

Introducción a la Programación Orientada a Objetos en Java

Bibliografía

- Programming in Kotlin
Stephen Samuel, Stefan Bocutiu
- Kotlin in Action
Dmitry Jemerov, Svetlana Isakova



Samsung
TECH INSTITUTE

Qué es Kotlin

Lenguaje producido por JetBrains

1. Diseñado para JVM
2. Fácil para los que saben Java
3. Compatible con Java
4. Fuertemente tipificado
5. Compatible con Android

En Google I/O 2017 se anunció soporte oficial a Kotlin como lenguaje de primer nivel para desarrollo en Android (1.2.50)

Contenido

- Clases en Kotlin.
 - Constructores, propiedades y funciones
 - Visibilidad
- Creación de Objetos
- Data clases
- Herencia
 - Vinculación
- Interfaces
- Más sobre data clases
- Excepciones
- Operadores
 - equals y hashCode
 - Clases ordenables
- Interoperabilidad Java - Kotlin
- Genericidad
- Varianza y contravarianza
- Métodos extensión
- Colecciones
- Singleton y companion
- Lambdas y Funciones de orden superior

Tipos Básicos en Kotlin

No existen los tipos básicos.

| <u>Java (tipos básicos)</u> | <u>Kotlin (clases)</u> |
|-----------------------------|------------------------|
| byte | Byte |
| int | Int |
| short | Short |
| long | Long |
| double | Double |
| boolean | Boolean |
| char | Char |

Mas sobre tipos.

Declaración de variables

```
val i : Int = 4
```

```
var d : Double = 4.6
```

- Una variable declarada como **val** puede ser accedida pero no modificada.
- Una variable declarada como **var** puede ser accedida y modificada.
- Los tipos pueden ser inferidos:

```
val i = 4 // i es un Int
```

```
val d = 4.6 // d es un Double
```

- La igualdad es **===**

Mas sobre tipos.

```
val i : Int = 4  
println(i.hashCode())  
val res : Int = i.compareTo(6)  
val sol : Int = 6.compareTo(i)
```

Clases en Kotlin

```
class Nombre {  
    var x: Int = 5  
    fun cambiaX(xx:Int) {  
        x = xx;  
    }  
}
```

- Las variables de instancia se llaman ahora **propiedades**
- Los métodos se llaman **funciones**
- Además, puede haber propiedades y funciones globales (fuera de cualquier clase)

Clases en Kotlin

```
class Nombre {  
    var x: Int = 5  
    fun cambiaX(xx:Int) {  
        x = xx;  
    }  
}
```

Todas las clases, funciones y propiedades de Kotlin son por defecto **public**. Pueden ser también **private**, **internal** o **protected**.

- **private** Visible dentro de la clase.
- **protected** Como private + visible en subclase.
- **internal** Visible a clientes dentro de este módulo.
- **public** Visible a cualquier cliente.

Constructores y propiedades

- Toda clase Kotlin tiene un constructor primario y cero o más secundarios.
- Las clases pueden incluir propiedades (sustituyen a las variables de instancia)

```
class Punto constructor (a: Double, b: Double) {  
    private var x: Double  
    private var y: Double  
    init {  
        x = a  
        y = b  
    }  
    constructor(): this(0.0,0.0) {}  
}
```

Propiedades

Bloque de
inicialización del
constructor primario

Constructor
primario

Constructor
secundario

La clase Punto versión 1.

Métodos

```
package puntosv1;
import kotlin.math.pow
import kotlin.math.sqrt
class Punto constructor (a: Double, b: Double) {
    private var x: Double;
    private var y: Double;
    init {
        x = a
        y = b
    }
    constructor(): this(0.0,0.0) {}
    fun getX(): Double = x
    fun getY(): Double = y
    fun distancia(p: Punto): Double =
        sqrt((p.x - this.x).pow(2.0) + (p.y - this.y).pow(2.0))
    fun trasladar(a: Double, b: Double) {
        x += a
        y += b
    }
    override fun toString(): String {
        return "($x,$y)"
    }
}
```

Creación de objetos

- Para crear objetos se utiliza un constructor (no se usa **new**).

```
import puntosv1.Punto;
```

```
fun main(args:Array<String>) {  
    val p: Punto = Punto(3.0,5.0)  
    p.trasladar(2.0,1.0)  
    println(p)  
    println(p.getX())  
    val q: Punto = Punto()  
    println(q)  
}
```

Mas sobre propiedades.

- Una propiedad declarada como **val** puede ser accedida en el ámbito de su visibilidad.
- Una propiedad declarada como **var** puede ser accedida y modificada en el ámbito de su visibilidad.
- Pueden establecerse ámbitos separados para la lectura (get) y escritura (set) de la propiedad.

