класса Class, соответствующий обычному классу – разные сущности
- В Python, например, это одно и то же:

```
class Pet:
    pass
```

Объявил одновременно и класс Pet, и экземпляр класса Class, соответствующий Pet.

Могу внутри него писать методы для Pet как обычного класса,

Могу для Pet как для объекта класса Class (class methods)

Прочитать

- http://www.quizful.net/post/java-reflection-api (rus)
- http://tutorials.jenkov.com/java-reflection/methods.html (eng)