

АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 4

ПЛАН

- Статика
- Расширения
- Инфиксные вызовы

КРАТКОЕ СОДЕРЖАНИЕ

- Статика живет вне классов
- Один вариант - внеклассовые свойства и функции
- Типичное применение: вспомогательные функции уровня пакета и выше
- С приватностью на уровне файла
- Разберем другие варианты: синглтон и компаньон

СИНГЛТОН КАК ШАБЛОН

- Один из классических шаблонов
- Класс, у которого по смыслу бывает один объект
- Пример: `java.lang.Runtime`

JAVA: BASELINE

```
1 abstract class Context {  
2     protected String name;  
3  
4     protected Context(String name) {  
5         this.name = name;  
6     }  
7  
8     public String getName() {  
9         return name;  
10    }  
11 }
```

JAVA: BASELINE

```
1 class RequestContext extends Context {  
2     public RequestContext(String request) {  
3         super("request:" + request);  
4     }  
5 }
```

JAVA: BASELINE

```
1 class GlobalContext extends Context {
2     private static GlobalContext INSTANCE =
3         new GlobalContext();
4
5     private GlobalContext() {
6         super("name");
7     }
8
9     public static GlobalContext getInstance() {
10         return INSTANCE;
11     }
12 }
```

JAVA: BASELINE

```
1 class C {  
2     public static void m() {  
3         System.out.println(  
4             GlobalContext.getInstance().getName()  
5         );  
6     }  
7 }
```


РАЗБЕРЕМ

- Бывают разные контексты
- Например, контекст запроса
- У каждого запроса - свой
- И глобальный контекст

РАЗБЕРЕМ

- Глобальный контекст один
- Но мы не хотим его держать как класс
- Потому что класс нельзя унаследовать
- Или десериализовать

ЛЕНИВЫЕ СИНГЛТОНЫ

- Часто синглтонам соответствует дорогостоящий объект
- Например, представляющий удаленное соединение
- С тяжелой инициализацией
- Которую не хочется делать без надобности

JAVA: ЛЕНИВЫЙ СИНГЛТОН

```
1  class LazyGlobalContext extends Context {
2      private static GlobalContext INSTANCE = null;
3
4      private GlobalContext() {
5          super("name");
6      }
7
8      public static GlobalContext getInstance() {
9          if (INSTANCE == null) {
10             INSTANCE = new LazyGlobalContext();
11         }
12         return INSTANCE;
13     }
14 }
```

ЕЩЕ НЕ ВСЕ

- Это непотокобезопасный синглтон
- Его можно сделать потокобезопасным разными способами
- Подороже и подешевле
- Покомпактнее в коде и не очень

JAVA: THREAD-SAFE

```
1 public class LazyGlobalContext {
2     private static volatile LazyGlobalContext INSTANCE;
3
4     public static LazyGlobalContext getInstance() {
5         if (INSTANCE == null) {
6             synchronized (LazyGlobalContext.class) {
7                 if (INSTANCE == null) {
8                     INSTANCE = new LazyGlobalContext();
9                 }
10            }
11        }
12        return INSTANCE;
13    }
14 }
```

JAVA: КРАСИВЫЙ ВАРИАНТ

```
1 public class LazyGlobalContext {
2     private static class Holder {
3         public static final LazyGlobalContext
4             INSTANCE = new LazyGlobalContext();
5     }
6
7     public static LazyGlobalContext getInstance() {
8         return SingletonHolder.instance;
9     }
10 }
```

СУММИРУЕМ

- Полезная конструкция
- Как языковая конструкция отсутствует
- Реализуется через boilerplate-код
- С нюансами про ленивость и потокобезопасность

ЧТО ДАЕТ KOTLIN

- Конструкция object
- Ставится там же, где класс
- Круглых скобок нет
- Одно пространство имен с классами

OBJECT

- В фигурных скобках - все как в классе
- `val`, `fun`, `init`, и даже `this`
- В глубине души это даже и не статика
- Но может использоваться как `namespace` для статических конструкций

ПОД КАПОТОМ

- Под капотом есть поле INSTANCE
- Инициализируется в статическом инициализаторе
- Решается проблема базового boilerplate-кода
- И потокобезопасности
- С ленивостью посложнее