Mas sobre propiedades.

Punto versión 2

```
package puntosv2;
import kotlin.math.pow
import kotlin.math.sqrt
class Punto (a: Double, b: Double) {
    var x: Double
        private set
    var y: Double
        private set
    init {
        x = a
        y = b
    }
    constructor(): this(0.0,0.0) {}
    fun distancia(p: Punto): Double =
        sqrt((p.x - this.x).pow(2.0) + (p.y - this.y).pow(2.0))
    fun trasladar(a: Double, b: Double){
        x += a
        y += b
    }
    override fun toString(): String {
        return "($x,$y)"
    }
}
```

La palabra
constructor puede
ser eliminada en el
constructor primario

Punto Versión 2

```
import puntosv2.Punto;

fun main(args:Array<String>) {
    val p: Punto = Punto(3.0,5.0)
    p.trasladar(2.0,1.0)
    println(p)
    println(p.x)
    // p.x = 5.0 // Produce error de compilación
    val q = Punto()
    println(q)
}
```

Mas sobre propiedades.

- Las **propiedades** de una clase pueden ser declaradas como **argumentos del constructor primario** cualificándolas con **val** o con **var**.
- En este caso no se puede restringir la visibilidad de los set y get por separado.

Mas sobre propiedades.

Punto versión 3

```
package puntosv3;
import kotlin.math.pow
import kotlin.math.sqrt
class Punto (var x: Double, var y: Double) {
    constructor(): this(0.0,0.0) {}
    fun distancia(p: Punto): Double =
        sqrt((p.x - this.x).pow(2.0) +
              (p.y - this.y).pow(2.0));
    fun trasladar(a: Double, b: Double){
        x += a
        y += b
    }
    override fun toString(): String {
        return "($x,$y)"
    }
}
```


Versión 3

```
import puntosv3.Punto;

fun main(args:Array<String>) {
    val p: Punto = Punto(3.0,5.0)
    p.trasladar(2.0,1.0)
    println(p)
    println(p.x)
    p.x = 2.1 // Ahora si es posible
    val q: Punto = Punto()
    println(q)
}
```

Mas sobre clases.

data class

- Si anteponemos al nombre de la clase la palabra reservada **data** entonces:
 - Todos los argumentos del constructor primario deben ser propiedades.
 - La clase incluye automáticamente la sobrescritura de los métodos **equals** y **hashCode** compatible que utiliza en su definición las propiedades declaradas en el constructor primario.
 - La clase incluye automáticamente la sobrescritura del método **toString** que muestra las propiedades incluidas en el constructor primario.
- En cualquier caso, si es necesario estos métodos pueden redefinirse.

Data classes.

Punto versión 4

```
package puntosv4;
import kotlin.math.pow
import kotlin.math.sqrt
data class Punto (var x: Double, var y: Double) {
    constructor(): this(0.0,0.0) {}
    fun distancia(p: Punto): Double =
        sqrt((p.x - this.x).pow(2.0) +
            (p.y - this.y).pow(2.0));
    fun trasladar(a: Double, b: Double) {
        x += a
        y += b
    }
}
```

Punto Versión 4

```
import puntosv4.Punto;

fun main(args:Array<String>) {
    val p: Punto = Punto(3.0,5.0)
    p.trasladar(2.0,1.0)
    println(p)
    println(p.x)
    p.x = 2.1
    val q = Punto()
    println(p == q)    // Asi se usa el equals
}
```

Propiedades con get y set

- Una propiedad puede definir los métodos set y get

```
package puntosgys;
```

```
class Punto (a: Double, b: Double) {
```

```
    var x: Double = 0.0
        set(value:Double) {
            println("antiguo valor = $field")
            field = value
            println("nuevo valor = $field")
        }
        get() {
            println("Se consulta el valor = $field")
            return field
        }
}
```

```
var y :Double
```

```
init {
    x = a
    y = b
}
constructor(): this(0.0,0.0) {}

override fun toString():String {
    return "($x,$y)";
}
}
```

Si tiene set hay que inicializarla previamente

```
import puntosgys.Punto
```

```
fun main(args:Array<String>) {
    val p: Punto = Punto(3.0, 5.0)
    p.x = 4.0
    println(p.x)
}
```

