



Resumen Kotlin vs Java

Kotlin vs Java

16 de diciembre de 2023

CIFP Carlos III - Cartagena

Santiago Francisco San Pablo Raposo

2º curso DAM

|  |  |  |
| --- | --- | --- |
| Concepto | Kotlin | PHP |
| Variables (y tipos que admiten) | **Creación e inicialización de variables:** | |
| // Escribe variables de cada tipo de Kotlin  val entero = 1 // Int  val decimal = 1.0 // Double  val caracter = 'a' // Char  val cadena = "Hola" // String  val booleano = true // Boolean  val flotante = 1.0f // Float  val largo = 1L // Long  val corto: Short = 1 // Short  val byte: Byte = 1 // Byte  println(entero)  // En Kotlin por defecto, una variable no es nullable (Null safety), a menos que la declaremos con ?  var nulo: String? = "Hola"  nulo = null  // Sin embargo, no es una práctica recomendada, aunque puede darse el caso de que necesites declarar una variable e inicializarla más tarde. Para ello, Kotlin provee otras formas de hacerlo.  /\*-- Declaración de variables con lateinit --\*/  // Lateinit permite declarar una variable sin inicializarla, por ejemplo para poder inicializarla desde un constructor en una clase  lateinit var variableLateinit: String  variableLateinit = "Hola"  println(variableLateinit)  // Lateinit no se puede utilizar con tipos primitivos para variables locales. Con String sí, al ser internamente un objeto u array de caracteres    // Declaración de variables con lazy, permite que la variable sea inicializada solo cuando se accede a ella. Útil para inicializaciones costosas en tiempo  val variableLazy: String by lazy {  println("Inicializando variableLazy")  "Hola"  } | $a = $b = 3.1415; // float. Se pueden definir y asignar el valor de dos variables al mismo tiempo    $exp = 7.3e-1; // float    $a = "\"ahora\" 'la variable' a es un string: esto es el tipado dinámico"; // string    $comillasSimples = 'probando \'las comillas simples\''; // string    $bo = true; // boolean    $edad = 20; // integer    $nulo = null; // null    $arr = array(); // array    $conexion = new mysqli('localhost', 'usuario', 'contraseña', 'base\_de\_datos'); // objeto |
| **Destruir variables** | |
| No se puede hacer como tal. Como mucho, se pone a null como ya hemos visto | unset($bo); // Destruye una variable |
| **Null safety** | |
| if (nulo != null) { // Null safety  // println(nulo!!) // !! es para decirle a Kotlin que estamos seguros de que no es null, pero a pesar de ello, vemos que no nos sirve  }  // En su lugar, usaremos la función let, que nos permite ejecutar un bloque de código y devolver un valor a partir de un objeto que no es null  nulo?.let {  println(it ?: "Valor por defecto") // Elvis operator: Si es null, devuelve el valor por defecto  } ?: run { // Bloque run (parecido a else, es opcional) Si es null, ejecuta el bloque de código  println("La variable es nula")  }  println(nulo?.let { it } ?: "La variable es nula") // Es equivalente a la anterior instrucción (sin el run, obviamente)  println("length del string nulo: ${nulo?.length}") // Null safety: Safe calls: Si es null, no se ejecuta el length. Las safe calls es la práctica recomendada en Kotlin para no estar haciendo comprobaciones con if | En lenguajes antiguos como PHP o Java, esto no existe |
| **Ver el tipo de una variable:** | |
| // Saber de qué tipo es una variable  println(entero::class.simpleName)  println("Clase del número: ${entero.javaClass}")  if (entero is Int) {  println("Es un Int")  } | isset($variable); //permite verificar si está definida una variable y está inicializada.  empty($variable); // se utiliza para saber si una variable se considera como vacía (por ejemplo cuando su valor es cero).  gettype($variable); //sirve para obtener el tipo de una variable |
| **Hacer casting:** | |
| // Conversión de tipos (casting)  val numero: Int = 1  val numeroLargo: Long = numero.toLong()  val numeroString: String = numero.toString()  val stringNumero: Int = numeroString.toInt()  val objeto: Any = "Hola"  val cadena2: String = objeto as String // Usar solo si estas seguro de que es de ese tipo  val cadena3: String? = objeto as? String // Usar si no estas seguro de que es de ese tipo  // Formateo de cadenas (como sprintf en PHP)  val pi = 3.14159  val formattedPi = String.format("El valor de pi es %.2f", pi)  println("Pi formateado = $formattedPi") | **Casting directo**:  /\*-- Prueba de casting --\*/  $foo = 10;  $bar = (boolean) $foo; // casting manual  $str = "$foo";  $prueba = "20";  $valorInt = intval($prueba);  **settype($var, $tipo);** //tipo puede ser: boolean, integer, float, double, string, array, object o null.  $foo = "5bar"; // cadena $bar = true;   // booleano  settype($foo, "integer"); // $foo es ahora 5   (entero) settype($bar, "string");  // $bar es ahora "1" (cadena)  **sprintf**: permite pasar cualquier tipo (que tenga especificador para printf) a string, formatearlo, y guardarlo en una variable.  $txt\_pi = sprintf("El número PI vale %+.2f", 3.1416); |
| **Visibilidad de las variables desde funciones (variables globales):** | |
| /\*-- Visibilidad de las variables desde funciones --\*/  val variableGlobal = 1  fun funcion() {  println("Variable global desde función: $variableGlobal")  }  funcion() | /\*-- Visibilidad --\*/  $visibilidad = true;  function testVisibilidad() {  global $visibilidad; // Traslada la variable global a una variable local a la funcion  echo "$visibilidad";  }  testVisibilidad(); |
| Constantes | /\*-- Constantes --\*/  val testConstante = 10;  println(testConstante)  // testConstante = 11 // Error, no se puede reasignar | const nombreConstante = "valor";  define("nombreConstante", "valor");  echo nombreConstante; |
| Operaciones con números | **Operaciones con Math** | |
