kotlin

see: https://kotlinlang.org/docs/reference/

I. Variables

use val for immutable identifiers
- assign once (when declared or later)

use var for variables

declare as: [val | var] <name> : <type> { = <value> }

```
EXs:  val size    : Int = 12
      var count   : Int = 0
      var the_age : Int
```

Much more on types at:
https://kotlinlang.org/docs/reference/basic-types.html

II. Main Function

- execution starts here

- template

```
fun main(args : Array<String>) {
  if (args.size == 0) {
    println("Please provide a name as a command-line argument")
    return
  }
  println("Hello, ${args[0]}!")
}
```

III. Functions (from: https://kotlinlang.org/docs/reference/functions.html)

A) use keyword *fun*

```
fun name(<parameters>): <return type> {
      <body>
}
```

a) parameters

```
      name: type   or  name: type = <default value>
```

b) return explicit: return <value>

```
                 return a > b

          implicit: use value from last statement execute
                 a > b
```

if no return value, indicate with *Unit* return type

```
            fun display(Int x, String name): Unit {
                  :
                  :
            }
```

if no return type specified then Unit is assumed

EX:
```
fun best(a: Int, b: Int): Boolean {
      a > b
}
```

c) Function with an *expression body*, return type is implied

```
fun add(x: Int, y: Int) = x + y
```

d) Default parameters

final parameters need not be specified if defaults used

```
fun doStuff(a: Int = 1, b: Int = 2, c: Int = 3) …
```

The calls doStuff(10, 20, 30), doStuff(0, 4), doStuff(12), and doStuff() are all appropriate.

Supplied actual parameters are assigned to a, b, c in order.

i) using named parameters:

you can also indicate actual to formal parameter assignment by using parameter names, as in:

```
doStuff(9, c = 11)  <— a is 9, b uses default 2, c is 11
```

B) Infix functions

You can write your own infix functions (like + or / that are built in)

For example, the exponentiation function can be defined as:

==========

```
fun main(args : Array<String>) {
      val result: Int = 2 powerof 4
      println("The result is $result")
}

infix fun Int.powerof(exp: Int): Int {
      if (exp == 0) {
            return 1;
      } else if (exp > 0) {
            var ans: Int = this
            var count: Int = exp - 1
            while (count > 0) {
                  ans *= this
```

```
                    count--
                }
                return ans;
            } else return 0;
    }
    ====================================
```

C) Extension Functions

- add your own functions into existing classes
  (alternative to subclassing)

```
fun String.hello() {
  println("Hello, $this!")
}

fun main(args : Array<String>) {
  "world".hello() // prints 'Hello, world!'
}
```

D) *Spread* operator

```
val a = arrayOf(1,2,3)
```

The Spread operator * unpacks the array as individual elements

```
fun streak(Int x, Int y, Int z): Boolean {
      x + 1 == y && y + 1 == z
}
```

The call, streak(*a), maps x to 1, y to 2, and z to 3

IV. Kotlin Basics

A) Types

no primitives - all values are objects

types can be explicit or inferred

```
var isOK: Boolean = true
val sum = 0 //type Int is inferred
```

B) Comments

```
// comment to end of line

/*  multi-line comment (these nest)  */
```

C) String templates

```
var a = 1
// simple name in template:
val s1 = "a is $a"
```

```
a = 2
// arbitrary expression in template:
val s2 = "${s1.replace("is", "was")}, but now is $a"
```

Strings may contain template expressions, i.e. pieces of code that are evaluated and whose results are concatenated into the string. A template expression starts with a dollar sign ($) and consists of either a simple name:

```
val i = 10
val s = "i = $i" // evaluates to "i = 10"
```

or an arbitrary expression in curly braces:

```
val s = "abc"
val str = "$s.length is ${s.length}" // evaluates to "abc.length is 3"
```

Templates are supported both inside raw strings and inside escaped strings. If you need to represent a literal `$` character in a raw string (which doesn't support backslash escaping), you can use the following syntax:

```
val price = """
${'$'}9.99
"""
```

D) Conditional Assignment

```
if (x > y) {
      value = x
} else if (x == y) {
      value = 0
} else {
      value = y
}
```

OR

```
val value = if (x > y) {
            x
      } else if (x == y) {
            0
      } else {
            y
      }
```

E) Null???

Variables that might hold a null value must be marked as such (same for function return values).

```
val thing1: Thing? = abc.getThing()
```

```
if (thing != null) {
      // use thing here - we know it's OK
      answer = thing.getAnswer()
}

      OR

if (thing == null) {
      return
}
// use thing here - we know it's OK
answer = thing.getAnswer()

      OR

answer = thing?.getAnswer()   // returns null if thing is null
```

If you want to assign a non-null value for a null reference, use the Elvis operator ?: as in:

```
val size = thing?.getSize() ?: 0
```

If you are sure the value will not be null, then you can insist with !!

```
val size = thing!!.getSize()
```

Casting

If casting fails then null is returned - use as?

```
val num: Int? = a as? Int
```

F) Loops

For loops:

```
for (item: <type> in <collection> {
      :
      :
}
```

EXs:

```
for (value: Int in myIntArray)

for ((idx, value) in myIntArray.withIndex())
```

Using ranges with a for-loop

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

Other use for ranges

```
if (x in 1..10) { ... }
```

G) While loops - just like Java and C and …
Includes *break* and *continue*

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

H) The *when* statement - (like a switch)

a) Like a simple switch - don't need a break statement

```
when (x) {
      0, 1 -> print("x == 0 or x == 1")
      2     -> print("x is two")
      else -> print("otherwise")
}
```

b) using arbitrary expressions as branch conditions

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

c) checking a value for being in or !in a range or a collection:

```
when (x) {
      in 1..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")
}
```

V. Object-Oriented Programming

A) Overview

```
class Meal {   // this line is the class header
                  // it can contain primary-constructor parameters
                  // these parameters can be used as instance variables
}
```

Expanded example

```
class Meal(val name: String) {//initialization code is in init blocks
```

```
        val description = "This is the $name meal"
        var calories
        init {
            calories = 0
        }
    }
```

B) Secondary Constructors

prefix with keyword constructor

```
class Abc {
    constructor(size: Int) {
        :
    }
  :
}
```

If the class has a primary constructor it must always be used even when a secondary constructor is called. This action is accomplished through *delegation*, a chain of constructor calls ending with the primary constructor. Use keyword *this* to call other constructors.

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

C) Creating an Object (class instance)

```
val weekendMeal = Meal("Brunch")    // no keyword new
```

D) Inheritance

a) Common superclass, ***Any***

b) Indicate your own superclass at end of header.

```
class Banquet(val event : String) : Meal(event) {
```

c) All classes are *final* unless the open keyword is used
The keyword *override* is used by the subclass

```
open class Base {
    open fun v() {}
    fun nv() {}
}
class Derived() : Base() {
    override fun v() {}
}
```

d) refer to superclass with keyword *super*

```
override fun go() {
      super.go()
      :
}
```

e) anonymous inner classes implemented with object declarations using the keyword, *object*

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

VI. Accessing Java from Kotlin

EX: using Java ArrayLists

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source) {
        list.add(item)
    }
    // Operator conventions work as well:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // get and set are called
    }
}
```

X. Getting Started

A. Follow Tutorial at:
https://kotlinlang.org/docs/tutorials/kotlin-android.html

B. Follow Directions at: https://developer.android.com/kotlin/index.html