Propiedades calculadas

- Una clase puede añadir propiedades calculadas.
- No tienen valor sino que se calculan en el momento de usarlas.

```
class Punto (var x: Double, var y: Double) {  
  
    constructor(): this(0.0,0.0) {}  
  
    fun distancia(p: Punto): Double =  
        sqrt((p.x - this.x).pow(2.0) + (p.y - this.y).pow(2.0));  
  
    fun trasladar(a: Double, b: Double){  
        x += a;  
        y += b;  
    }  
  
    val distanciaAOrigen  
        get() = this.distancia(Punto())  
  
    override fun toString():String {  
        return "($x,$y)";  
    }  
}  
  
import puntospc.Punto  
  
fun main(args:Array<String>) {  
    val p: Punto = Punto(3.0,5.0)  
    println(p.distanciaAOrigen)  
}
```

Mensajes en cadena

- Se usa with(<objeto>){...}

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

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

Enumerados

- Como en Java pero añadiendo **class**

```
enum class Direccion {  
    NORTE, SUR, ESTE, OESTE  
}
```

```
enum class ColorBola{  
    BLANCA, NEGRA  
}
```


Herencia

- Si se quiere heredar de una clase, la clase padre debe ser declarada como **open**. (No puede ser **data class**)
- Si declaramos la clase **Punto** como **open class Punto {...}**, entonces

```
import puntosv3.Punto
const val G : Double = 6.67e-11
class Particula(a:Double,
               b:Double,
               var masa:Double) : Punto(a,b) {
    constructor (m:Double ) : this(0.0, 0.0, m) {}
    fun atraccion(part: Particula): Double {
        val d : Double = this.distancia(part)
        return G * masa * part.masa / (d * d)
    }
}
```

Así se llama al constructor del padre

La clase partícula añade una propiedad **var masa**

Vinculación estática

- La **vinculación** de los métodos por defecto es **estática**.
- Si se quiere redefinir un método (**fun**) y que use vinculación dinámica, éste debe ser declarado como
 - **open** en la clase padre y
 - **override** en la clase hija.
- Así ocurre por ejemplo con la función **toString()**

Herencia

- La clase **Any** es parecida a **Object** de Java.
- Sólo incluye los métodos **open**:
open fun equals(Any): Boolean
open fun hashCode(): Int
open fun toString(): String
- Si una clase no hereda de otra, entonces hereda de **Any**.

Interfaces

- Son similares a las interfaces de java8.
- Pueden declarar métodos abstractos.
- Pueden tener métodos por defecto.

```
interface MyInterface {  
    fun bar()  
    fun foo() {  
        // optional body  
    }  
}
```

Data Clases

- Automáticamente crea:
 - **equals** y **hashCode**
 - Componentes 1..n
 - Deconstrucción
 - **toString()**
 - Copia
- No se puede heredar de una data class

Data Classes. Componentes

```
package puntosv4;
import kotlin.math.pow
import kotlin.math.sqrt
data class Punto (var x: Double, var y: Double) {
    constructor(): this(0.0,0.0) {}
    fun distancia(p: Punto): Double =
        sqrt((p.x - this.x).pow(2.0) +
            (p.y - this.y).pow(2.0));
    fun trasladar(a: Double, b: Double){
        x += a
        y += b
    }
}
```

Data Classes. Componentes

```
import puntosv4.Punto;
fun main(args:Array<String>) {
    val p: Punto = Punto(3.0,5.0)
    p.trasladar(2.0,1.0)
    val mx = p.component1()
    val my = p.component2()
    println("x = $mx, y = $my")
}
```

Son operadores component1,..., componentN

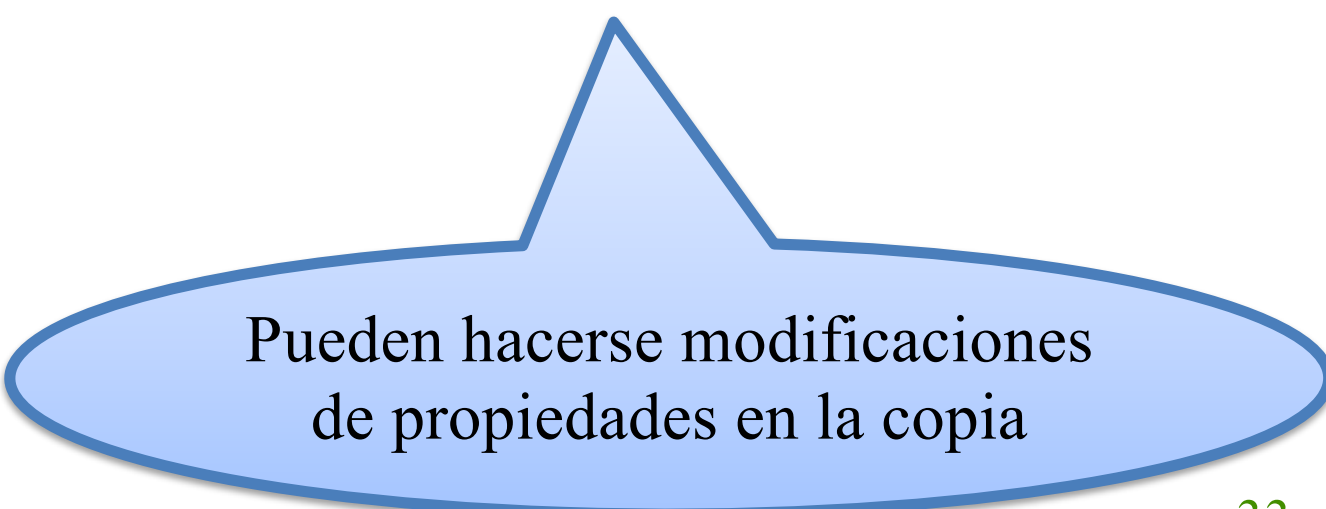
Data Clases. Deconstrucción

```
import puntosv4.Punto;
fun main(args:Array<String>) {
    val p: Punto = Punto(3.0,5.0)
    p.trasladar(2.0,1.0)
    val (mx,my) = p
    println("x = $mx, y = $my")
}
```


