KOTLIN

Collection Of Frequently Used Idioms In Kotlin

By DJ Last updated Feb 13, 2018

3 min read

A collection of random and frequently used idioms in Kotlin. If you have a favorite idiom, contribute it by sending a pull request.

Creating DTOs (POJOs/POCOs)

```
data class Customer(val name: String, val email: String)
```

provides a `Customer` class with the following functionality:

- getters (and setters in case of *var*s) for all properties
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- `component1()` , `component2()` , ..., for all properties (see Data classes)

Default values for function parameters

```
fun foo(a: Int = 0, b: String = "") { ... }
```

Filtering a list

```
val positives = list.filter { x -> x > 0 }
```

Or alternatively, even shorter:

```
val positives = list.filter { it > 0 }
```

String Interpolation

```
println("Name $name")
```

Instance Checks

```
when (x) {
    is Foo -> ...
    is Bar -> ...
    else   -> ...
}
```

Traversing a map/list of pairs

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

`k`, `v` can be called anything.

Using ranges

```
for (i in 1..100) { ... }  // closed range: includes 100
for (i in 1 until 100) { ... } // half-open range: does not include 100
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

Read-only list

```
val list = listOf("a", "b", "c")
```

Read-only map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

Accessing a map

```
println(map["key"])
map["key"] = value
```

Lazy property

```
val p: String by lazy {
    // compute the string
}
```

Extension Functions

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

Creating a singleton

```
object Resource {
    val name = "Name"
}
```

If not null shorthand

```
val files = File("Test").listFiles()

println(files?.size)
```

If not null and else shorthand

```
val files = File("Test").listFiles()

println(files?.size ?: "empty")
```

Executing a statement if null

```
val data = ...
val email = data["email"] ?: throw IllegalStateException("Email is missing!")
```

Execute if not null

```
val data = ...

data?.let {
    ... // execute this block if not null
}
```

Map nullable value if not null

```
val data = ...

val mapped = data?.let { transformData(it) } ?: defaultValueIfDataIsNull
```

Return on when statement

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

‘try/catch’ expression

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // Working with result
}
```

‘if’ expression

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

Builder-style usage of methods that return `Unit`

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

Single-expression functions

```
fun theAnswer() = 42
```

This is equivalent to

```
fun theAnswer(): Int {
    return 42
}
```

This can be effectively combined with other idioms, leading to shorter code. E.g. with the *when*-expression:

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

Calling multiple methods on an object instance (‘with’)

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { //draw a 100 pix square
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

Java 7’s try with resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

Convenient form for a generic function that requires the generic type information

```
//  public final class Gson {
//     ...
//     public <T> T fromJson(JsonElement json, Class<T> classOfT) throws
JsonSyntaxException {
//     ...

inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T = this.fromJson(json,
T::class.java)
```

Consuming a nullable Boolean

```
val b: Boolean? = ...
if (b == true) {
    ...
} else {
    // `b` is false or null
}
```

Reference: Idioms

idioms kotlin