

Kotlin.

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
Maestría en Computación
MC-8812 Teoría de los Lenguajes de Programación

CARLOS MARTÍN FLORES GONZÁLEZ, Carné: 2015183528

ÍNDICE

Índice	1
1. Historia	3
2. Características del Lenguaje	3
2.1. Valores y tipos de datos base	3
2.1.1. Números	3
2.1.2. Caracteres	3
2.1.3. Booleanos	3
2.1.4. Arrays	4
2.1.5. Strings	4
2.1.6. String Literals	4
2.1.7. String Templates	4
2.2. Variables y almacenamiento	4
2.3. Estructuración de datos	4
2.3.1. object: declaración e instanciación combinada de una clase	4
2.4. Declaraciones, manejo del alcance de los identificadores	5
2.4.1. Declaración de tipos	5
2.4.2. Declaración de constantes	7
2.4.3. Declaración de variables	7
2.4.4. Declaración de funciones y procedimientos	7
2.4.5. Manejo del alcance de los identificadores	8
2.5. Estructuras de control	8
2.5.1. Expresión when	8
2.5.2. Ciclos for	8
2.5.3. Ciclos while	9
2.6. Secuenciadores	9
2.7. Mecanismos de abstracción	10
2.8. Mecanismos de modularización	10
2.9. Soporte a concurrencia, paralelismo, distribución	11
2.9.1. Corutinas (<i>Coroutines</i>)	11
2.10. Sistemas de tipos	12
2.10.1. Herencia	12
2.10.2. Sobreescritura de Métodos	12
2.10.3. Extensiones	13
2.11. Genericidad	13
2.11.1. Varianza	13
2.11.2. Funciones genéricas	14
2.12. Soporte a paradigmas	15
2.13. Soporte a “programación en grande”	15

2.14.	Peculiaridades	16
3.	Ejemplos	16
4.	Análisis y conclusiones	16
	Referencias	16

1. HISTORIA

La idea de Kotlin se concibió en el 2010 en la compañía JetBrains¹, fabricantes de herramientas de desarrollo para muchos lenguajes incluido Java, C#, JavaScript, Python, Ruby y PHP[1]. El Proyecto Kotlin fue revelado en Julio del 2011. Era un nuevo lenguaje para la *Java Virtual Machine* (JVM) que había estado en desarrollo por un año[2]. Dentro de la compañía se acusaba que muchos lenguajes de programación no contaban con las características que ellos estaban buscando, con la excepción de Scala, pero el lento tiempo de compilación de Scala era una deficiencia obvia. Uno de los objetivos de Kotlin era contar con un tiempo de compilación tan rápido como en el de Java.

De acuerdo con Dmitry Jemerov, ingeniero en JetBrains [1]:

“La experiencia en la construcción de herramientas para un conjunto diverso de lenguajes nos ha dado una perspectiva y entendimiento único en el diseño de lenguajes pero, a pesar de esto varios de nuestros entornos de desarrollo estaban contruidos en Java. Teníamos cierta envidia de nuestro equipo de .Net quienes desarrollaban en C#, un lenguaje rápido, moderno y de rápida evolución. Pero no encontrábamos ningún lenguaje que pudiéramos usar en lugar de Java.

¿Cuáles eran nuestro requerimientos para este lenguaje? Lo primero y lo más obvio era tipos estáticos. No conocemos ninguna otra forma de desarrollar miles de millones de líneas de código sin volvernos locos. Segundo, necesitábamos total compatibilidad con el código Java existente. El código es un gran activo para JetBrains, y no nos podíamos permitir perderlo o devaluarlo debido a dificultades en interoperabilidad. En tercer lugar, no queríamos aceptar ningún compromiso en términos de calidad de herramientas. La productividad del desarrollador es el valor más importante para una compañía como JetBrains, y el contar con buenas herramientas es esencial para lograr esto. Por último, necesitábamos un lenguaje que fuera fácil de leer y de razonar. Con esto en mente nos decidimos a embarcarnos en un proyecto para crear un nuevo lenguaje: Kotlin”.

Kotlin v1.0 fue lanzado en Febrero del 2016[3]. En Google I/O 2017, Google anunció soporte para Kotlin en Android[4]. Kotlin v1.2 se lanzó en Noviembre del 2017. Compartir código entre la JVM y JavaScript fue una característica agregada a esta version[5].

