v1.1.2

Funciones

Kotlin

Funciones

Definición de una función

```
fun double(x: Int): Int {
}
```

Llamada a una función

```
val result = double(2)

Sample().foo()
```

Parámetros y argumentos

Parámetros y argumentos

- Parámetros: declaraciones de variables para utilizar desde dentro de la función
- Argumentos: valores usados al hacer la llamada a la función

Parámetros

```
fun powerOf(number: Int, exponent: Int) {
    ...
}
```

Parámetros por defecto

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size()) {
    ...
}
```

Etiquetas de argumentos explícitas

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {
    ...
}
```

Etiquetas de argumentos explícitas

```
reformat(str)

reformat(str, true, true, false, '_')   // no da pistas

reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)

reformat(str, wordSeparator = '_')
```

Tipo de retorno Unit

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}

fun printHello(name: String?) {
    ...
}
```

Funciones de una expresión

```
fun double(x: Int): Int = x * 2
```

```
fun double(x: Int) = x * 2
```

Parámetros indeterminados

- Son parámetros que permiten introducir múltiples valores
- Se declaran con el modificador `vararg`
- Los valores tienen que ser del mismo tipo
- Los valores llegan a la función como un array del tipo apropiado
- Sólo puede haber uno

Parámetros indeterminados

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}


val list = asList(1, 2, 3)
```

Ámbito de las funciones

Ámbito de las funciones

- El Kotlin, las funciones se pueden crear en el ámbito global, no es necesario crear una clase para contenerlas como en Java
- También pueden ser locales, miembro o extensiones

Funciones locales

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

Funciones miembro

```
class Sample() {
    fun foo() { print("Foo") }
}


Sample().foo()
```

Funciones genéricas

```
fun <T> singletonList(item: T): List<T> {
    // ...
}
```

Funciones de primer orden y λ

Funciones de primer orden

- Una función de primer orden es aquella que admite funciones como parámetros o devuelve una función

Funciones de primer orden

```
fun <T> lock(lock: Lock, body: () -> T): T
{
    lock.lock()
    try {
        return body()
    }
    finally {
        lock.unlock()
    }
}
```

- El parámetro `body` es de tipo función `() -> T`
- Tiene que recibir como parámetro una función, cualquiera, que tenga esa firma

Paso de funciones como argumento

```
fun toBeSynchronized() = sharedResource.operation()

val result = lock(lock, ::toBeSynchronized) // referencia a función


val result = lock(lock, { sharedResource.operation() }) // lambda
```

Ejemplo

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = arrayListOf<R>()
    for (item in this)
        result.add(transform(item))
    return result
}
```

```
val doubled = ints.map { value -> value * 2 }
```

Expresiones lambda

```
max(strings, { a, b -> a.length < b.length })
```

- Son funciones anónimas
- Podemos imaginarlas como funciones que no se declaran, sino que se pasan directamente como expresión

Sintaxis de las expresiones lambda

```
val sum = { x: Int, y: Int -> x + y }
```

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

Uso de it

```
ints.filter { it > 0 }
    // (it: Int) -> Boolean
```

- Si solo hay un parámetro y Kotlin puede deducir el tipo, podemos usar `it` para referirnos a él

Clausuras

```
var sum = 0

ints.filter { it > 0 }.forEach {
    sum += it
}

print(sum)
```

- La expresión lambda puede acceder a su *closure*, que son las variables declaradas en su ámbito superior
- Puede modificar estas variables