АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 3

ПЛАН ЛЕКЦИИ

- Статика: продолжение
- Нюансы расширений

НА ЧЕМ ОСТАНОВИЛИСЬ

- Статика как что-то непривязанное ни к какому классу: внеклассовые функции/свойства
- Статика как группа утилит к внешнему классу: расширения
- Но есть другие: синглтоны и компаньоны
- Синглтон - скорее псевдостатика

СИНГЛТОН КАК ШАБЛОН

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

СИНГЛТОН КАК ШАБЛОН

- Разберемся с шаблоном в общем виде
- Почему не подходит класс со статическими полями/методами
- Что предлагает Kotlin

ЧЕМ ПЛОХА JAVA-СТАТИКА ДЛЯ СИНГЛТОНА

- Хотим передавать синглтон параметром и возвращать как значение
- Хотим встраивать его в иерархию наследования
- Иногда - передавать параметры
- Управлять моментом инициализации

JAVA: BASELINE

```java
abstract class Context {
    protected String name;

    protected Context(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

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

МОМЕНТ ИНИЦИАЛИЗАЦИИ

```java
// Что напечатается ?
class C2 {
    class C {
        static {
            System.out.println("C class init");
        }
    }
}
public class Main {
    public static void main(String[] args) {
    }
}
```

МОМЕНТ ИНИЦИАЛИЗАЦИИ

```java
// А так ?
class C {
    static {
        System.out.println("C class init");
    }
}
class C2 {
}
public class Main {
    public static void m(String[] args) {
        C c = new  C();
        System.out.println("c: " + c);
    }
    public static void main(String[] args) {
    }
```

МОМЕНТ ИНИЦИАЛИЗАЦИИ

```java
// А так ?
class C {
    static {
        System.out.println("C class init");
    }
}
public class Main {
    static C c = new C();

    public static void main(String[] args) {
    }
}
```

ЕЩЕ ИНТЕРЕСНОЕ

```
1  class C {
2      static {
3          System.out.println("C class init");
4      }
5  }
6  public class Main {
7      public static void main(String[] args) {
8          System.out.println("main: " + C.class);
9          C c = new C();
10     }
11 }
```

ЕЩЕ ИНТЕРЕСНОЕ

```
1  class C1 {
2      static {
3          System.out.println("C1 class init: " + C2.c1);
4      }
5      static C2 c2 = new C2();
6      static int v = 5;
7  }
8
9  class C2 {
10     static {
11         System.out.println("C2 class init: " + C1.c2);
12     }
13     static C1 c1 = new C1();
14 }
15
```

ЖАДНЫЕ/ЛЕНИВЫЕ

- "статический класс" = "состояние" + "поведение"
- Иногда нужна жадность, иногда ленивость
- Жадность - для активации какой-то деятельности

ЛЕНИВЫЕ СИНГЛТОНЫ

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

JAVA: ЛЕНИВЫЙ СИНГЛТОН

```java
class LazyGlobalContext extends Context {
    private static GlobalContext INSTANCE = null;

    private GlobalContext() {
        super("name");
    }

    public static GlobalContext getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new LazyGlobalContext();
        }
        return INSTANCE;
    }
}
```

ЕЩЕ НЕ ВСЕ

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

JAVA: THREAD-SAFE

```java
public class LazyGlobalContext {
    private static volatile LazyGlobalContext INSTANCE;

    public static LazyGlobalContext getInstance() {
        if (INSTANCE == null) {
            synchronized (LazyGlobalContext.class) {
                if (INSTANCE == null) {
                    INSTANCE = new LazyGlobalContext();
                }
            }
        }
        return INSTANCE;
    }
}
```

JAVA: КРАСИВЫЙ ВАРИАНТ

```java
// "Holder" pattern
public class LazyGlobalContext {
    private LazyGlobalContext() {
       // инициализация
       System.out.println("AAAA")
    }

    private static class Holder {
        public static final LazyGlobalContext
            INSTANCE = new LazyGlobalContext();
    }

    public static LazyGlobalContext getInstance() {
        return Holder.INSTANCE;
    }
```

СУММИРУЕМ

- Полезная конструкция
- Как языковая конструкция остутствует
- 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/жадностью и параметрами посложнее

НАСЛЕДИЕ JVM

```
object Singleton {
    val VALUE = 12345
    init {
        println("I'm singleton")
    }
}

fun main() {
    println("hello")
}
// hello
```

ПРИМЕР

```
object Singleton {
    val VALUE = 12345
    init {
        println("I'm singleton")
    }
}
object Other {
    val ownValue = 23456;
    val value = Singleton.VALUE
    init {
        println("I'm other")
    }
}
// ..........
```

ПРИМЕР

```
// ..........

fun main() {
    println("hello")
    println(Other.ownValue)
}
// hello
// I'm singleton
// I'm other
// 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 сработает ?
- 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 "ассиметричная" операция над несколькими объктами 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

```
// ..................

fun main() {
    Point.utils.nearestPair(
        Point(1, 2, 3),
        Point(2, 3, 4),
        Point(3, 4, 5)
    )
}
```

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 }
```

РАСШИРЕНИЯ ПРИМИТИВНЫХ ТИПОВ

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

СТИЛИ ИСПОЛЬЗОВАНИЯ РАСШИРЕНИЙ

- 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 "закрыть" 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 }
```

РАЗБЕРЕМ

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