Kotlin lleva el nombre de una isla cerca de San Petersburgo², Rusia, donde se encuentra la mayor parte del equipo de desarrollo de Kotlin[1].

2. CARACTERÍSTICAS DEL LENGUAJE

2.1. Valores y tipos de datos base

En Kotlin, todo es un objeto en el sentido que se pueden invocar funciones y propiedades en cualquier variable. Algunos de los tipos pueden tener una representación interna especial (por ejemplo, números, caracteres y booleanos pueden ser representados como tipos primitivos en tiempo de ejecución) pero que para el usuario lucen como una clase ordinaria.

2.1.1. *Números.* Kotlin proporciona las siguientes tipos para representar números:

Double 64bit, Int 32bit, Float 32bit, Short 16bit, Long 64bit, Byte 8bit

2.1.2. *Caracteres.* Son representados por el tipo Char. No pueden ser tratados directamente como números. Los literales van en comillas simples: '1'. Caracteres especiales pueden ser escapados usando un *backslash*. Las siguientes secuencias de escape son soportadas:

\t, \b, \n, \r, \', \", \\\, \\$

2.1.3. *Booleanos.* El tipo Boolean representa booleanos y tiene dos tipos true y false.

¹<https://www.jetbrains.com/>

²https://en.wikipedia.org/wiki/Kotlin_Island

2.1.4. Arrays. Los *Arrays* en Kotlin se representan por la clase `Array`, que tiene la propiedad `size` y funciones `get`, `set`, entre otras que son útiles para trabajar con este tipo de estructura.

2.1.5. Strings. Se representan con la clase `String`. Los *String* son inmutables. Los elementos de un `String` son caracteres que puede ser accedidos por la operación de indexación `s[i]`. Un `String` puede ser recorrido por medio de un ciclo `for-loop`.

2.1.6. String Literals. Kotlin tiene dos tipo de *string literals*: *strings* escapados que pueden tener caracteres escapados en él y *strings* crudos (*raw*) que pueden contener varias líneas y texto arbitrario.

```
val s = "Hello, world!\n"
val text = """
    for (c in "foo")
        print(c)
    """
```

2.1.7. String Templates. Los *strings* pueden contener expresiones emplantilladas, piezas de código que puede ser evaluadas y cuyos resultados se concatenan dentro del *string*.

```
val i = 10
print("i = $i") //imprime "i = 10"
```

2.2. Variables y almacenamiento

Para declarar una variable en Kotlin, se inicia con el nombre del identificador se puede o no poner el tipo luego del nombre

```
val question = "The Ultimate Question of Life, the Universe, and Everything"
val answer: Int = 42
```

Si no se especifica el tipo, el compilador analiza la expresión y su inicializador, y usa el tipo como el tipo de variable[1]. En el ejemplo anterior, 42, el inicializador tiene un tipo `Int`, por lo tanto la variable va a tener el mismo tipo. Si la variable no se inicializa, se necesita especificar su tipo explícitamente:

```
val answer: Int
answer = 42
```

Existen dos palabras reservadas para declarar una variable:

1. **val** (de valor – *value*). Referencia Inmutable. Una variable declarada con `val` no puede ser reasignada luego de ser inicializada. Corresponde a una variable final en Java.
2. **var** (de variable). Referencia mutable. El valor de una variable puede ser cambiado. Esta declaración corresponde a un variable regular (no final) en Java.

Almacenamiento. Debido a que Kotlin es un lenguaje construido por encima de la JVM delega el manejo del almacenamiento de variables, estructuras de datos y de control a esta. Una breve introducción sobre el manejo de la memoria en la JVM se puede encontrar en Apéndice 1.