Data Classes. Copia

```
import puntosv4.Punto;

fun main(args:Array<String>) {
    val p: Punto = Punto(3.0,5.0)
    p.trasladar(2.0,1.0)
    val q : Punto = p.copy()
    val r : Punto = p.copy(y = 3.0)
    println(p)
    println(q)
    println(r)
}
```



Pueden hacerse modificaciones
de propiedades en la copia

Excepciones

- Se lanzan igual que java

```
if (imagenEstereograma == null) {  
    throw NoSuchElementException("No hay imagenEstereograma")  
}
```

- Kotlin no tiene excepciones comprobadas.
- Por interoperabilidad con java las que en java son de obligada captura se anuncian con una anotación

```
@Throws(IOException::class)  
fun leeDatosLocal(ficheroCSV: String) {  
    ...  
}
```

Excepciones. Bloque vigilado

- Se vigila igual que en java

```
try {  
    musculacion.leeDatosUrl(urlDatosMalaga)  
} catch (e: IOException) {  
    println("Error I/O: " + e.message)  
}
```

- Pero **try** puede ser una expresión que devuelve un resultado.

Excepciones. Closeables

- Es parecido al try con closeables

```
url.openStream().use { ins ->
    InputStreamReader(ins).use { isr ->
        BufferedReader(isr).use { bin ->
            CSVReader(bin).use { reader ->
                leeDatos(reader)
            }
        }
    }
}
```

Operadores is y as

- is: es como instanceof de java pero con alguna ventaja: **Si se comprueba no hace falta cast.**

```
if (other is Persona) {  
    res = nombre.toLowerCase() == other.nombre.toLowerCase()  
        && edad == other.edad  
}
```

- as: es como el cast de java

Tipos nulables

- En Kotlin ningún tipo puede recibir como valor **null**.
- Si se quiere que un valor pueda ser **null** su tipo debe indicarlo explícitamente:

```
fun main(args:Array<String>) {  
    var p: Punto? = null  
    p = Punto(2.0,1.0)  
    val mx = p.component1()  
    val my = p.component2()  
    println("x = $mx, y = $my")  
}
```

Operadores ?. ?: !!

- Se utilizan con tipos nullable: **Int?**, **String?**, etc
 - ?. Ejecuta el mensaje si la referencia no es nula. En otro caso devuelve **null**
`ref?.getX()`
 - ?: Operador elvis. Devuelve el contenido de la referencia si no es **null**. Si es **null** proporciona un valor alternativo
`val ref : String? = ...`
`val x: String = ref ?: "era null"`
 - !! Ejecuta el mensaje aunque la referencia sea nula. Lanza un **NullPointerException** si lo es.

Operadores

- Operadores unarios

| Expression | Translated to |
|------------|----------------|
| +a | a.unaryPlus() |
| -a | a.unaryMinus() |
| !a | a.not() |

- Operadores de incremento y decremento

| Expression | Translated to |
|------------|---------------|
| a++ | a.inc() |
| a-- | a.dec() |

Operadores

- Operadores aritméticos

| Expression | Translated to |
|------------|--|
| $a + b$ | <code>a.plus(b)</code> |
| $a - b$ | <code>a.minus(b)</code> |
| $a * b$ | <code>a.times(b)</code> |
| a / b | <code>a.div(b)</code> |
| $a \% b$ | <code>a.rem(b)</code> , <code>a.mod(b)</code> (deprecated) |
| $a..b$ | <code>a.rangeTo(b)</code> |

