Kotlin
Some useful stuff

Alessio Vecchio

Generics

- Basic use of generic types is similar to Java

```
// A generic class with a type parameter T
class Box<T>(t: T) {
  var value: T = t
}

// Declaring a generic function
fun <T> boxIn(value: T) = Box(value)

fun main() {
  // Creating instances of Box with different types
  var intBox = Box<Int>(1)
  val stringBox = Box<String>("Hello")
  val anotherIntBox = boxIn<Int>(2)
  val anotherStringBox = boxIn<String>("World")

  // intBox = stringBox // Error: Type mismatch
  // type inference
  val intBox2 = Box(1) // infers type as Int
  val stringBox2 = Box("Hello") // infers type as String
  val anotherIntBox2 = boxIn(2) // infers type as Int
  val anotherStringBox2 = boxIn("World") // infers type as String

  // Returned type is different for each instance
  val i1: Int = intBox.value
  val s1: String = stringBox.value

  println(intBox.value) // Prints 1
  println(stringBox.value) // Prints Hello
}
```

lateinit

- property will have a non-null value but not known when object is created
- not for primitive values

```
class Element(val x: Int) {
  fun printX() {
    println(x)
  }
}
class MyClass() {
  lateinit var e1: Element
  var e2: Element? = null
  fun onCreation(e1: Element, e2: Element?) {
    this.e1 = e1
    this.e2 = e2
  }
  fun exampleFunction() {
    e1.printX()
    e2?.printX()
  }
}
fun main(args: Array<String>) {
  val m = MyClass()
  m.onCreation(Element(5), Element(10))
  m.exampleFunction()
}
```

by

- storage of a property value is delegated

```
import kotlin.reflect.KProperty

class MyDelegate {

  var counter: Int = 0

  operator fun getValue(thisRef: Any?, property: KProperty<*>): Int {
    println("getValue: ${property.name} in $thisRef")
    return counter++
  }

  operator fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
    println("$value has been assigned to '${property.name}' in $thisRef.")
    counter = value
  }
}
```

by

```
class Example {
  var x: Int by MyDelegate()
}

fun main() {
  val e = Example()
  println(e.x)
  println(e.x)
  e.x = 10
  println(e.x)
}
```

```
getValue: x in Example@722c41f4
0
getValue: x in Example@722c41f4
1
10 has been assigned to 'x' in Example@722c41f4.
getValue: x in Example@722c41f4
10
```

by lazy

- lazy initialization of a val

```
class Example {
  val x: Int by lazy {
      println("First time")
      (0..100).random()
    }
}

fun main() {
  val e = Example()
  println(e.x)
  println(e.x)
}
```

First time
19
19

with

- *with* is actually a function that takes as input
  - an object
  - a lambda
- In the lambda *this* points to the object
- the return value is the lambda result

```
class X(var y: Int = 0) {
  fun inc() {
    y++
  }
}

fun main() {
  val x = X()
  x.inc()
  println(x.y)
  with(x) {
    inc()
    println(y)
  }
  val w = with(x) {
    inc()
    y
  }
  println(w)
}
```

apply

- within the lambda, ***this*** is the object *apply* is called on
- the object is returned
- generally used for initialization

```
class X(var y: Int = 0) {
  fun inc() {
    y++
  }
}

fun main() {
  val x1 = X()
  x1.inc()
  x1.y += 10
  println(x1.y)

  val x2 = X().apply {
    inc()
    y += 10
  }
  println(x2.y)
}
```

also

- In the lambda, the object *also* is called on becomes available as ***it***
- returns the object itself

```
fun main() {
  val l1 = mutableListOf(1, 2, 3).also {
    it.add(4)
    it.add(5)
    for (i in it) {
      println(i)
    }
  }
}
```

run

- Inside the lambda, the object run is called on is available as ***this***
- returns the result of the lambda

```
fun main() {
  val l1: Int = mutableListOf(1, 2, 3).run {
                  add(4)
                  add(5)
                  sum()
                }
  println(l1) // Prints 15
}
```

run

- run can also be used without any object, to execute statements where an expression is needed
- In this case there is no *this*

```
fun main() {
  val a = listOf(1, 2, 3)
  val b = run{
          var tmp = 0
          for (i in a) {
            tmp += i
          }
          tmp
        }
  println(b) // Prints 6
}
```

let

- Inside the lambda, the object run is called on is available as ***it***
- returns the result of the lambda

```
fun main() {
  val x: Int = mutableListOf(1, 2, 3).let {
                 it.add(4)
                 it.add(5)
                 it.sum()
               }
  println(x) // Prints 15
}
```