2.3. Estructuración de datos

2.3.1. object: declaración e instanciación combinada de una clase. La palabra reservada `object` define una clase y crea una instancia (es decir, objeto) de esa clase al mismo tiempo. Las diferentes formas en las que puede ser utilizado son:

Como una declaración de un objeto. es una forma de definir un singleton, una clase para cual se necesita solo una instancia. Kotlin proporciona soporte para esto por medio de la declaración de un

objeto. La declaración de objeto combina la declaración de una clase y la declaración de una sola instancia de la clase.

```
object Payroll {
    val allEmployees = arrayListOf<Person>()

    fun calculateSalary() {
        for (person in allEmployees) {
            ...
        }
    }
}
```

Tal y como pasa con las clases, las declaraciones de objetos puede contener declaraciones de propiedades, métodos, bloques inicializadores, entre otros. Lo único que no se permite son constructores. A diferencia de las clases normales, las declaraciones de objetos se crean inmediatamente después del punto de definición y no a través de llamadas a constructores. Tal y como una variable, la declaración de un objeto permite llamar métodos y acceder a las propiedades por medio del nombre del objeto a la izquierda seguido del punto(.).

```
Payroll.allEmployees.add(Person(...))
Payroll.calculateSalary()
```

Las declaraciones de objetos pueden también heredar de clases e interfaces.

Companion objects: utilizados como *factory methods* y miembros estáticos. Las clases en Kotlin no pueden tener miembros estáticos. A modo de reemplazo, Kotlin utiliza funciones a nivel de paquetes (que pueden reemplazar los métodos estáticos que hay en Java en muchas situaciones) y declaraciones de objetos (que reemplazan propiedades estáticas). Uno de los objetos definidos en una clase puede ser utilizar la palabra reservada: `companion`. Al hacerlo, se gana la habilidad de acceder a los métodos y propiedades de este objeto directamente a través del nombre de la clase que lo contiene. La sintaxis resultante luce exactamente como una invocación a un método estático en Java.

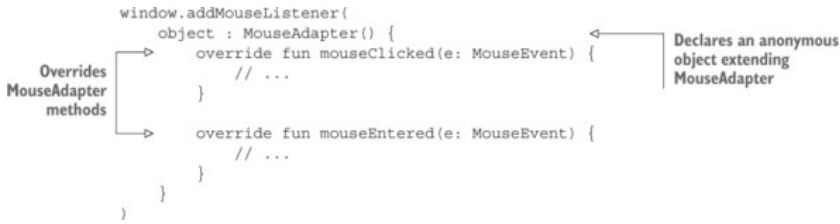
```
class A {
    companion object {
        fun bar() {
            println("Companion object called")
        }
    }
}
```

```
>>> A.bar()
Companion object called
```

Object expression. La palabra reservada `object` puede ser usada para declarar objetos anónimos. Los objetos anónimos reemplazan el uso de clases anónimas internas en Java. En la figura 1 se puede ver como se puede convertir un uso típico de una clase anónima interna en Java - un *event listener* - a Kotlin.

2.4. Declaraciones, manejo del alcance de los identificadores

2.4.1. Declaración de tipos. Nuevos tipos son introducidos en Kotlin a través de clases, Data Classes, Enum Classes, interfaces y objetos.

Fig. 1. *Object Expression*. Fuente [1]*Declaración de una clase.*

```

class Invoice {
}
// Clase con constructor primario
class User(_nickname: String) {
    val nickname = _nickname
}
//
val invoice = Invoice()
val user = User("Luigi")

```

Data Classes. Clases cuyo propósito principal es contener datos. En una clase de este tipo, algunas funcionalidades estándar y funciones de utilidad a menudo se derivan mecánicamente de los datos.

```

data class User(val name: String, val age: Int)
val admin = User("John", 40)

```

El compilador automáticamente deriva los siguientes miembros para todas las propiedades declaradas en el constructor primario: `equals()`, `hashCode()`, `toString()`, `copy()`. Para asegurar la consistencia y significado del código generado, las *data classes* tienen que tener los siguientes requisitos:

- El constructor primario necesita tener al menos un parámetro
- Todos los parámetros del constructor primario necesitan estar marcados con `val` o `var`
- *Data classes* no pueden ser abstractas, internas o selladas (*sealed*).

Enum Classes.

```

enum class Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
//
val primaryColor = Color.GREEN

```

Interfaces. Interfaces en Kotlin son similares a las de Java8: pueden contener definiciones de métodos abstractos así como implementaciones de métodos no abstractos (similar a los métodos *default* de Java8), pero no pueden contener ningún estado.

```

interface Clickable {
    fun click()
}

class Button : Clickable {

```


2.4.5. *Manejo del alcance de los identificadores.* Kotlin no introduce ningún cambio en el manejo del alcance de los identificadores en comparación con Java. Los identificadores “viven” dentro del ámbito en el que fueron declaradas (clases, funciones, bloques).