| /\*-- Operaciones con numeros (requiere kotlin.math.\*) --\*/  println("Operaciones con números")  println(abs(-10))  // Ceil = Redondea al número entero más alto  println(ceil(10.01)) // Devuelve el mayor entero más próximo del número introducido, devolvería 11  println(ceil(10.0)) // Devolvería 10  // println(ceil(10)) // No es válido  println(ceil(-10.01)) // Devuelve el mayor entero más próximo del número introducido, devolvería -10  // Floor = Redondea al número entero más bajo  println(floor(5.05)) // Redondea hacia abajo siempre, devolvería 5  println(floor(-5.9)) // Redondea hacia abajo siempre, devolvería -6  println(max(1, 2))  println(maxOf(1, 2, 3)) // Para calcular de más de dos numeros  println(min(1, 2))  println(minOf(1, 2, 3)) // Para calcular de más de dos numeros  // Pow = Potencia. Round = Redondea. Sqrt = Raíz cuadrada  println(2.0.pow(2.0)) // Tienen que ser numeros decimales.  println((0..10).random())  println(round(3.1415)) // Redondea al número entero más cercano  println(sqrt(4.0)) // Raíz cuadrada de 4  println(truncate(5.9)) // Trunca el número decimal, devolvería 5.0  println(truncate(-5.9)) // Trunca el número decimal, devolvería -5.0 | /\*-- Funciones matemáticas --\*/  $numero = 3.14;  echo abs($numero) . "<br>";  echo floor($numero) . "<br>";  echo ceil($numero) . "<br>";  echo max($numero, 2, 1) . "<br>";  echo max($numero, 2, 1) . "<br>";  echo pow(5, 0) . "<br>";  echo rand() . "<br>"; // Random entre 0 y 1  echo random\_int(1, 10) . "<br>"; // Random entero entre 1 y 10  echo round($numero) . "<br>";  echo sqrt($numero) . "<br>"; |
| Operaciones con Strings | **Comillas simples vs comillas dobles:** | |
| /\*-- Comillas simples vs comillas dobles (diferencia) --\*/  println("Hola 'amigo'")  println("Hola \"amigo\"")  println('a') // Comillas simples denota Char, comillas dobles denota String |  |
| **Operaciones típicas con String:** | |
| /\*-- Operaciones con cadenas --\*/  val nombre = "Juan"  val apellido = "García"  val nombreCompleto = nombre + " " + apellido  val nombreCompleto2 = "$nombre ${apellido}"  println("Length nombreCompleto: ${nombreCompleto.length}")  println("Subcadena: ${nombreCompleto.substring(0, 4)}")  println("Comparación de cadenas: ${nombreCompleto == nombreCompleto2}") // No es con equals en Kotlin  if (nombreCompleto.contains("Juan")) {  println("Contiene Juan")  }  println("Empieza por Juan: ${nombreCompleto.startsWith("Juan")}")  println("Termina por García: ${nombreCompleto.endsWith("García")}")  println("Index de Juan: ${nombreCompleto.indexOf("Juan")}")  println("Dividir una cadena: ${nombreCompleto.split(" ")}")  println("Reemplazar una cadena: ${nombreCompleto.replace("Juan", "Pedro")}")  println("Convertir a mayúsculas: ${nombreCompleto.uppercase()}")  println("Convertir a minúsculas: ${nombreCompleto.lowercase()}")  println("Eliminar espacios en blanco: ${nombreCompleto.trim()}") | /\*-- Operaciones con strings --\*/  $cadena = "Prueba";  echo strlen($cadena) . "<br>"; // Longitud de una cadena  $datos = explode("/", $fecha); // idem a split en javascript  /\*-- Búsqueda de subcadenas --\*/  /\*-- Busca si una cadena está contenida en otra --\*/  $cadena = "Sin León no hubiera España";  $busqueda = "León";  if (strpos($cadena, $busqueda) !== false) { //El método strpos también sirve para obtener la posición de esa subcadena dentro de la cadena principal  echo "La cadena '$busqueda' fue encontrada en la cadena '$cadena'";  } else {  echo "La cadena '$busqueda' no fue encontrada en la cadena '$cadena'";  } |
| **Obtener un carácter por posición:** | |
| println("Acceso a caracteres individuales: ${nombreCompleto[0]}") |  |
| **Obtener código ASCII de un carácter por posición:** | |
| // Obtener código ASCII de un caracter por posición  println("Código ASCII de la primera letra de nombreCompleto: ${nombreCompleto[0].toInt()}") |  |
| **Obtener el carácter a partir de un código ASCII:** | |
| // Convertir código ASCII a caracter  println("Caracter de código ASCII 74: ${74.toChar()}") |  |
| **Parsear de int a string y viceversa** | |
| val numeroString: String = numero.toString()  val stringNumero: Int = numeroString.toInt() |  |
| **Concatenación de cadenas mediante StringBuilder** | |
| // Concatenación de cadenas por StringBuilder  val fullName = buildString {  append("Nombre completo: ")  append("Santiago")  append(" ")  append("San Pablo")  } |  |
| Expresiones regulares | /\*-- Expresiones regulares --\*/  val regex = Regex("[0-9]+")  val texto = "123"  if (regex.matches(texto)) {  println("Es un número")  }  // Otra manera  val regex2 = "\\d+".toRegex()  if (regex2.matches(texto)) {  println("Es un número")  }  // Otra manera  if (texto.matches("[0-9]+".toRegex())) {  println("Es un número")  } | $cadena = "Sin León no hubiera España";  $patron = "/León/";  if (preg\_match($patron, $cadena)) {  echo "La cadena '$patron' fue encontrada en la cadena '$cadena'";  } else {  echo "La cadena '$patron' no fue encontrada en la cadena '$cadena'";  } |
| Manejo de fechas | /\*-- Manejo de fechas --\*/  // Crear fecha  val momentoActual = System.currentTimeMillis()  val fechaActual = java.time.LocalDate.now()  val fecha2 = java.time.LocalDate.parse("2021-01-01")  val fecha3 = java.time.LocalDate.from(java.time.format.DateTimeFormatter.ofPattern("dd/MM/yyyy").parse("01/01/2021"))  // Formatear fecha  val fechaFormateada = java.text.SimpleDateFormat("dd/MM/yyyy").format(momentoActual)  val fechaFormateada2 = java.text.SimpleDateFormat("dd/MM/yyyy").format(fechaActual)  val fechaFormateada3 = fechaActual.format(java.time.format.DateTimeFormatter.ofPattern("dd/MM/yyyy"))  val fechaFormateada4 = java.time.format.DateTimeFormatter.ofPattern("dd/MM/yyyy").format(fechaActual)  println("Fecha formateada: $fechaFormateada")  println("Fecha formateada 2: $fechaFormateada2")  // Comparar fechas  val fecha5 = java.time.LocalDate.now();  if (fecha3.isBefore(fecha5)) {  