Введение в дженерики – обобщенные классы и функции

В программировании иногда приходится обрабатывать разные типы данных похожим образом. С другой стороны, в языках со статической типизацией, к которым относится Kotlin, параметры функции типизированы. Это значит, что если функция принимает целое число, ей нельзя передать вещественное.

Частично проблему решает наследование, ведь мы можем присваивать переменным родительских типов объекты дочерних:

```
fun main() {
    val a: Int = 10
    val b: Double = 1.5
    fraction(a) // 2.0
    fraction(b) // 0.3
}
```

```
fun fraction(n: Number) {
    println(n.toDouble() / 5)
}
```

В данном случае `Number` – это класс Kotlin, который является родительским для числовых типов данных. Однако подобное не всегда подходит. Например, мы хотели бы возвращать из функции данные заранее точно неизвестного типа.

Некоторые языки программирования позволяют создавать так называемые дженерики (generics) – обобщенные функции и классы. Рассмотрим пример определения и вызова обобщенной функции на языке Kotlin.

```
fun main() {
    val a: Int = 10
    val b: String = "Hello"
    val c: List<Int> = listOf(1, 5)
```

```
    val aa: List<Int> = doTwo(a)
    val bb: List<String> = doTwo(b)
    val cc: List<List<Int>> = doTwo(c)

    println(aa) // [10, 10]
    println(bb) // [Hello, Hello]
    println(cc) // [[1, 5], [1, 5]]
}
```

```
fun <T> doTwo(obj: T): List<T> {
    val list: List<T> = listOf(obj, obj)
    return list
}
```

Функция `doTwo()` не только способна принимать разный тип данных, но и возвращает разное.

T – это неизвестный на момент определения функции тип параметра. О том, что функция параметризована таким образом, что она обобщенная, относится к дженерикам, сообщается угловыми скобками перед ее именем. Буква *T* – это просто соглашение, обозначить неизвестный тип можно любым идентификатором.

Другими словами, записывая перед именем функции `<T>`, мы говорим, что везде где в функции будет встречаться идентификатор *T*, его нужно будет заменить на тип, который будет известен в момент вызова функции. Когда функция `doTwo()` вызывается с аргументом-целым числом, то *T* становится **`Int`**, когда со списком – *T* становится **`List<Int>`**. Когда мы вызываем функцию, передавая ей строку, то тип параметра *obj* – это **`String`**, а возвращаемого из функции значения – **`List<String>`**.

Не обязательно, чтобы все параметры функции-дженерика были параметризованы. Так ниже, у функции *parePrint* неизвестный тип имеет только один параметр, у второго тип определен – **`Char`**.

```
fun main() {
    val a: Int = 10
    val b: String = "Hello"
```

```
    val c: List<Int> = listOf(10, 16, 3)
    parePrint(a, '{') // {10}
    parePrint(b, '[') // [Hello]
    parePrint(c, '"') // "[10, 16, 3]"
}
```

```
fun <T> parePrint(obj: T, p: Char) {
    when(p) {
        '(', ')' -> println("($obj)")
        '[', ']' -> println("[$obj]")
        '{', '}' -> println("{$obj}")
        else -> println("$p$obj$p")
    }
}
```

У обобщенного параметра может быть ограничение, которое записывается после двоеточия в его объявлении. Например, чтобы ограничить обобщенный параметр только числовыми типами, его следует объявить как `<T: Number>`:

```
fun main() {
    val a: Int = 10
    val b: Double = 1.5
    println(fraction(a, 5)) // 2.0
    println(fraction(b, 3)) // 0.5
}
```

```
fun <T: Number> fraction(n: T, f: Int): Double {
    return n.toDouble() / f
}
```

В отличие от приведенного в начале урока примера обычной функции, в которой параметр *n* имеет тип `Number`, здесь *n* в момент вызова функции принимает более конкретный тип. Например, `Int`.

Обобщенными могут быть не только функции, но и классы. Хотя обобщены не они сами, а описанные в них свойства и методы. В случае класса обобщенный параметр указывается после имени класса.

```
class Something<T> {
    val prop: T

    constructor(p: T) {
        prop = p
    }
}
```

```
fun main() {
    val a: Something<Int> = Something(10)
    val b: Something<String> = Something("Hello")
    println(a.prop) // 10
    println(b.prop) // Hello
}
```

Такой класс называют параметризованным, так как на его основе создаются объекты по сути разных типов. В примере мы не можем объект типа `Something<Int>` присвоить переменной, объявленной как `Something<String>`.

Класс выше описан через вторичный конструктор для наглядности. Обычно используется первичный конструктор. Класс будет выглядеть так:

```
class Something<T>(p: T) {
    val prop: T = p
}
```

Или даже так:

```
class Something<T>(val prop: T)
```

С подобным мы уже сталкивались, используя стандартную библиотеку Kotlin. Так массивы, списки и словари – это параметризованные классы.

```
fun main() {
    val a: List<Int> = listOf(4, 5)
```

```
    val b: Map<Char, Int> =
        mapOf('a' to 2, 'b' to 10)
}
```

В случае со словарями у класса не один, а два обобщенных параметра. В объявлении класса это выглядит примерно так:

```
class Something<T, V>(p: T, q: V) {
    val prop: T = p
    val qty: V = q
}
```

Какими типами окажутся поля *prop* и *qty* определится только при создании объекта.

```
fun main() {
    val a: Something<String, Int>
    a = Something("Hello", 5)
}
```

PDF-версия курса с дополнительными уроками