2.5. Estructuras de control

Expresión if. Uso tradicional

```
var max = a
if (a < b) max = b
```

Utilizando else.

```
var max : Int
if (a > b) {
    max = a
} else {
    max = b
}
```

Como expresión.

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

2.5.1. *Expresión when.* reemplaza al operador *switch* de lenguajes basados en C. Evalúa el argumento contra todos los caso de manera secuencial hasta que se cumpla una de las condiciones. El caso *else* se evalúa si ninguno de los otros casos se pudo satisfacer. Puede utilizar expresiones arbitrarias (no solo constantes) y también se puede evaluar si un valor se encuentra dentro de un rango.

```
when (x) {
    1          -> print("x == 1")
    2          -> print("x == 2")
    3, 4       -> print("x == 3 or x == 4")
    in 5..10   -> print("x is between 5 and 10")
    !in 20..30 -> print("x is outside the range")
    else -> {
        print("none of the above")
    }
}
```

2.5.2. *Ciclos for.* Tiene una forma equivalente al ciclo *for-each* de Java.

```
for (item: Int in ints) {
    print(item)
}
```

Se puede iterar sobre un rango de números:

```
for (i in 1..3) {
    println(i)
}
for (i in 6 downTo 0 step 2) {
    println(i)
}
```


2.5.3. *Ciclos while*. Tanto el ciclo `while` como el `do..while` trabajan de la misma forma que en Java:

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

do {
    val y = retrieveData()
} while (y != null) // y es visible aquí
```

2.6. Secuenciadores

Kotlin tiene tres expresiones para saltos/continuaciones:

- `return`: retorna de la función más cercana
- `break`: termina un ciclo
- `continue`: continua con el siguiente paso de un ciclo

`break` y `continue` *con etiquetas*. Cualquier expresión en Kotlin puede ser etiquetada. Las etiquetas tienen un identificador seguido del signo `@`. Para etiquetar una expresión, se apunta una etiqueta en frente de ella

```
loop@ for (i in 1..100) {
    //...
}
```

Se puede marcar un `break` o un `continue` con una etiqueta:

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

Un `break` marcado con una etiqueta salta al punto de ejecución justo después del ciclo que fue marcado con esa etiqueta. Un `continue` prosigue con la próxima iteración del ciclo.

Excepciones. Todas las excepciones en Kotlin son descendientes de la clase `Throwable`. Todas las excepciones tienen un mensaje, un *stack trace* y una causa (opcional). Para lanzar una excepción se usa la expresión `throw`

```
throw MyException("Hi There!")
```

Para atrapar una excepción, se usa la expresión `try`:

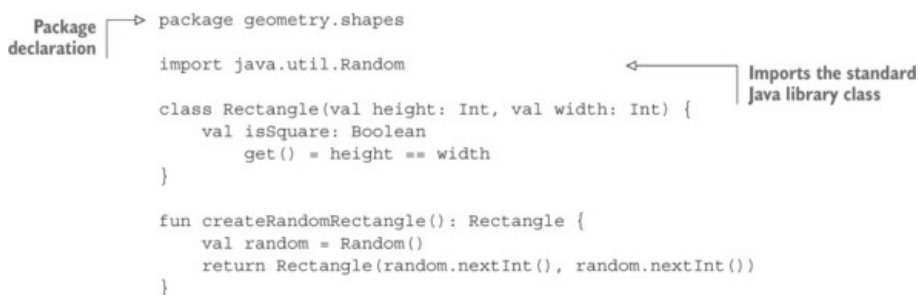
```
try {
    // algún código
} catch (e: SomeException) {
    // código manejador de excepción
} finally {
    // bloque finalizador opcional
}
```

Pueden haber uno o varios bloques `catch`. Los bloques `finally` pueden ser omitidos.

2.7. Mecanismos de abstracción

2.8. Mecanismos de modularización

Java organiza todas las clases en paquetes. Kotlin también tiene el concepto de paquetes. Cada archivo Kotlin puede tener una declaración de `package` al inicio y, todas las demás declaraciones (clases, funciones y propiedades) definidas en el archivo serán colocados bajo ese paquete. Las declaraciones definidas en otros archivos puede ser usados directamente si ellos están en el mismo paquete. Si se encuentran en diferentes paquetes necesitan ser importados. Tal y como en Java las declaraciones de importación se colocan al inicio del archivo por medio de la palabra reservada `import`. En la figura 3 se muestra un ejemplo de una declaración de `package` e `import` en un archivo.