println("Fecha3 es anterior a Fecha5")  }  if (fecha5.isAfter(fecha3)) {  println("Fecha5 es posterior a Fecha3")  }  // Operaciones con fechas  val fecha6 = java.time.LocalDate.now()  println("Fecha6: $fecha6")  val fecha7 = fecha6.plusDays(1)  println("Fecha7 (fecha6 + 1 dia): $fecha7")  println("Diferecia en días entre fecha6 y fecha7: ${java.time.temporal.ChronoUnit.DAYS.between(fecha6, fecha7)}")  println("Diferencia entre fecha6 y fecha7: ${fecha6.until(fecha7, java.time.temporal.ChronoUnit.DAYS)}")  println("Dia del mes: ${fecha6.dayOfMonth}")  println("Dia de la semana: ${fecha6.dayOfWeek}")  println("Month: ${fecha6.month}")  println("Monthvalue: ${fecha6.monthValue}")  println("Año: ${fecha6.year}")  println("Días que han pasado desde inicio de año: ${fecha6.dayOfYear}")  println("Es año bisiesto: ${fecha6.isLeapYear}")  println("Fecha6 en formato ISO: ${fecha6.format(java.time.format.DateTimeFormatter.ISO\_DATE)}")  println("Fecha6 con día del mes alterado: ${fecha6.withDayOfMonth(31)}") |  |
| Condicionales y operador ternario | // If  if (1 == 1) {  println("Es igual")  } else if (1 != 1) {  println("Es distinto")  } else {  println("Ninguna de las anteriores")  } |  |
| // When (como el switch de Java, pero mejor)  when (1) {  1 -> println("Es 1")  2 -> println("Es 2")  else -> println("Ninguna de las anteriores")  }  // When con expresión  val resultado = when (1) {  1 -> "Es 1"  2 -> "Es 2"  else -> "Ninguna de las anteriores"  }  // When con varios casos  when (1) {  1, 2 -> println("Es 1 o 2")  else -> println("Ninguna de las anteriores")  }  // When con rango  when (1) {  in 1..10 -> println("Está entre 1 y 10") // Operador in permite simplificar la condición  else -> println("No está entre 1 y 10")  }  // Ejemplo de uso de when más complejo  fun cases(obj: Any) {  when (obj) { // 1  1 -> println("One") // 2  "Hello" -> println("Greeting") // 3  is Long -> println("Long") // 4  !is String -> println("Not a string") // 5  else -> println("Unknown") // 6  }  } |  |
| // Operador ternario  val numeroMayor = if (1 > 2) 1 else 2 |  |
| Bucles (excepto for each) | // Bucles for  for (i in 0..10) {  println(i)  }    for (i in 0 until 10) { // Until hace que el 10 no lo incluya en el recorrido  println(i)  }    for (i in 10 downTo 0) {  println(i)  }    for (i in 0..10 step 2) {  println(i)  } | **Bucle while**:  $a = 1;  while ($a < 8)  $a += 3;  print $a; // el valor obtenido es 10  **Bucle do while**:  $a = 5;  do  $a -= 3;  while ($a > 10);  print $a; // el bucle se ejecuta una sola vez, con lo que el valor obtenido es  **Bucle for**:  for ($a = 5; $a<10; $a+=3) {  print $a; // Se muestran los valores 5 y 8  print "<br />";  } |
| // Bucle while  var i = 0  while (i < 10) {  println(i)  i++  } |
| // Bucle do while  var j = 0  do {  println(j)  j++  } while (j < 10) |
| Arrays y listas (y bucles for each) | **Creación e inicialización de Arrays** | |
| // Los arrays y listas en Kotlin no pueden tener elementos de diferente tipo, salvo que se especifique List<Any> o Array<Any>, y después habría que hacer safe castings.  val mixedList: List<Any> = listOf("Hello", 123, true, 45.6)  val mixedArray: Array<Any> = arrayOf("World", 456, false, 78.9)  /\*-- Arrays --\*/  val array = arrayOf(1, 2, 3) // Array<Int>  val array2 = intArrayOf(1, 2, 3) // IntArray  val array3 = Array(3) { i -> i \* 2 } // Array<Int>. Es una serie de 3 números, cada uno con el doble del índice  val array4 = Array(3) { 0 } // Array<Int>  val array5 = (1..10) // IntRange, crea un array con los números del 1 al 10 mediante un rango  // Listas  val lista = listOf(1, 2, 3) // List<Int>: Lista inmutable  val listaMutable = mutableListOf(1, 2, 3) // MutableList<Int>  val listaMutable2 = MutableList(26) { i -> 'A' + 1 } // MutableList<Int> con 26 elementos, cada uno con una letra del abecedario | //Los arrays en PHP pueden mezclar distintos tipos de valores (por ejemplo, strings, numbers, booleans, etc.)List |
| **Acceso a elementos de un array/lista** | |
| // Acceso a elementos de un array  val elemento = array[0]  array[0] = 1  // Acceso a elementos y alteración de datos de una lista  val elementoLista = lista[0]  // lista[0] = -2 // No se puede porque la lista es inmutable  listaMutable[0] = 1 // En una lista mutable sí se puede |  |
| **Imprimir array o lista por pantalla (sin tener que recorrerlo con un bucle)** | |
| // Imprimir un array sin recorrerlo  println("Array: ${array}") // Así no saldrán los elementos  println("Array: ${array.joinToString()}") // Así sí saldrán los elementos del array. Es el opuesto a splittear un string  // Imprimir una lista  println("Lista mutable: ${listaMutable.joinToString()}") |  |
| **Añadir elementos (arrays dinámicos)** | |
| // Añadir elementos de un array mutable  array += 4 // Equivalente a array = array + numeroquesea  array = array.plus(5)  // array[5] = 6 // Error, no se puede añadir un elemento a un array ya creado  // Añadir y eliminar elementos de una lista mutable  listaMutable.add(4) |  |
| **Eliminar elementos** | |
| // Eliminar elementos de una lista mutable  listaMutable.remove(4) | $miArray = array("Prueba", 0, "2", 3, "4", "cinco");  print\_r($miArray);  unset($miArray[3]);  echo "<br>";  print\_r($miArray);    $miArray = array\_values($miArray);  echo "<br>";  print\_r($miArray); |
| **Saber si un elemento pertenece al array o lista** | |
| // Operadores in  if (1 in lista) {  println("Está en la lista")  } |  |
| **Función para volver a tener los índices de un array consecutivos tras eliminar un valor** | |
| Esto no aplica en Kotlin, ya que en el único sitio donde se pueden eliminar valores es en las listas mutables, y nunca quedan índices con valores en blanco al utilizar el método remove(). |  |
| **Recorrer arrays con for each** | |
