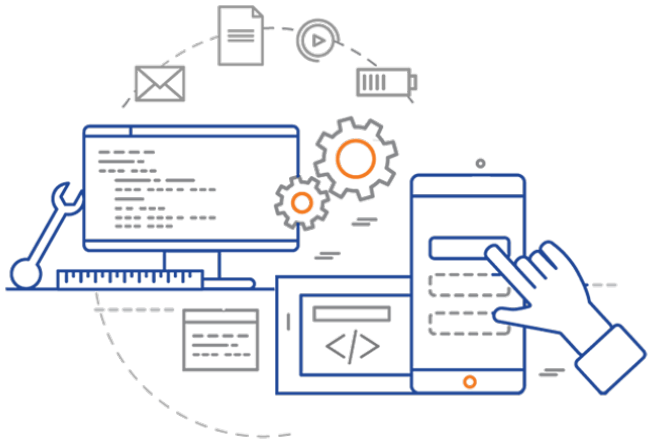


CS 398: Application Development

Week 03 Video: Kotlin 2

Control-Flow

Control Flow

https://kotlinlang.org/docs/reference/control-flow.html

“Traditional”

- `if... then.. else`
- `while, do... while`
- `break, continue`

New!

- `when // replaces switch`
- `for (s : collection) // iteration`
- `for (a in 1.. 5) // iteration through range`

If... Then

`if... then` has both a statement form (no return value) and an expression form (return value).

```
// statement
if (a > b) {
  println(a)
} else {
  println(b)
}

// expression
val max = if (a > b) a else b
```

This is why Kotlin doesn't have a ternary operator: 'if' used as an expression serves the same purpose.

For Loops

A ‘for‘ loop iterates through anything that provides an iterator. This is equivalent to the ‘foreach‘ loop in languages like C#.

```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
      println(item)
}
val items = listOf("apple", "banana", “kiwifruit")
for (index in items.indices) {
      println("item at $index is ${items[index]}")
}
```

Ranges

The range operator '..' generates a sequence of values. You often use these with 'for...' loops

```
for (i in 1..3) {
  println(i)
}

for (i in 6 downTo 0 step 2) {
  println(i)
}
```

Check if a number is within range:

```
val x = 10
val y = 9

if (x in 1..y+1) {
  println("fits in range")
}
```

When

‘when‘ replaces the switch operator of C-like languages:

```
when (x) {
  1 -> print("x == 1")
  2 -> print("x == 2")
  else -> print("x is neither 1 nor 2")
}

when (x) {
  0, 1 -> print("x == 0 or x == 1")
  in 2..10 -> print("x is in the range")
  in validNumbers -> print("x is valid")
  !in 10..20 -> print("x is outside the range")
  else -> print("none of the above")
}
```

```
// ‘when’ as an expression returns a value
fun describe(obj: Any): String =
  when (obj) {
     1 -> “One”
     “Hello” -> "Greeting"
     isLong -> “Long"
     !is String -> "Not a string"
     else -> “Unknown”
    }
```

Collections

Collections

A collection is a group of some variable number of items (possibly zero) of the same type. Objects in a collection are called `elements`.

Kotlin uses Java collection classes, but provides mutable and immutable interfaces.

Pair	A tuple[1] of two values.
Triple	A tuple of three values.
List	An ordered collection of objects.
Set	An unordered collection of objects.
Map	An associative dictionary of keys and values.
Array	Indexed, fixed-size collection of object or primaries.

Built-in Collection classes

1. A tuple is a data structure representing a sequence of **n** elements.

Array

Arrays are indexed, fixed-sized collection of objects and primitives. We prefer other collections, but these are offered for legacy and compatibility with Java.

```
// Create using the `arrayOf()` library function
arrayOf(1, 2, 3)

// Create using the Array class constructor
// Array<String> ["0", "1", "4", "9", "16"]
val asc = Array(5) {
  i -> (i*i).toString()
}
asc.forEach { println(it) }
```

You can access array elements through using the ‘[]‘ operators, or the ‘get()‘ and ‘set()‘ methods.

Pair

A pair is a tuple of two values. Use ‘var‘ or ‘val‘ to indicate mutability. The ‘to‘ keyword indicates a Pair.

```
// immutable
val newfoundland = Pair("Gander Airport", "YQX")
val nova_scotia = "Halifax Airport" to "YHZ" // ‘to’ implies a Pair
println(nova_scotia)

// mutable
var ontario = Pair("Toronto Pearson", "YYZ")
ontario = Pair("Billy Bishop", "YTZ")
```

```
// mixed types
val canadian_exchange = Pair("CDN", 1.38)

// access elements
val characters = Pair("Tom", "Jerry")
println(characters.first)
println(characters.second)

// de-structuring
val (first, second) = Pair("Calvin", "Hobbes") // split a Pair
println(first)

println(second)
```

List

A list is an ordered collection of objects.

```
// immutable (due to listOf)
var fruits = listOf( "advocado","banana","cantaloupe")
println(fruits.get(0))
println(fruits[0])
// fruits.add("dragon fruit") // unresolved

// mutable (due to mutableListOf)
var mutableFruits = mutableListOf( "advocado","banana")
mutableFruits.add("cantaloupe") // this works!
```