```

package geometry.shapes

import java.util.Random

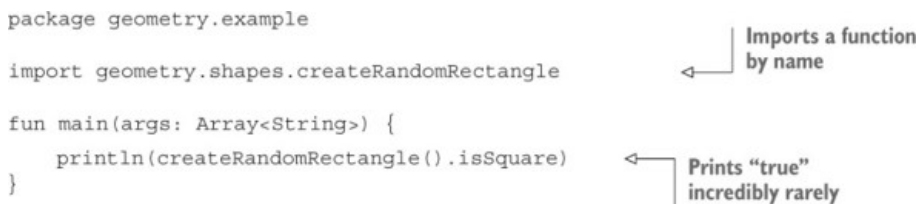
class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
    get() = height == width
}

fun createRandomRectangle(): Rectangle {
    val random = Random()
    return Rectangle(random.nextInt(), random.nextInt())
}

```

Fig. 3. Declaración de una clase y una función en un paquete. Fuente [1]

Kotlin no hace distinción entre importar clases y funciones, y esto permita importar cualquier clase de declaración usando la declaración `import`.



```

package geometry.example

import geometry.shapes.createRandomRectangle

fun main(args: Array<String>) {
    println(createRandomRectangle().isSquare)
}

```

Fig. 4. Importando una función desde otro paquete. Fuente [1]

También se puede importar todas las declaraciones de un paquete en particular poniendo `.*` luego del nombre del paquete. Este *star import* hace visible no solamente las clases definidas del paquete sino también funciones y propiedades. Si en la figura 4 se escribiera `import geometry.shapes.*` en lugar de la importación explícita, el código continuaría siendo correcto.

En Kotlin, se pueden poner varias clases en el mismo archivo y se escoger cualquier nombre para ese archivo. Kotlin no impone ninguna restricción en cómo se disponen los archivos en el disco. Se puede usar cualquier estructura de directorios para organizar los archivos. Sin embargo, en la mayoría de los casos es una buena práctica seguir la estructura de directorios de Java para organizar los archivos. Esta estructura se hace particularmente importante en proyectos en donde se combina código Java y Kotlin porque ayuda a migrar el código gradualmente.

2.9. Soporte a concurrencia, paralelismo, distribución

Al ser 100 % interoperable con Java, en Kotlin se pueden utilizar todas las librerías (*Futures*, *CompletableFuture*, *Reactive Extensions*) para concurrencia, paralelismo y programación asíncrona en general que son utilizadas en Java.

2.9.1. Corutinas (Coroutines). A partir de la versión 1.1 de Kotlin se introdujo el concepto de Corutina que proporciona una forma de evitar bloquear un hilo(*thread*) y reemplazarlo con una operación más económica y de mayor control: la *suspensión* de una corutina.

Básicamente, las corutinas son cálculos que pueden ser *suspendidos* sin bloquear un hilo. El bloqueo de un hilo es usualmente una operación cara en términos de CPU y memoria, especialmente bajo grandes cargas de trabajo, porque solamente un número relativamente pequeño de hilos puede ser mantenido, por lo que bloquear uno de ellos lleva a que se retrase un trabajo importante.

Por otro lado, la suspensión de una corutina es casi gratis. No hay cambios de contexto o cualquier otro tipo de involucramiento del sistema operativo es requerido. Aparte de esto, la suspensión puede ser contralada por el código del usuario y de esta forma decidir qué pasa con la misma y de esta forma optimizar, interceptar o capturar una excepción de acuerdo al escenario dado.

Otra diferencia es que las corutinas no pueden ser suspendidas a partir de cualquier instrucción, sino solamente con los llamados puntos de suspensión(*suspension points*), que son llamados a funciones con anotadas de manera especial(*suspend*).

Suspendiendo funciones. una suspensión pasa cuando se llama a una función marcada con la palabra reservada *suspend*:

```
suspend fun doSomething(foo: Foo): Bar {
    ...
}
```

Estas funciones son llamadas *suspending functions* porque las llamadas a ellas pueden suspender una corutina. Las *suspending functions* pueden tomar parámetros y retornar valores de la misma forma que las funciones normales, pero solamente pueden ser llamadas desde corutinas y otras *suspending functions*.

De hecho para iniciar una corutina, debe haber al menos una *suspending function*, y es usualmente una función lambda. El siguiente es un ejemplo simplificado de la función *async*³.