ПРИМЕР

```
1 object Singleton {  
2     val VALUE = 12345  
3     init {  
4         println("I'm singleton")  
5     }  
6 }  
7  
8 fun main() {  
9     println("hello")  
10 }  
11 // hello
```

ПРИМЕР

```
1 object Singleton {
2     val VALUE = 12345
3     init {
4         println("I'm singleton")
5     }
6 }
7 object Other {
8     val ownValue = 23456;
9     val value = Singleton.VALUE
10    init {
11        println("I'm other")
12    }
13 }
14 // .....
```

ПРИМЕР

```
1 // .....
2
3 fun main() {
4     println("hello")
5     println(Other.ownValue)
6 }
7 // hello
8 // I'm singleton
9 // I'm other
10 // 23456
```

РАЗБЕРЕМ ВТОРОЙ

- Начали работу
- Класс `Other` нам известен, но не инициализирован
- Только когда `Other.ownValue` стал нужен, начинаем инициализировать `Other`
- И нам становится нужен `Singleton.VALUE`

РАЗБЕРЕМ ВТОРОЙ

- Тут инициализируем Singleton
- Печатаем "I'm singleton"
- Доходим до печати "I'm other"
- Инициализировали Singleton - хотя в main им не воспользовались

ЕЩЕ ТОНКОСТЬ

- Первый кажется простым: jvm ничего не знает про Singleton
- Но есть промежуточный вариант: обращение к Other под if
- Если if не срабатывает ?
- Гарантирована ли неактивность инициализаторов ?

ЕЩЕ ТОНКОСТЬ

- В моей инсталляции - не срабатывает
- Но не факт, что так будет всегда
- JVM-спецификация допускает и ленивую, и жадную линковку
- Де-факто популярна ленивая - но без гарантии
- Есть механизм гарантированной ленивости

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ

- Собираем данные из конфиг файлов в синглтон Config
- Создаем сложный компаратор (объект без состояния)
- Формируем namespace для констант/свойств/функций

КОМПАНЬОНЫ

- Объединение методов и свойств
- Не привязанных к одному объекту
- Но имеющих особые отношения с объектами данного класса
- Например, когда есть "асимметричная" операция над несколькими объектами одного типа

ПРИМЕР

```
1 class Point(private val x: Int, private val y: Int,  
2             val payload: Int) {  
3     companion object {  
4         fun nearestPair(p1: Point,  
5                         p2: Point,  
6                         p3: Point): Pair<Point, Point> {  
7             //...  
8             return Pair(p1, p2)  
9         }  
10    }  
11 }  
12  
13 // .....
```

ПРИМЕР

```
1 // .....
2
3 fun main() {
4     Point.nearestPair(
5         Point(1, 2, 3),
6         Point(2, 3, 4),
7         Point(3, 4, 5)
8     )
9 }
```

МОЖНО И ПО-ДРУГОМУ

- Обычный объект
- Поместить внутрь класса
- Только надо явно указать его имя

ПРИМЕР

```
1 class Point(private val x: Int, private val y: Int,  
2             val payload: Int) {  
3     object utils {  
4         fun nearestPair(p1: Point,  
5                         p2: Point,  
6                         p3: Point): Pair<Point, Point> {  
7             //...  
8             return Pair(p1, p2)  
9         }  
10    }  
11 }  
12  
13 // .....
```


ПРИМЕР

```
1 // .....
2
3 fun main() {
4     Point.utils.nearestPair(
5         Point(1, 2, 3),
6         Point(2, 3, 4),
7         Point(3, 4, 5)
8     )
9 }
```

ЕЩЕ САХАРОК

- Элементы компаньона "напрямую" доступны из объектов
- Ну примерно как статические методы в Java
- Вариант применения: вспомогательная логика реализации класса
- Не привязанная к объекту

ПРИМЕР

```
1 class Rational(private val a: Int, private val b: Int) {
2     companion object {
3         fun gcd(p: Int, q: Int): Int =
4             if (q == 0) p else gcd(q, p % q)
5     }
6     val nom: Int
7     val denom: Int
8     init {
9         val gcd = gcd(a, b)
10        nom = a / gcd
11        denom = b / gcd
12    }
13 }
```