| // Bucle for each  for (elemento in lista) {  println(elemento)  }  // Bucle for each con índice  for ((indice, elemento) in lista.withIndex()) {  println("$indice: $elemento")  }  // Bucle for each desde la propia lista  lista.forEach {  println(it)  }  // Bucle for each desde la propia lista con índice  lista.forEachIndexed { index, element ->  println("$index: $element")  }  // Iteradores  val iterador = lista.iterator()  while (iterador.hasNext()) {  println(iterador.next())  }    // Definición de iterador como operador en una clase contenedora de una lista de objetos de otra clase  class Contenedor(val lista: List<Int>) {  operator fun iterator(): Iterator<Int> {  return lista.iterator()  }  }  val contenedor = Contenedor(lista)  for (elemento in contenedor) {  println(elemento)  } | // Recupera solo el valor  $modulos = array("PR" => "Programación", "BD" => "Bases de datos", ..., "DWES" => "Desarrollo web en entorno servidor");  foreach ($modulos as $modulo) {  print "Módulo: ".$modulo. "<br />"  }  // Recupera clave y valor  $modulos = arrays("PR" => "Programación", "BD" => "Bases de datos", ..., "DWES" => "Desarrollo web en entorno servidor");  foreach ($modulos as $codigo => $modulo) {  print "El código del módulo ".$modulo." es ".$codigo."<br />"  }  // Mediante punteros (la función each ya avanza el puntero una posición)  while ($modulo = each($modulos)) {  print "El código del módulo ".$modulo[1]." es ".$modulo[0]."<br />"  } |
| **Ordenar arrays o listas** | |
| // Si tienes una lista de objetos  val listaDePeliculas: List<Movie> = // Tu lista de películas  listaDePeliculas.sortBy { it.anioLanzamiento }    // Si quieres comparar por múltiples campos de los objetos, implementa la interfaz Comparable y su método compareTo. Luego llama a sort()  class Movie(val anioLanzamiento: Int, val titulo: String) : Comparable<Movie> {  override fun compareTo(other: Movie): Int {  return this.anioLanzamiento.compareTo(other.anioLanzamiento)  }  }  val listaDePeliculas: MutableList<Movie> = // Tu lista de películas  listaDePeliculas.sort()  // Si quieres proporcionar un comparador personalizado, utiliza sortWith  data class Person(val nombre: String, val edad: Int)    fun main() {  val listaDePersonas = mutableListOf(  Person("Ana", 30),  Person("Carlos", 25),  Person("Elena", 30),  Person("David", 22)  )    listaDePersonas.sortWith(compareBy({ it.edad }, { it.nombre }))    for (persona in listaDePersonas) {  println("${persona.nombre}, ${persona.edad} años")  }  } |  |
| **Extraer un subarray** | |
| // Extrae un subarray  val subarray = array.slice(1..2)  println("Subarray: ${subarray.joinToString()}") |  |
| Funciones | **Sintaxis de las funciones** | |
| // Función normal  fun printMessage(message: String): Unit { // 1 Unit = void en java  println(message)  } // Unit es el tipo de retorno por defecto si no se especifica nada  // Función con argumento con valor por defecto (opcional)  fun printMessageWithPrefix(message: String, prefix: String = "Info") { // 2  println("[$prefix] $message")  }    // Función en forma de “expresión”  fun multiply(x: Int, y: Int) = x \* y // 4    // Infix functions aplicadas sobre funciones que has creado tú:  class Person(val name: String) {  val likedPeople = mutableListOf<Person>()  infix fun likes(other: Person) { likedPeople.add(other) } // 6  }    // Infix functions sobre funciones que no has creado tú (funcionan como “extensiones”)  infix fun Int.times(str: String) = str.repeat(this) // 1  println(2 times "Bye ") // 2    // Funciones que actúan como operadores (ver listado de operadores de Kotlin que se pueden reimplementar)  operator fun Int.times(str: String) = str.repeat(this) // 1  println(2 \* "Bye ") // 2    operator fun String.get(range: IntRange) = substring(range) // 3  val str = "Always forgive your enemies; nothing annoys them so much."  println(str[0..14]) | /\* Acceso a variables globales \*/ |
| **Paso de parámetros por valor y por referencia** | |
| En Kotlin, todas las variables son objetos. Como tal, se pasan siempre por referencia a las funciones.  Sin embargo, los tipos primitivos (Int, Double, Float, String, etc.), son pasados siempre por valor.  Es decir, un objeto de una clase, o bien un array -> por referencia  Un objeto de un tipo primitivo -> por valor |  |
| **Paso de muchos argumentos iguales** | |
| fun printAll(vararg messages: String) { // 1  for (m in messages) println(m)  }  printAll("Hello", "Hallo", "Salut", "Hola", "你好") // 2 | public void prueba(String.. prueba) |
| **Llamadas a las funciones:** | |
| // Funciones con parámetros nombrados  fun suma3(a: Int, b: Int) = a + b  val resultadoSuma3 = suma3(b = 1, a = 1)    // Funciones con retorno nullable  fun suma5(a: Int, b: Int): Int? {  return a + b  }  val resultadoSuma5 = suma5(1, 1) |  |
| Lambdas | **Expresiones lambdas** | |
| /\*-- Lambdas --\*/  // Sintaxis general (sirve para lambdas con uno o más de un parámetro)  val sumaLambda = { a: Int, b: Int -> a + b } // Version simplificada (inferencia de tipos fuera de la lambda)  val ejecucionLambda1 = sumaLambda(1, 2) // Ejecución de la lambda  val declaracionEjecucionLambda = { a: Int, b: Int -> a + b }(1, 2) // Declaración y ejecución de la lambda en misma línea  val sumaLambda2: (Int, Int) -> Int = { a, b -> a + b } // Version semi-completa (inferencia de tipos dentro de la lambda)  val sumaLambda3: (Int, Int) -> Int = { a: Int, b: Int -> a + b } // Version completa  // Sintaxis para lambdas de un solo parámetro (se puede utilizar si se quiere)  val upperCaseLambda1: (String) -> String = { it.uppercase() } // Lambda simplificada con un solo parámetro. it es el nombre por defecto del parámetro  val upperCaseLambda2: (String) -> String = { s -> s.uppercase() } // Lambda semicompleta con un solo parámetro, dándole un nombre  val upperCaseLambda3: (String) -> String = { s: String -> s.uppercase() } // Lambda completa con un solo parámetro  val upperCaseLambda4: (String) -> String = String::uppercase // Lambda con referencia a método de clase  // Uso de lambdas para funciones del sistema  val myIntList = arrayListOf(0,1,2,3,4,5,6,7,8,9,10)  val myFilterIntList = myIntList.filter { myInt ->  println(myInt)  if (myInt == 1) {  return@filter false  }  myInt > 5 // Las lambdas siempre devuelven la última línea, si antes no se ha hecho un return  }  // Uso de lambdas para funciones propias, en este caso, para una función de orden superior (callback)  fun myAsyncFun(name: String, hello: (String) -> Unit) {  thread {  Thread.sleep(1000)  hello(name)  }  }  myAsyncFun("Santiago") { println("Hola $it") } // Podemos implementar la lambda fuera de la llamada | // Ejemplo de ordenamiento con función lambda  $myArray = [3, 1, 4, 1, 5, 9];  usort($myArray, function ($a, $b) {  return $a < $b;  });    // Ejemplo de ordenamiento con función previamente definida  function mySort($a, $b) {  return $a < $b;  }  usort($myArray, 'mySort');    $suma = fn($a, $b) => $a + $b;  echo $suma(3, 5); // Imprime 8 |
|  | |
|  |  |
| Objetos | **Creación/instanciación de objetos** | |
| fun main() {  // […]  // Formas de instanciación de objetos  p1 = Punto2D(1, 2)  p2 = Punto2D(1, 2)    val p3: Punto2D by lazy {  Punto2D(1, 2)  }    lateinit var p4: Punto2D  p4 = Punto2D(1, 2)    val p5 = object { // Con la palabra reservada object: Objecto anónimo Punto3D  val x: Int = 1  val y: Int = 2  val z: Int = 3    fun suma() = this.x + this.y + this.z  }  println("x: ${Punto3D.x}, y: ${Punto3D.y}, z: ${Punto3D.z}")  println("Suma de Punto3D: ${Punto3D.suma()}")  }    object Punto3D { // Objeto singleton Punto3D, declarado fuera del main  val x = 1  val y = 2  val z = 3  fun suma() = x + y + z  } |  |
| **Acceso a los elementos del objeto?:** | |
| Si son públicos, se pueden acceder directamente. Y si tienes dos variables que se llaman iguales, una fuera y otra dentro del objeto (como por ejemplo es nuestro caso en el ejemplo anterior del objeto p5, puedes usar la palabra reservada this. |  |
| **Recorrer atributos de objetos** | |
| Se puede lograr haciendo uso de la reflexión. En Kotlin no está soportada nativamente, hay que importar la biblioteca de reflexión kotlin-reflect. Ejemplo:  import kotlin.reflect.full.memberProperties    class MiClase(val atributo1: Int, val atributo2: String, val atributo3: Boolean)    fun main() {  val miObjeto = MiClase(1, "Hola", true)    val clase = miObjeto::class  for (prop in clase.memberProperties) {  println("${prop.name} = ${prop.get(miObjeto)}")  }  }  La reflexión puede ser costosa en términos de rendimiento y no se recomienda para operaciones críticas de rendimiento. Además, puede tener implicaciones de seguridad, ya que permite el acceso a campos y métodos que normalmente estarían ocultos. | Que yo sepa esto en PHP no se puede hacer. Solo se puede en JavaScript |
| **Objetos singleton\*** | |
| En Kotlin, es posible definir un singleton como ya hemos visto con el ejemplo de Punto3D. | En Java no se puede hacer que yo sepa. Y en PHP creo que tampoco. |
| Clases | **Definición de clase** | |
| // Clases normales  class Punto2D (val x: Int = 0, val y: Int = 0) {  init { // Runnable code for the primary constructor  // Código que se ejecuta en el constructor primario.  // El constructor primario se crea por defecto sin especificarlo con los atributos especificados en la cabecera de la clase  require(x >= 0) { "x must be greater than or equal to 0" }  require(y >= 0) { "y must be greater than or equal to 0" }  println("First initializer block that prints x=$x, y=$y")  }  // constructor() : this(0, 0) { // Constructor secundario  // Más código que quieras incluir en este segundo constructor  // println("Secondary constructor")  // }  constructor(x: String, y: String) : this(x.toInt(), y.toInt()) // Otro constructor secundario  constructor(punto: Punto2D) : this(punto.x, punto.y) // Constructor por copia    override fun equals(other: Any?): Boolean {  when (other) {  is Punto2D -> {  if (this === other) return true  if (x != other.x) return false  if (y != other.y) return false  return true  }  else -> return false  }  }    override fun hashCode(): Int {  var result = x  result = 31 \* result + y  return result  }    override fun toString(): String {  return "Punto2D(x=$x, y=$y)"  }    // Esta función permite copiar un objeto de la clase Punto2D y modificar sus atributos  fun copy(x: Int = this.x, y: Int = this.y) = Punto2D(x, y)  }    val p1: Punto2D  val p2: Punto2D    p1 = Punto2D(1, 2)  p2 = Punto2D(1, 2)    println("punto1 == punto2: ${p1 == p2}") // == llama a equals en Kotlin, a diferencia de Java  // Data classes  data class DataClase (var x: Int, var y: Int) {  constructor(punto: DataClase):this(punto.x, punto.y)  constructor(): this(0,0) // Sería equivalente a poner valores por defecto en el constructor primario    // En una clase de datos, el compilador genera automáticamente funciones como equals(), hashCode(), toString() y copy() basados en las propiedades declaradas en el constructor primario.  // Además, los atributos x e y son de solo lectura, ya que son constantes (val)  }  **Importante 1**: cómo se puede apreciar, los atributos se pueden crear dentro del propio constructor primario, o fuera, pero nunca desde un constructor secundario.  **Importante 2**: en el constructor primario, si los **atributos** no se definen como **val** o **var**, **no serán atributos accesibles desde dentro de la clase**, sino que solamente quedarán como parámetros del constructor primario, y solo serán accesibles desde el mismo, es decir, desde el init.  **Otra forma de definir los atributos** puede ser así (prescindiríamos de constructor primario, y crearíamos cuantos constructores secundarios necesitásemos, brindando así más control, aunque no es la práctica recomendada en Kotlin, ya que se parecería más a Java):  open class DefinicionAlternativaAtributosClases {  var nombre: String // También se podría inicializar directamente sin pasar por el constructor, poniendo var nombre: String = ""  var edad: Int // Lo mismo aplicaría para la edad  var tallaPie: Int = 2 // Mediante esta sintaxis, es posible controlar las operaciones get y set de los atributos  get() = field // field es una palabra reservada que se refiere al atributo en sí  set(value) {  field = value  }    constructor(nombre: String, edad: Int) {  this.nombre = nombre  this.edad = edad  }  Además, aquí hemos visto **otra forma de definir los atributos**, controlando el setter y el getter asociados a él. De esta forma, vemos que en Kotlin no hace falta implementar getters y setters, porque los propios atributos, en realidad no son atributos, son propiedades que ya tienen estas operaciones definidas por el propio lenguaje. **Importante**: se pueden implementar get y set cuando sea var. Si es val (constante), solo se puede implementar get. |  |
| **Visibilidad de los atributos/métodos** | |
| class Persona(private var nombre: String, private var edad: Int = 0) {  // Getter público para el nombre  fun getNombre(): String {  return nombre  }    // Setter público para el nombre  fun setNombre(nombre: String) {  this.nombre = nombre  }    // Getter público para la edad  fun getEdad(): Int {  return edad  }    // Setter público para la edad  fun setEdad(edad: Int) {  if (edad >= 0) {  this.edad = edad  }  }  }  **Existen más** [**modificadores de acceso**](https://kotlinlang.org/docs/java-to-kotlin-interop.html#visibility), además de private:   * **public**: por defecto. No hace falta ponerlo. * **internal**: el miembro es visible en todas partes dentro del mismo módulo. * **protected**: el miembro es visible dentro de la clase y sus subclases. |  |
| **Visibilidad de las clases** | |
| * **public**: modificador por defecto. * **internal**: la clase es visible en todas partes dentro del mismo módulo * **private**: la clase es visible dentro del archivo en el que se declara. |  |
| **Miembros de clase** | |
| En Kotlin, los miembros de clase se declaran mediante un companion object:  class ClaseConCompanionObject {  companion object {  val atributoEstatico = 1  fun metodoEstatico() {  println("Método estático: $atributoEstatico")  }  }    val atributoNoEstatico = 2  fun metodoNoEstatico() {  println("Método no estático: $atributoNoEstatico")  }  } | En Java, se haría así:  public class EjemploClase {  private static int contadorEstatico = 0;    private int contadorNoEstatico;    public EjemploClase() {  contadorEstatico++;  contadorNoEstatico = 0;  }    public static int obtenerContadorEstatico() {  return contadorEstatico;  }    public int obtenerContadorNoEstatico() {  return contadorNoEstatico;  } |
| **Herencia** | |
| **Definir una clase heredable**:  Las clases por defecto son no heredables. Si quieres que una clase sea heredable, debes anteponer la palabra “open” a la palabra class en su definición.  Para que una clase sea heredable debe ser:   * “open class”: cualquier clase puede heredar de la clase. * “sealed class”: solo las clases que estén en el mismo paquete que la clase pueden heredar de esta.   Hay una opción intermedia: **sealed** classes. que solo puedan heredar de una clase aquellas clases que pertenezcan al mismo paquete que esta.  **Crear una clase que herede de otra**:  Es tan simple como poner a continuación del nombre de tu clase dos puntos, y el nombre de la clase de la que quieres heredar, y entre paréntesis, pasar los parámetros del constructor de la superclase. |  |
| **Herencia múltiple** | |
| En Kotlin, para conseguir el efecto de la herencia múltiple, hacemos igual que en Java: utilizamos clases e interfaces. Ejemplo:  // Clase base  open class ClaseEInterfaz(val nombre: String) {  fun comer() {  println("$nombre está comiendo")  }  }    // Interfaz  interface Nadador {  fun nadar()  }    // Clase que hereda de una clase base y también implementa una interfaz  class Pez(nombre: String) : ClaseEInterfaz(nombre), Nadador {  override fun nadar() {  println("$nombre está nadando")  }  }    fun main() {  val pez = Pez("Nemo")  pez.comer()  pez.nadar()  }    Igual que en Java, aunque las interfaces pueden contener constantes, no pueden tener campos de otro tipo. [En Kotlin, estos campos se declaran dentro de un companion object](https://ricardev.gitbook.io/apuntes-kotlin/programacion-orientada-a-objetos/herencia-multiple) |  |
| **Polimorfismo** | |
| Puedes **implementar métodos** abstractos de la clase que herede tu clase, **o** incluso **reimplementar** aquellos métodos que la clase padre de la que hereda tu clase tenga definidos como no finales.  //Superclase  **open** class Figura {  **open** fun area(): Double {  return 0.0  }  }  //Subclase  class Circulo(val radio: Double) : Figura() {  **override** fun area(): Double {  return Math.PI \* radio \* radio  **Otro ejemplo**:  sealed class Persona(internal var nombre: String, private var edad: Int = 0) {  open val prueba = "Hola" // Los atributos que uno no quiera que vayan en el constructor primario, se pueden declarar de esta forma  […]  }    class Hombre(nombre: String, edad: Int = 0) : Persona(nombre, edad) { // Se llama al constructor de la clase padre  override var prueba = "Adiós" // Sobreescribimos la propiedad 'prueba' de la clase padre. Se puede sobreescribir de val a var, pero no de var a val  // No se puede acceder a 'edad' porque es privado en 'Persona'  override fun prueba() {  super.prueba()  println(nombre)  }  }    class Elle : Persona {  constructor(nombre: String, edad: Int) : super(nombre, edad) // Otra forma de llamar al constructor de la clase padre  } |  |