```
fun <T> async(block: suspend() -> T)
```

En el ejemplo, *async* es una función normal pero el parámetro *block* es de tipo función con la palabra reservada *suspend*: *suspend() ->T*. De esta forma, cuando se pasa una función lambda a *async*, es una *suspending lambda*, y se puede invocar una *suspending function* desde ahí.

RxKotlin[6]. La programación reactiva es un paradigma de programación asíncrona que gira en torno a flujos de datos *streams* y propagación de cambios. Los programas que propagan todos los cambios que afectaron sus datos o sus flujos de datos (a usuarios finales, componentes u otro programa relacionado) son llamados programas reactivos.

RxKotlin es una implementación específica de programación reactiva para Kotlin, que está influenciada en programación funcional. Favorece composición de funciones, evita el manejo de estados globales y efectos secundarios. Se basa en el patrón observador → productor/consumidor, con muchos operadores que permiten componer, calendarizar(*scheduling*), transformar y gestionar errores.

³De la librería `kotlinx.coroutines`

2.10. Sistemas de tipos

Como se señaló en la sección 2.4, nuevos tipos son introducidos en Kotlin a través de clases, Data Classes, Enum Classes, interfaces y objetos.

Modificadores de acceso para clases: open, final y abstract.

- final: No puede ser sobre escrita. Utilizada por defecto para los miembros de clase
- open: Puede ser sobreescrita. Tiene que se especificada explícitamente
- abstract: Debe ser sobre escrita. Puede ser utilizada solamente en clases abstractas, los miembros abstractos no pueden tener una implementación
- override: sobre escribe un miembro de una super clase o interfaz. El miembro sobreescrito está abierto por defecto, sino está marcado como final.

Modificadores de visibilidad:

- public (por defecto). Visible en todas partes
- internal. Visible en un módulo
- protected. Visible en subclases
- private. Visible en una clase

2.10.1. *Herencia.* Todas las clases en Kotlin tienen una superclase Any en común, esta es la superclase por defecto para una clase sin ningún super tipo declarado:

```
class Example // Implícitamente hereda de Any
```

Para declarar un super tipo de forma explícita, se pone el tipo después de dos puntos en el encabezado de la clase:

```
open class Base(p: Int)
```

```
class Derived(p: Int) : Base(p)
```

Si la clase derivada tiene un constructor primario, la clase base puede (y debe) ser inicializada de una vez, utilizando los parámetros del constructor primario. Si la clase no tiene un constructor primario, entonces cada constructor secundario tiene que inicializar el tipo base utilizando la palabra reservada super, o bien, delegar a que otro constructor lo haga.

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

2.10.2. *Sobreescritura de Métodos.* Kotlin requiere anotaciones explícitas para miembros que pueden ser sobreescritos(open) y para las sobreescrituras como tal:

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

La palabra reservada override es requerida en Derived.v(). Si no hay una anotación open en una función, como en Base.nv(), la declaración de un métodos con la misma firma en una subclase sería ilegal.

2.10.3. Extensiones. De manera similar a C#, Kotlin proporciona la habilidad de extender una clase con nueva funcionalidad sin tener que heredar de la clase o usar algún tipo de patrón de diseño como el patrón decorador⁴. Esto se realiza por medio de una declaración especial llamada extensión. Kotlin soporta funciones y de propiedades extendidas (*extension functions* y *property extension*).

Extension functions. Para su declaración se necesita añadir el tipo que va a ser extendido como prefijo del nombre. El siguiente código añade la función swap a *MutableList<Int>*:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] //'this' corresponde a la lista
    this[index1] = this[index2]
    this[index2] = tmp
}
// Uso
val l = mutableListOf(1, 2, 3)
l.swap(0,2)
```

La palabra reservada *this* dentro de un *extension function* corresponde al objeto que es pasado antes del punto.

Las extensiones son resueltas estáticamente. Las extensiones no modifican las clases con su extensión. Cuando se define una extensión, no se insertan nuevos miembros en la clase sino que se hace que las funciones puedan ser invocadas por medio de la notación basada en punto (*dot notation*) en las variables de este tipo.