Operadores

- Operator in

| Expression | Translated to |
|------------|----------------|
| a in b | b.contains(a) |
| a !in b | !b.contains(a) |

Operadores

- Operadores de acceso

| Expression | Translated to |
|----------------------------|-------------------------------|
| $a[i]$ | $a.get(i)$ |
| $a[i, j]$ | $a.get(i, j)$ |
| $a[i_1, \dots, i_n]$ | $a.get(i_1, \dots, i_n)$ |
| $a[i] = b$ | $a.set(i, b)$ |
| $a[i, j] = b$ | $a.set(i, j, b)$ |
| $a[i_1, \dots, i_n] = b$ | $a.set(i_1, \dots, i_n, b)$ |

Operadores

- Operador de invocación

| Expression | Translated to |
|------------------|-------------------------|
| a() | a.invoke() |
| a(i) | a.invoke(i) |
| a(i, j) | a.invoke(i, j) |
| a(i_1, ..., i_n) | a.invoke(i_1, ..., i_n) |

Operadores

- Operador de asignación aumentada

| Expression | Translated to |
|---------------------|--|
| <code>a += b</code> | <code>a.plusAssign(b)</code> |
| <code>a -= b</code> | <code>a.minusAssign(b)</code> |
| <code>a *= b</code> | <code>a.timesAssign(b)</code> |
| <code>a /= b</code> | <code>a.divAssign(b)</code> |
| <code>a %= b</code> | <code>a.remAssign(b)</code> , <code>a.modAssign(b)</code> (deprecated) |

Operadores

- Operadores de igualdad y desigualdad

| Expression | Translated to |
|------------|-----------------------------------|
| $a == b$ | $a?.equals(b) ? : (b == null)$ |
| $a != b$ | $!(a?.equals(b) ? : (b == null))$ |

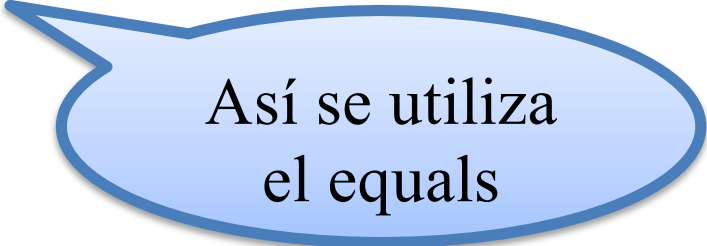
Operadores

- Operadores de comparación

| Expression | Translated to |
|------------|------------------------------------|
| $a > b$ | <code>a.compareTo(b) > 0</code> |
| $a < b$ | <code>a.compareTo(b) < 0</code> |
| $a \geq b$ | <code>a.compareTo(b) \geq 0</code> |
| $a \leq b$ | <code>a.compareTo(b) \leq 0</code> |

Operadores equals y hashCode

```
package persona
class Persona (val nombre:String, val edad:Double) {
    override operator fun equals(obj : Any?) : Boolean {
        var res = false
        if (obj is Persona) {
            res = nombre.toLowerCase() == obj.nombre.toLowerCase()
                && edad == obj.edad
        }
        return res
    }
    override fun hashCode(): Int =
        nombre.toLowerCase().hashCode() + edad.hashCode()
    override fun toString(): String {
        return "Persona($nombre,$edad)"
    }
}
```



Así se utiliza
el equals

Equals y hashCode

```
import persona.Persona
fun main(args:Array<String>) {
    val p1 = Persona("juan garcia", 23.1)
    val p2 = Persona("pedro perez", 20.0)
    val p3 = Persona("JUAN garcia", 23.1)
    println(p1)
    println(p2)
    println(p3)
    println(p1 == p2)
    println(p1 == p3)
    println(p1.hashCode())
    println(p2.hashCode())
    println(p3.hashCode())
}
```

La identidad es ===

Classes ordenables

```
class Persona (val nombre:String, val edad:Double) : Comparable<Persona> {  
    override fun equals(other: Any?) : Boolean {  
        var res = false  
        if (other is Persona) {  
            res = nombre.toLowerCase() == other.nombre.toLowerCase()  
                && edad == other.edad  
        }  
        return res  
    }  
    override fun hashCode(): Int =  
        nombre.toLowerCase().hashCode()+edad.hashCode()  
    override fun toString(): String {  
        return "Persona($nombre,$edad)"  
    }  
    operator fun compareTo(other: Persona): Int {  
        var res = nombre.toLowerCase().compareTo(other.nombre.toLowerCase())  
        if (res == 0) {  
            res = edad.compareTo(p.edad)  
        }  
        return res  
    }  
}
```