| **El modificador abstract** | |
| Una clase abstracta impide que se creen instancias de la clase. Se podrán crear instancias de subclases que hereden de ella.  Los **miembros** (métodos o **atributos**) de una clase abstracta también pueden ser marcados como abstract, lo que exigirá a las subclases que implementen dichos miembros.  abstract class ClaseAbstractaKotlin {  abstract val propiedadAbstracta: Int  abstract var propiedadAbstracta2: Int    abstract fun metodoAbstracto()  }  class ClaseConcretaKotlin : ClaseAbstractaKotlin() {  override val propiedadAbstracta: Int = 1  override var propiedadAbstracta2: Int = 2    override fun metodoAbstracto() {  println("Método abstracto")  }  } | En java, solo se pueden marcar como abstractos los métodos, pero no los atributos.  public abstract class ClaseAbstractaJava {  public abstract void prueba();    public abstract String atributo; // A diferencia de Kotlin, Java no permite declarar atributos abstractos  } |
| **Sobrecarga de métodos** | |
|  |  |
| **Extensiones a métodos y atributos (alternativa a la herencia)** | |
| // [Fichero PersonaExtensions.kt]  /\*-- Extension functions (se pueden crear aunque la clase sea final, es decir, no open) --\*/  // Es posible crearlas dentro de una clase, o incluso dentro de una función, pero no es recomendable  private fun Persona.esHombre() = this is Hombre  fun Persona.esMujer() = this is Mujer  fun Persona?.numeroGenero() : Int? = if (this == null) null else if (this.esHombre()) 1 else 2  // Las extensiones de propiedades no se pueden hacer desde un miembro local. Es decir, desde el main no se podría.  // En cambio, desde fuera si se puede.    /\*-- Extensions properties (se han de escribir fuera del main y de cualquier bloque) --\*/  // Mapa que almacena los DNI de las personas por su nombre  val dniPorNombre = mutableMapOf<String, String>()    // Extension property para la clase Persona  var Persona.dni: String  get() = dniPorNombre[this.nombre] ?: "No se encontró DNI" // Operador elvis para devolver un valor por defecto si no se encuentra el DNI  set(value) {  dniPorNombre[this.nombre] = value  }  // Recuerda que las extension properties no añaden campos a la clase, solo añaden métodos get y set  // Por tanto, para tener un campo de respaldo, se necesita crear una variable externa  // En este caso, se ha optado por un mapa que almacena los DNI por nombre    // Extension property que depende de una extension function que es nullable  val Persona?.genero: Int  get() = numeroGenero() ?: 0  // [Fichero Main.kt]  fun main() {  […]  /\*-- Extensions (ver fichero PersonaExtensions.kt) --\*/  val perNullable3: Persona? = null  val per4: Persona = Mujer("Marta", 29)  // Function extensions  println("per3 es mujer?: ${perNullable3.numeroGenero()}") // numeroGenero() devolverá null, ya que per3 es null, y el método es nullable  // Property extensions  per1.dni = "23829353L" // dni es una propiedad de extensión  println("DNI de per1: ${per1.dni}")  println("DNI de per2: ${per2.dni}") // Como a per2 no le hemos asignado dni, devolverá "No se encontró DNI"  println("Género de per4: ${per4.genero}") // genero es una propiedad de extensión que depende de una función de extensión  println("Número de género de per3: ${perNullable3.genero}") // genero es una propiedad de extensión que depende de una función de extensión. Devolverá 0 si es null, ya que tiene un valor por defecto  // Visibilidad de las extensiones  // println("per1 es hombre? ${per1.esHombre()}") // No podemos porque esHombre() es private, solo se puede usar en el mismo archivo en el que fue definido  println("per1 es mujer?: ${per1.esMujer()}") // esMujer() es pública, por lo que se puede usar en cualquier parte del código  } | Que yo sepa, PHP no tiene esto |
| **Delegación (alternativa a la herencia)** | |
| [Delegation | Kotlin Documentation (kotlinlang.org)](https://kotlinlang.org/docs/delegation.html) |  |
| Interfaces | **Sintaxis** | |
| En Kotlin, las interfaces pueden contener funciones (igual que en Java).  Sin embargo, a diferencia de Java, **en las interfaces Kotlin, pueden haber funciones ya implementadas**. Además, aunque no pueden tener estado (es decir, no tienen atributos a los que poder acceder desde una instancia de objeto del tipo de la interfaz), si que pueden tener propiedades a nivel de clase.  interface Nadador {  // Las interfaces no pueden tener constructores  // Pero sí pueden tener propiedades abstractas  val nombre: String  var estado: String  // var edad: Int = 0 // Esto no es válido en una interfaz, ya que una interfaz no puede tener estado  // var edad: Int // Esto tampoco, ya que al poner field, estamos diciendo que el atributo es mutable, y por tanto, estaríamos diciendo que tiene un estado  // get() = field  // set(value) {field = value}  var edad: Int  get() = 0  set(value) {println("No se puede modificar la edad")}    fun nadar()  fun ahogarse() {  println("¡Oh no! se está ahogando")  }  } | En Java las interfaces permiten definir comportamientos mediante las cabeceras de métodos, pero no pueden tener cuerpo. Es decir, no pueden implementarse desde la propia interfaz en Java. |
| Enumerados | **Enumerados sin valores** | |
| /\*-- Enumerados --\*/  // Definición de enumerados simples fuera de cualquier método  enum class EnumeradoEstadoKotlin { // Enumerado simple  IDLE, RUNNING, FINISHED  }    // Uso de enumerados simples  val state = EnumeradoEstadoKotlin.RUNNING  val message = when (state) {  EnumeradoEstadoKotlin.IDLE -> "It's idle"  EnumeradoEstadoKotlin.RUNNING -> "It's running"  EnumeradoEstadoKotlin.FINISHED -> "It's finished"  }  println(message) | Esto es Java, no PHP:  public enum Demarcacion {  PORTERO,  DEFENSA,  CENTROCAMPISTA,  DELANTERO  }  // Ejemplo de uso:  Demarcacion delantero = Demarcacion.DELANTERO;  System.out.println("Nombre de la demarcación: " + delantero.name());  System.out.println("Posición en el enum: " + delantero.ordinal()); |