Extension properties.

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

2.11. Genericidad

Tal y como el Java, las clases en Kotlin pueden tener tipos parametrizados:

```
class Box<T>(t: T) {
    var value = t
}
//Uso
val box: Box<Int> = Box<Int>(1)
val box: Box(1) // 1 es de tipo Int, así que se puede hacer inferir el tipo
```

2.11.1. Varianza. Una de las partes más difíciles de los tipos de Java son los tipos *wildcard*⁵. Kotlin no tiene ninguno de estos.

Declaration-site variance vs. Java wildcards. Supóngase que se tiene una interfaz *Source<T>* que no tiene ningún método que tome *T* como parámetro, solamente tiene métodos que retornan *T*:

```
//Java
interface Source<T> {
    T nextT();
}
```

⁴Patrón Decorador: https://en.wikipedia.org/wiki/Decorator_pattern

⁵<http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>

Podría ser seguro almacenar una referencia a una instancia de `Source<String>` en una variable de tipo `Source<Object>`, pero Java no sabe de esto y lo prohíbe:

```
//Java
void demo(Source<String> strs) {
    Source<Object> objects = strs // NO se permite en Java
}
```

Para arreglar esto, se tiene que declarar `objects` de tipo `Source<? extends Object>`. En Kotlin, existe una forma de explicar esto al compilador. Se le llama *declaration-site variance*: se puede another el tipo parametrizado `T` de `Source` para asegurarse de que solo se devuelve(produce) a partir de miembros de `Source<T>`. Para hacer esto se proporciona el modificador `out`.

```
interface Source<out T> {
    fun nextT() : T
}

fun demo(strs : Source<String>) {
    val objects : Source<Any> = strs // Esto está bien porque T es un out-parameter
}
```

Al modificador `out` se le llama anotación de varianza y, debido a que es proporcionado en la declaración del tipo parametrizado, se llama *declaration-site variance*. Además de `out`, Kotlin provee una anotación complementaria de varianza: `in`, hace que un tipo parametrizado solo puede ser consumido pero nunca producido.

```
interface Comparable<in T> {
    operator fun compareTo(other: T) : Int
}

fun demo(x: Comparable<Number>){
    x.compareTo(1.0) // 1.0 es de tipo Double, que es un subtipo de Number
    // Por lo tanto, se puede asignar x a una variable de tipo Comparable<Double>
    val y : Comparable<Double> = x // OK!
}
```

Star projections. Existen situaciones en donde no se tiene conocimiento del tipo específico del tipo del valor. Supóngase que solamente se desea imprimir todos los elementos de un arreglo y que no importa el tipo de los elemento en ese arreglo. Para esto se puede utilizar una *star projection*:

```
fun printArray(array: Array<*>) {
    array.forEach { println(it) }
}
// Uso
val array = arrayOf(1, 2, 3)
printArray(array)
```

2.11.2. *Funciones genéricas.* No solamente las clases pueden tener tipos parametrizados, las funciones también pueden tenerlo. Los tipos parametrizados se colocan antes del nombre de la función. En la figura 5 se puede ver un ejemplo.

```
//Uso
val letters = ('a'..'z').toList()
println(letters.slice<Char>(0..2))
```

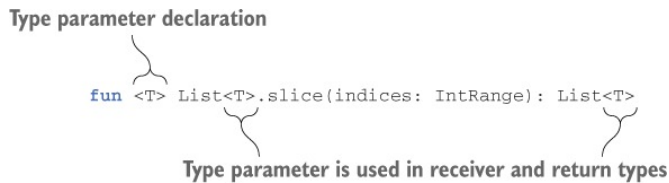


Fig. 5. Declaración de una función genérica en Kotlin. Fuente [1]

```
-> [a, b, b]
println(letters.slice(10..13))
-> [k, l, m, n]
```

2.12. Soporte a paradigmas

Los dos grandes paradigmas soportados por Kotlin son: programación orientada a objetos y programación funcional. Características de programación orientada a objetos se han señalado en secciones anteriores y se ha visto como las clases son consideradas como “ciudadanos de primera clase” para introducir nuevos tipos y modelado de programas a través de interfaces, herencia, tipos parametrizados, entre otros.