Clases ordenables

```
import persona.Persona
fun main(args: Array<String>) {
    val p1 = Persona("juan garcia", 23.1)
    val p2 = Persona("pedro perez", 20.0)
    val p3 = Persona("JUAN garcia", 23.1)
    println(p1)
    println(p2)
    println(p3)
    println(p1 > p2)
    println(p1 >= p3)
}
```

Mezcla de Java y Kotlin

- Una clase de Java puede heredar de otra clase de Kotlin o usarla en composición.
- Persona debe ser **open**

```
package persona;
```

```
public class Empleado extends Persona {  
    private double sueldo;  
    public Empleado(String nombre, double edad, double sueldo) {  
        super(nombre, edad);  
        this.sueldo = sueldo;  
    }  
    public double getSueldo() {  
        return sueldo;  
    }  
    @Override  
    public String toString() {  
        return "Empleado("+getNombre()+", "  
                +getEdad()+", "+getSueldo()+")";  
    }  
}
```

Mezcla de Java y Kotlin

- La usa como una clase cualquiera.

```
import persona.Empleado
fun main(args:Array<String>) {
    val empleado = Empleado("juan perez", 20.0, 2500.0)
    println(empleado)
    println(empleado.suelo)
}
```

- La variable de instancia **suelo** convierte en la propiedad **suelo** y **getSuelo()** de Java es la posibilidad de lectura en Kotlin.

Mezcla de Java y Kotlin

- Una clase de Kotlin puede heredar de otra de Java o usarla en composición.

```
package persona
class Jefe(nombre: String,
          edad: Double,
          sueldo: Double,
          val sobresueldo: Double)
          : Empleado(nombre, edad, sueldo) {
override fun toString(): String {
    return "Jefe($nombre, $edad, $sueldo, $sobresueldo)"
}
}
```

Mezcla de Java y Kotlin

- La usa como una clase cualquiera

```
import persona.Jefe;
public class MainJefe {
    public static void main(String [] args) {
        Jefe jefe = new Jefe("Pedro Marmol", 23.0,
                               2500.0, 1200.0);

        System.out.println(jefe);
        System.out.println(jefe.getNombre());
        System.out.println(jefe.getSobresueldo());
    }
}
```

- Las propiedades de lectura de Kotlin se convierte en variables de instancia de Java con getters.
- Las de escritura con getters y setters

Genericidad

- Es igual que en Java
- Los arrays también son genéricos con la notación normal.
 - **List**<Persona>
 - **Array**<String>

Varianza y Contravarianza

- Kotlin
- in T
- out T
- *

Java

? super T

? extends T

?

Varianza y Contravarianza

```
class Programa {  
    static public <T> void  
        copia (List<? extends T> orig,  
                Lis<? super T> dest) {  
        dest.addAll(orig);  
    }  
}
```

- Copiar una lista de jefes en una lista de personas
 - T Empleado
 - ? extends T Jefe
 - ? super T Persona

Varianza y Contravarianza

```
package copy
import persona.Jefe
import persona.Persona
fun main(args: Array<String>) {
    val lOrigen = listOf(Jefe("juan", 23.0, 2500.0, 1200.0),
                        Jefe("pedro", 25.0, 2300.0, 1400.0))
    val lDestino = mutableListof<Persona>()
    copia(lOrigen, lDestino)
    println(lDestino)
}
fun <T>copia(origen: List<out T>, destino: MutableList<in T>) {
    destino.addAll(0, origen)
}
```

- Copiar una lista de jefes en una lista de personas
 - T Empleado
 - out T Jefe
 - in T Persona

Métodos extensión

- Extienden los métodos o propiedades de una clase sin tener que redefinirlas.

```
package extension
fun comienzaConMayuscula1(s:String) =
    s[0].isUpperCase()

fun String.comienzaConMayuscula2() =
    this.get(0).isUpperCase()

val String.comienzaConMayuscula
get() = this.get(0).isUpperCase()
```