| **Enumerados con valores y métodos** | |
| // Definición de enumerados complejos fuera de cualquier método  enum class EnumeradoEstadoKotlin2(val rgb: Int) { // Enumerado con constructor  RED(0xFF0000),  GREEN(0x00FF00),  BLUE(0x0000FF);    fun containsRed() = this.rgb and 0xFF0000 != 0  }    // Enumerado con constructor y funciones  val color = EnumeradoEstadoKotlin2.RED  println("Color rojo contiene rojo: ${color.containsRed()}") | Esto es Java, no PHP:  public enum Demarcacion {  PORTERO("Guantes"),  DEFENSA("Marcaje"),  CENTROCAMPISTA("Pase"),  DELANTERO("Goles");    private final String habilidad;    private Demarcacion(String habilidad) {  this.habilidad = habilidad;  }    public String getHabilidad() {  return habilidad;  }  }  Demarcacion delantero = Demarcacion.DELANTERO;  System.out.println("Habilidad del delantero: " + delantero.getHabilidad()); |
| Pares | **Por ejemplo, pares clave-valor** | |
| /\*-- Clase Pair --\*/  // Construcción de un Pair  val par = Pair(1, 2) // Constructor de Pair  val par2: Pair<String, Int> = Pair("Manzana", 5) // Pair con tipos explícitos (declaración completa)  val par3 = "Pera" to 3 // Pair con notación simplificada (to)  // Acceso a los datos del par  val fruta = par2.first  val cantidad = par2.second  // Destructuring de un Pair  val (primero, segundo) = par // Guarda los valores del par en las variables primero y segundo |  |
| Maps o diccionarios | **Asociación clave-valor**  No pueden existir dos claves iguales, pero sí dos valores iguales para claves diferentes.  /\*-- Mapas o diccionarios --\*/  // Sintaxis mapas inmutables  var myMap: Map<String, Int> = mapOf() // Mapa inmutableS  // Añadir elementos  myMap = mapOf("a" to 1, "b" to 2, "c" to 3) // Mapa inmutable con notación simplificada (to). Los objetos Pair serían equivalentes.  // Añadir un solo valor a un mapa inmutable  // myMap["Ana"] = 7 // No lo podemos hacer en un mapa inmutable  myMap = myMap.plus(Pair("Ana", 7)) // Añadir un solo valor a un mapa inmutable. Este método devuelve un nuevo mapa con el valor añadido, por lo que tiene un coste en memoria    // Sintaxis mapas mutables  Var myMutableMap1 = HashMap<String, Int>() // Mapa mutable, sintaxis clásica. Implementación específica de MutableMap  // Además de la implementación HashMap, también está la LinkedHashMap (subclase de HashMap que guarda el orden en que se guardan los elementos, útil para recorrerlos) y la SortedMap (es una subinterfaz de Map que garantiza que las entraedas están ordenadas por sus claves. Su implementación más común es TreeMap.  var myMutableMap2 = mutableMapOf<String, Int>() // Mapa mutable, sin implementación específica (por defecto es LinkedHashMap)  // Añadir varios elementos (utilizamos siempre la notación to, porque es más cómoda)  val mapa = mutableMapOf("a" to 1, "b" to 2, "c" to 3)  mapa.putAll(mapOf("Juan" to 10, "María" to 11)) // Añadir varios valores a un mapa mutable    // Añadir un solo valor a un mapa mutable  mapa["Ana"] = 7 // Añadir un solo valor a un mapa mutable  mapa.put("Pepe", 9) // Otra forma de añadir un solo valor a un mapa mutable  mapa["Pepe"] = 10 // En este caso, como la clave ya existe, modifica su valor en el mapa mutable. También haría lo mismo mapa.put("Pepe", 10)    // Acceso a elementos de un mapa  println("mapa[Ana]: ${mapa["Ana"]}") // Acceso a un valor de un mapa    // Eliminación de elementos de un mapa  mapa.remove("Ana") // Elimina la entrada con clave "Ana" |  |
| Type aliases | Se pueden crear alias de tipos, para identificarlos de manera más rápida, acortando (si fuera necesario) el nombre de los tipos.  typealias MyMapList = MutableMap<Int, ArrayList<String>>  typealias MyFun = (Int, String, MyMapList) -> Boolean  typealias MyNestedClass = ClasesAnidadasYInner.MyNestedClass  typealias MyInnerClass = ClasesAnidadasYInner.MyInnerClass    fun myFunction(value: Int, text: String, mapList: MyMapList): Boolean {  // Aquí va tu implementación  return true // Devuelve un valor booleano  }    fun main() {  […]  /\*-- Type alias --\*/  // Type alias para tipos muy largos  val myMapList: MyMapList = mutableMapOf()  // Type alias para funciones  val myFun: MyFun = ::myFunction  // Clase Anidada (nested)  val myNestedClass2 = MyNestedClass() // Ejemplo de type alias  // Clase internas (inner)  // val myInnerClass2 = MyInnerClass() // No es válido, ya que aún es necesario instanciar la clase que contiene a la clase interna para poder instanciar desde ella, la clase interna |  |
| Destructuring | Nos permite trabajar de forma más eficiente, y utilizar menos código y de forma más limpia. Permite crear variables individuales a partir de tipos más complejos.  data class DataClase2 (val x: Int, val y: Int, val z: Int)  fun main() {  […]  /\*-- Destructuring --\*/  // Destructuring de un Pair  val (first, second) = par  println("first: $first, second: $second")  // Destructuring de un Map.Entry  val map = mapOf("a" to 1, "b" to 2)  for ((key, value) in map) {  println("key: $key, value: $value")  }  // Destructuring de un objeto de tipo data class (gracias a componentN)  val dc2: DataClase2 = DataClase2(1, 2, 3)  fun getMyDC2() = dc2  val (coordenadaX, coordenadaY, coordenadaZ) = dc2 // Destructuring de las 3 variables de la data class  val (coordenadaX2, coordenadaY2) = dc2 // Destructuring de las 2 primeras variables de la data class  val (coordenadaX3, \_, coordenadaY3) = dc2 // Destructuring de la segunda variable de la data class, mediante variable subrayada  val (coordenadaX4, coordenadaY4, coordenadaZ4) = getMyDC2() // Destructuring de las 3 variables de la data class, mediante función |  |