Map

A map is an associative dictionary of key and value pairs (i.e. it maps one value to another).

```
// immutable (initialize with pairs)
val imap = mapOf(1 to "x", 2 to "y", 3 to "z")
println(imap)
// imap.put(4, "q") // immutable, so unresolved reference

// mutable
val mmap = mutableMapOf(5 to "x", 6 to "y")
mmap.put(7,"z") // ok
println(mmap)
```

```
// lookup a value
println(mmap.get(5)) // x

// iterate over map
for ((k, v) in imap) {
  println("$k = $v")
}

// alternate syntax
mmap.forEach { k, v -> println("$k = $v") }
mmap.forEach {
  println("${it.key} = ${it.value}”)
}
```

Accessing Elements

Kotlin has special properties that can be used to access data elements in collections.

```
val list = listOf("one", “two", "three", "four")
list.contains("four")) // true

// slice - extract into a new collection
list.slice(1..3) // [two, three, four]
list.slice(0..4 step 2) // [one, three]

// take - extract n elements
list.take(3) // [one, two, three]
list.takeLast(2) // [three, four]

// extract using iterators
list.first { it.length > 3 } // [three]
list.last { it.startsWith("o") } // [one]
```

Processing Elements

We can use functional-style processing with it as an iterator.

```
val list = listOf(1,2,3,4) // [1,2,3,4]

// filter returns elements matched by predicate
list.filter { it % 2 == 0 } // [2, 4]

// map returns elements after transformation
list.map { it * 2 } // [2, 4, 6, 8]

// flatMap returns elements from transform
list.flatMap { listOf(it, it + 10) } // [1, 11, ...]
```

```
// fold/reduce accumulates elements
list.fold(0.0) { acc, i -> acc + i } // 10.0
list.reduce { acc, i -> acc * i } // 24

// forEach/onEach perform action on each element
list.forEach { print(it) } // returns Unit
list.onEach { print(it) } // returns [1, 2, 3, 4]

// partition splits into pair of lists
val (even, odd) = l.partition { it % 2 == 0 }
print(even) // [2, 4]
print(odd) // [1, 3]

// first/firstBy elements
list.first() // 1
list.firstBy(it % 2 == 0) // 2 (first even number)
```

```
// last/lastBy elements
list.last() // 4
list.lastBy(it % 2 == 0) // 4 (last even number)


// min/max/minBy/maxBy
list.min() // 1, possible because we can compare Int
list.minBy { -it } // 4


// count elements matched by predicate
list.count { it % 2 == 0 } // 2


// sorted/sortedBy returns ordered collection
listOf(2,3,1,4).sorted() // [1, 2, 3, 4]
list.sortedBy { it % 2 } // [2, 4, 1, 3]
```

```
// groupBy groups elements on collection by key
list.groupBy { it % 2 } // Map: {1=[1, 3], 0=[2, 4]}

// distinct/distinctBy returns unique elements
listOf(1,1,2,2).distinct() // [1, 2]
```

Example

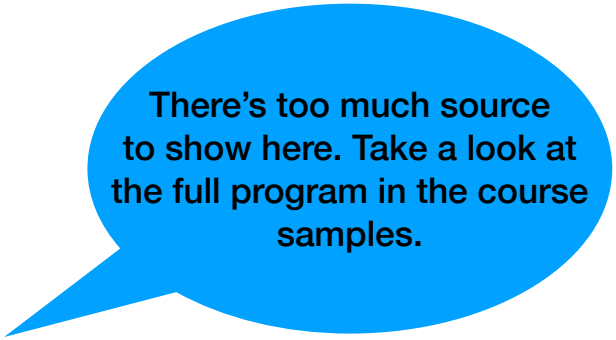
Example: calculator

This is a slightly more complicated example of a calculator. It was written in C++ for CS 247. Let's convert it to Kotlin!

calc/calc.cpp

```
#include <iostream>
#include <string>
using namespace std;

// prototypes
int add(int int1, int int2);
int subtract(int int1, int int2);
int multiply(int int1, int int2);
int divide(int int1, int int2);
```



C++ sample in-use:

```
$ ./calc
Usage: operator int1 int2

$ ./calc 0 1 2
3

$ ./calc 2 1 2
2
```

There's a few limitations of the previous version that we would like to address:

- **Numeric Operators**: Let's use symbols instead (+, -, * /).
- **Argument Order**: Let's use infix notation (e.g. 1 + 2. not + 1 2)

calc/calc.kts

```
if (args.size != 3) {
  println("Usage: number [+|-|*|/] number")
} else {
  val op1 = args.get(0).toInt()
  val operation = args.get(1)
  val op2 = args.get(2).toInt()
  println(
    when(operation) {
      "+" -> op1 + op2
      "-" -> op1 - op2
      "*" -> op1 * op2
      "/" -> op1 / op2
      else -> "Invalid command"
    })
}

$ ./calc.kts 6 * 7
$ 42
```

What is interesting about this code (vs. the original C++ code)?

- We don’t need to import anything (standard libraries included), or declare a namespace.
- ‘when‘ can switch on a ‘String‘ (i.e. it doesn’t require an ‘Integer‘). This is why we can easily add symbolic operators.
- ‘when‘ is also an ‘expression‘, where each case returns a value. This is why we can ‘println‘ the result directly (and why it is easy to eliminate the functions)

What are some of the problems with our implementation?

- It doesn’t handle Floating point numbers.
- It doesn’t handle errors very well. Try ”1 + b” and see what happens.
- What else?