Uso de métodos extensión

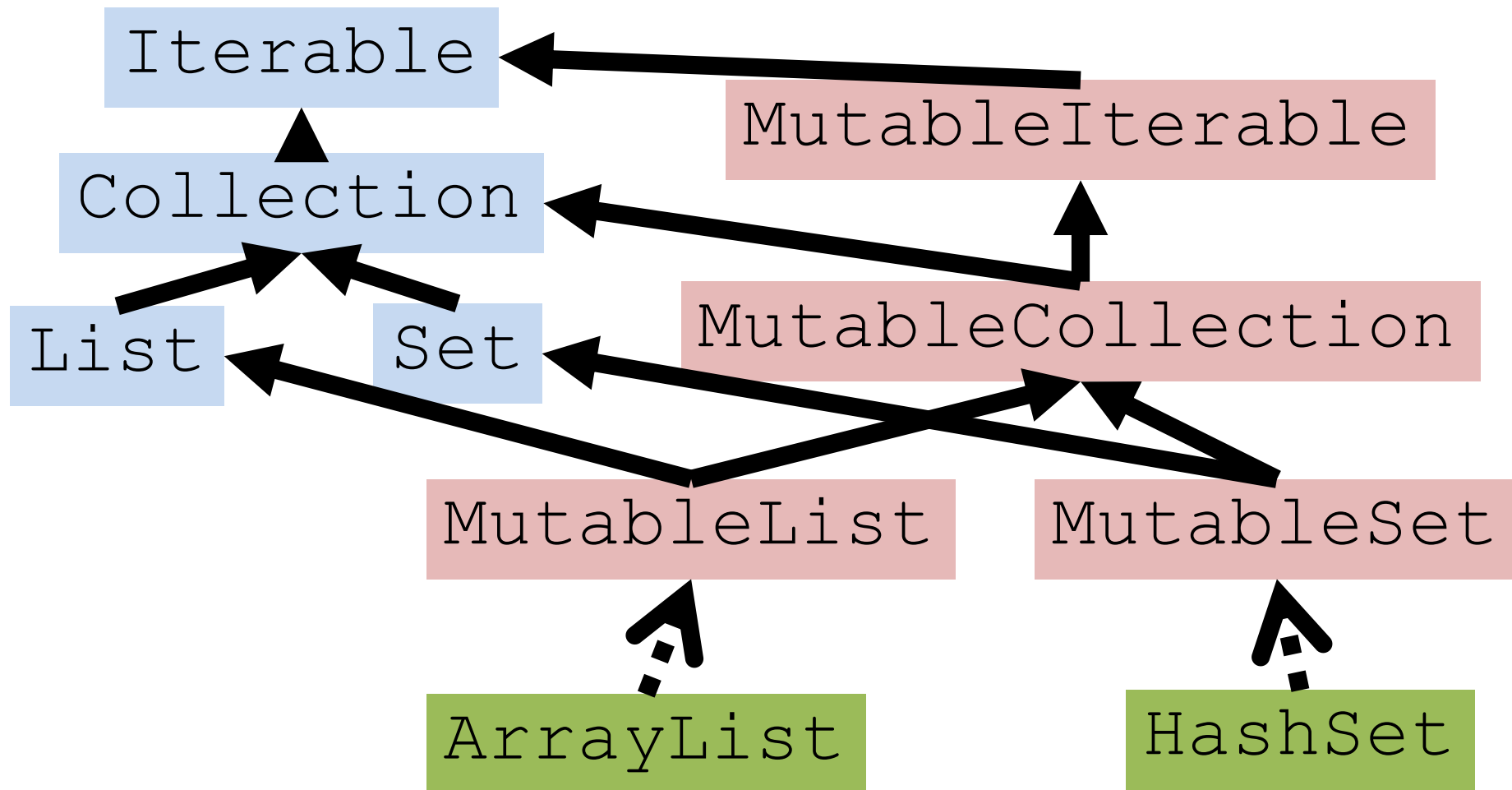
```
import extension.comienzaConMayuscula
import extension.comienzaConMayuscula1
import extension.comienzaConMayuscula2
fun main(args:Array<String>) {
    val s1 = "Hola a todos"

    println(comienzaConMayuscula1(s1))

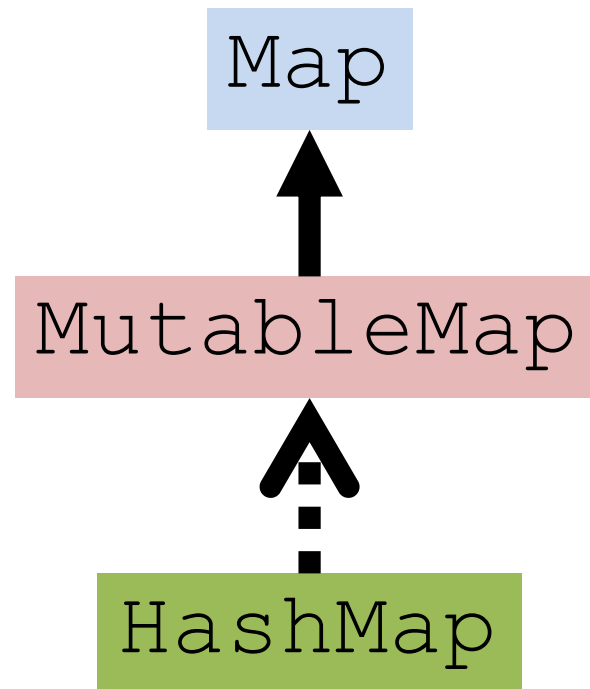
    println(s1.comienzaConMayuscula2())

    println(s1.comienzaConMayuscula)
}
```