Además de las clases, las funciones son también consideradas de primera clase en Kotlin, lo que significa que puede ser almacenadas en variables y estructuras de datos, pasadas como argumentos y retornadas desde otra función de orden superior (*high-order function*: una función que toma funciones como parámetro o retorna una función). Aparte de esto, Kotlin ofrece un amplio conjunto de funcionalidades para programación funcional dentro de las que destacan las expresiones lambda, *data classes* e interfaces de programación para trabajar con objetos y colecciones siguiendo un estilo funcional.

Kotlin permite que se pueda programar de forma funcional pero no obliga a esto. Cuando se escribe código en Kotlin se puede combinar ambos enfoques y usar las herramientas que sean más apropiadas para el problema que se resuelve.

Por último, Kotlin también puede ser utilizado como un lenguaje de *scripting*. Se puede utilizar la utilidad `kotlinc` como un REPL (*Read-Eval-Print Loop*) o bien para ejecutar archivos de *scripting* de Kotlin `.kts`

```
> kotlinc // Inicia el REPL
> kotlinc -script sample.kts // Ejecuta el código Kotlin dentro de sample.kts
```

2.13. Soporte a “programación en grande”

Kotlin ha ganado mucha popularidad en los últimos 3 años. Esto ha sido motivado principalmente por su interoperabilidad con Java además del soporte de primera clase de que le han brindado proyectos tales como Android y *Spring Framework*, dos de los más populares dentro de la comunidad de desarrolladores de Java.

Otra de las ventajas que tiene Kotlin tanto como lenguaje y plataforma es su versatilidad. Con Kotlin se puede programar aplicaciones *backend* utilizando proyectos como *Spring*⁶, *Vert.x*⁷ y *Ktor*⁸. Se pueden programar aplicaciones móviles con Android. Se puede transpilar código Kotlin a JavaScript para hacer aplicaciones del lado del cliente y también para hacer aplicaciones del lado del

⁶<https://spring.io/>

⁷<http://vertx.io/>

⁸<https://github.com/kotlin/ktor>

servidor utilizando NodeJS. Por último el proyecto Kotlin/Native⁹ es una tecnología para compilar Kotlin a binarios nativos que pueden correr sin necesidad de una máquina virtual. Este proyecto está pensado principalmente para permitir la compilación en plataformas en donde las máquinas virtuales no se pueden utilizar como en sistemas embebidos, o cuando el desarrollador necesita producir programas que no requieran de ambientes de ejecución adicionales.

¿*Quiénes usan Kotlin?* Se ha reportado que compañías tales como Google, Amazon, Netflix, Pinterest, Uber, Foursquare, Trello, Capital One, Coursera, Basecamp, Corda, JetBrains, Pivotal, Evernote, entre otros usan Kotlin para el desarrollo de sus aplicaciones.

2.14. Peculiaridades

3. EJEMPLOS

4. ANÁLISIS Y CONCLUSIONES

REFERENCIAS

- [1] Dmitry Jemerov, Svetlana Isakova. *Kotlin in Action*. Manning Publications Company. 2017. ISBN 9781617293290
- [2] Krill, Paul. *JetBrains readies JVM language Kotlin*. infoworld.com. InfoWorld. 2011. Obtenido el 11 de Abril del 2018 de <https://www.infoworld.com/article/2622405/java/jetbrains-readies-jvm-based-language.html>.
- [3] Kotlin Blog. *Kotlin 1.0 Released: Pragmatic Language for JVM and Android*". Febrero, 2015. Obtenido el 11 de Abril del 2018 de <https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/>
- [4] Kotlin Blog. *Kotlin on Android. Now official*. Mayo, 2017. Obtenido el 11 de Abril del 2018 de <https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/>.
- [5] Kotlin Blog. *Kotlin 1.2 Released: Sharing Code between Platforms*. Noviembre, 2017. Obtenido el 11 de Abril del 2018 de <https://blog.jetbrains.com/kotlin/2017/11/kotlin-1-2-released/>.
- [6] Rivu Chakraborty. *Reactive Programming in Kotlin*. Packt Publishing. Diciembre 2017. ISBN: 9781788473026
- [7] Kotlin. *Kotlin Reference*. <https://kotlinlang.org/docs/reference>

⁹<https://github.com/JetBrains/kotlin-native/>