МОТИВИРУЮЩИЙ ПРИМЕР

- Пишем GUI-приложение
- Дошло до использования библиотеки AWT
- Базовое GUI для JVM-платформы
- API разработано в прошлом веке

ПРИМЕР

```
1 val point = Point(100, 200)
2 println(point.x)
3 println(point.y)
4 point.move(10, 20)
5 println(point)
```

РАЗБЕРЕМ

- Что не устраивает
 - move - изменяющий метод
 - А хочется создавать новый объект
 - И в ООП стиле

ПРИМЕР: НЕ ООП

```
1 fun moved(p: Point, dx: Int, dy: Int) =  
2     Point(p.x + dx, p.y + dy)  
3  
4 fun scaled(p: Point, c: Int) =  
5     Point(p.x * c, p.y * c)  
6  
7     val newPoint =  
8         moved(  
9             scaled(  
10                moved(point, 10, 20),  
11                4),  
12                3, 4  
13            )
```

АЛЬТЕРНАТИВА

- По-честному "встроиться" в чужой класс нельзя
- Но можно поддерживать синтаксически
- Можно к имени функции приписать имя класса - слева через точку
- И потом вызывать этот дополнительный "метод"

ПРИМЕР: ООП

```
1 fun Point.moved(dx: Int, dy: Int) =  
2     Point(this.x + dx, this.y + dy)  
3  
4 fun Point.scaled(c: Int) =  
5     Point(this.x * c, this.y * c)  
6  
7 fun main() {  
8     val point = Point()  
9  
10    point.moved(10, 20)  
11        .scaled(3)  
12        .moved(3, 4)  
13 }
```

КАК ЭТО РАБОТАЕТ

- Создается статический метод
- В месте определения расширения
- Тип первого параметра - расширяемый класс
- Остальные параметры - из расширения

ВАЖНОЕ СЛЕДСТВИЕ

- Нет полноценного полиморфизма
- Мы можем класс и подкласс расширить одним методом
- С тем же именем и теми же параметрами
- Но это не будет "перекрытием"

ВАЖНОЕ СЛЕДСТВИЕ

- Внутри расширения нет слова `super`
- И выбор между методами делается статически
- И это опасная ошибка

ПРИМЕР

```
1 open class C1
2 class C2() : C1()
3
4 fun C1.m() = println("C1.m")
5 fun C2.m() {
6     println("C2.m")
7 }
8 fun f(c1: C1) = c1.m()
9
10 fun main() {
11     f(C1())
12     f(C2())
13 }
```

ПРИМИТИВНЫЕ ТИПЫ

- Примитивные типы ничем не хуже
- ```
fun Int.twice() = this * 2
```
- Здесь `this` - число

# СТИЛИ ИСПОЛЬЗОВАНИЯ

- Top-level рядом с местом использования
- Или даже локально
- Можно сгруппировать в импортированный пакет
- Отдельный жанр - адаптация Java-библиотек под Kotlin-стиль

# "СУЖЕНИЕ"

- Местами сделано на уровне реализации
- С ограниченным эффектом
- Можно вернуть через расширение или обойти через Java
- Самому "заккрыть" метод нельзя



# ВЫЗОВ ИЗ JAVA

- Разобраться, в каком классе
- Вызвать как обычный метод
- `Int` превратится в `int`

# ПРИМЕР

```
1 fun main(args: Array<String>) {
2 val map = mapOf(1 to "one", 2 to "two")
3 println(map[1])
4 }
```

# РАЗБЕРЕМ

- Что стоит за `to` между числом и строкой ?
  - `mapOf` принимает элементы типа `Pair`
  - `to` смотрится как литерал, описывающий `Pair`
  - можем даже напечатать

# ПРИМЕР

```
1 fun main(args: Array<String>) {
2 println(1 to "one")
3 println(1.to("one"))
4 }
```

# КАК РАБОТАЕТ

- Ключевое слово `infix`
  - Перед определением функции или расширения
  - Удобно сочетать с локальными расширениями
  - Например, заменить `to` на более подходящее

# ПРИМЕР

```
1 infix fun Any.with(v: Any) = Pair(this, v)
2
3 fun main() {
4 print(1 with 2)
5 }
```