Colecciones



Colecciones



Creación de colecciones

`listOf<T> (...)`

`setOf<T> (...)`

`mapOf<T> (...)`

`sortedSetOf (...)`

`sortedMapOf (...)`

`mutableListOf<T> (...)`

`mutableSetOf<T> (...)`

`mutableMapOf<T> (...)`

`kotlin-stdlib/kotlin.collections`

Creación de colecciones

```
fun main(args: Array<String>) {  
    val li = listOf("hola", "a", "todos", "que", "tal")  
    val si = setOf("hola", "a", "todos", "que", "tal", "a", "tal")  
    val mi = mapOf(1 to "hola", 4 to "todos", 3 to "tal")  
    val lm = mutableListOf("hola", "a", "todos", "que", "tal")  
    val sm = mutableSetOf("hola", "a", "todos", "que", "tal", "a", "tal")  
    val mm = mutableMapOf(1 to "hola", 4 to "todos", 3 to "tal")  
    println(li)  
    println(si)  
    println(mi)  
    println(lm)  
    println(sm)  
    println(mm)  
    // li[2] = "imposible"  
    lm[2] = "posible"  
    // mi["casa"] = 3  
    mm["casa"] = 4  
}
```

Declaraciones singleton

- Cuando de una clase solo se pretende crear un objeto puede definirse con **object**

```
package text
object Test {
    fun miFuncion() = "se devuelve 1"
}
```

- Y se usa

```
import text.Test
fun main(args: Array<String>) {
    println(Test.miFuncion())
}
```

Companion object

- Una clase no puede tener propiedades ni funciones estáticas. Si se quiere definir algo parecido hay dos opciones:
 - Usar funciones y propiedades globales
 - Usar un objeto **companion**

Creación de companion

```
package coches
```

```
class Coche(val nombre: String, val precio:Double) {  
    fun precioTotal() = precio*(1 + Coche.iva)  
  
    companion object {  
        var iva = 0.18  
    }  
    override fun toString():String {  
        return "Coche($nombre,${precioTotal()})"  
    }  
}
```

Uso de companion

```
import coches.Coche
fun main(args: Array<String>) {
    val cs = arrayOf(
        Coche("Seat Panda", 15000.0),
        Coche("Ferrari T-R", 65000.0),
        Coche("Seat Toledo", 21000.0),
        Coche("Jaguar XK", 41000.0),
        Coche("Porsche GT3", 44000.0))
    for(c:Coche in cs) {
        println(c)
    }
    println("Iva cambia a 0.21")
    Coche.iva = 0.21
    for(c:Coche in cs) {
        println(c)
    }
}
```

Expresiones lambda

- Expresiones que definen funciones

```
val f = {x: Double -> x + 3.0}
```

```
val g: (Double) -> Double = {x -> x + 3.0}
```

y se usan

```
fun main(args : Array<String>) {  
    println(f(4.0))  
    println(g(4.0))  
}
```

- El tipo de *f* es (Double) -> Double

Funciones de orden superior

- Funciones que reciben como parámetro funciones lambdas o devuelve una función lambda

```
fun os1(valor: Double, func: (Double) -> Double): Double {  
    return func (valor)  
}
```

```
fun os2(valor: String, func: (String, Int) -> Boolean): Boolean {  
    return func (valor,5)  
}
```

```
fun os3(valor: Int) : (String) -> Boolean {  
    return {s -> s.length > valor}  
}
```

Funciones de orden superior.

Uso

- Funciones que reciben como parámetro funciones lambdas o devuelve una función lambda

```
fun main(args : Array<String>) {  
    println(f(4.0))  
    println(os1(4.0, f))  
    println(os1(5.0, {x -> 2 * x}))  
    println(os1(5.0) {x -> 2 * x})  
    println(os2("Hola amigo", {s, n -> s.length > n}))  
    println(os2("Hola amigo") {s, n -> s.length > n})  
    println(os3(4)("Hola"))  
}
```


Funciones de orden superior.

Uso

- Si la lambda es de un solo parámetro puede utilizarse **it** como tal y no hay que declararlo
- Si la lambda es el último argumento puede extraerse de la lista de argumentos y colocarlo detrás

```
fun main(args : Array<String>) {  
    println( os1(5.0, {2 * it}) )  
    println( os1(5.0) {2 * it} )  
}
```

Funciones de orden superior y colecciones

- Existen gran cantidad de funciones para manipular las colecciones con funciones de orden superior