CONCEPTOS BÁSICOS KOTLIN

**Clases**

Para definir una clase usamos la palabra clave *class*.  
Si una clase implementa una interfaz usaremos “**:**”

class A : MyInterface

En el caso de que extendamos de una clase e implementemos otra, usaremos las comas.

class A : B(), MyInterface

**Herencia**

Por defecto todas las clases son “finales” por ello, si queremos extender de ella ésta debe ser declarada como *open* o *abstract*.  
Todas las clases extienden por defecto de *Any*  
Al extender una clase usaremos también los dos puntos y usaremos su constructor.

class A : B()

**Constructores**

**Lo básico**

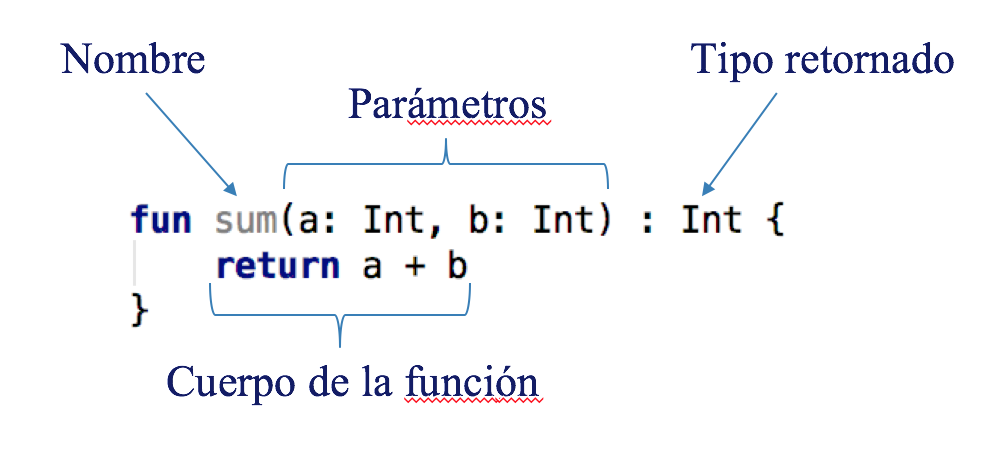
* Todas las clases tienen un constructor predeterminado por defecto
* Los tipos se escriben justo después del nombre
* No es necesario corchetes si la clase está vacía
* Declarar una estructura init si quieres implementar algo en el cuerpo de un constructor

**Funciones**

**Lo básico**

* Primero el nombre seguido de dos puntos y luego el tipo
* Podemos definir un valor por defecto a un parámetro para hacerlo opcional
* Al invocarlo, podemos usar el nombre del argumento que precede al valor
* Después del cierre de los parámetros iría el valor retornado precedido de dos puntos

**Estructura**



**Variables y propiedades**

En Kotlin todo es un objeto, por lo que no encontraremos tipos primitivos pero sí los tipos básicos

**Tipos básicos**

* Int
* Double
* Float
* Char
* Boolean
* String
* …

**Características**

**Diferencias con Java**

* No existen conversiones automáticas entre los tipos numéricos
* Puede ser declaradas mutables (var) o inmutables (val)
* Operaciones aritméticas a nivel de bit. Ej:  
  | -> or, & -> and
* No es necesario especificar el tipo, al inicializarla el compilador lo inferirá.
* Un String puede ser accedido como un array e incluso ser iterado.

**Propiedades**

* No es necesario especificar getters y setters
* Se pueden hacer accesos customizados a las variables con los bloques get() y set(value)
* Se pueden especificar como mutables e inmutables

**Nulabilidad**

**Control de los nulos**

Si una variable puede ser nula, el compilador nos obligará a manejarla de alguna manera y no nos dejará seguir si no hacemos comprobaciones previas

**Nomenclatura**

Como todo es un objeto, todo puede ser nulo. Para ello, simplemente debemos añadir un signo de interrogación al final de su tipo.

*val name: String? = null*

**Operadores**

* **?.** – Ej: *name?.toString()*

Sólo se ejecutará si la variable es distinta de nula. En caso contrario no hará nada.

* **?:** – Ej: *name?.toString() ?: “”*

Indicaremos qué hacer si la variable es nula

* **!!** – Ej: *name!!.toString()*

¡Cuidado! Estamos obligando al compilador a que ejecute dicho método aunque sea nula nuestra variable

**Función let**

Especialmente útiles para los nulos encadenados

*Person?.name?.let{printLog(it)}*

**Variables lateinit**

Variables no nulas pero que no podemos definir en el constructor, ya que por ejemplo la inicializaría un inyector de dependencias

* ¿Cómo sabemos si está inicializada?

Usaremos el método *.isInitialized*

**Casting de objetos**

**Operador *is***

Comprobará si una variable es del tipo al que precede. Si lo es, no será necesario hacer un casting posterior ya que esa variable será tratada con dicho tipo.

if (name is String)

log(name)

**Cast explícito**

Para realizar un cast expecífico usaremos el operador *as*

val aux = test as Num

**Inmutabilidad**

**Diferencias entre *val* y *var***

* *val* (de value). Variable inmutable, por lo que no se puede editar una vez inicializada.
* *var* (de variable). Variable mutable, es posible editarla cada vez que lo necesitemos.

**Mutabilidad de las colecciones**

* *Collection.* Contiene todos los métodos de iteración, consulta de tamaño, etc. Pero no los métodos de añadir y borrar elementos.
* *MutableCollection.* Extiende de *Collection* y añade los métodos de agregación, eliminación, limpieza, etc.

**Ejemplo**

* Creamos una lista de sólo lectura:

*val list = listOf(“Test1”, “Test2”)*

* Creamos una lista mutable:

*var mutableList = mutableListOf(“Test1”)*

*mutableList.add(“Test2”)*

**Consejo**

Se recomienda el uso de *val* y modificarlas a *var* si fuese necesario.

Al igual, si nuestras listas no se van a ver modificadas, se recomienda el uso de listas, conjuntos o mapas de sólo lectura.

**Iteradores *for* y *while***

### Diferencias con Java

En un principio, ambos operadores no tienen diferencias con Java, pero para los bucles for se añade la posibilidad del uso de rangos

### Rangos, operador in

Dos formas:

* **in numberFrom..numberTo**. Se iterará desde el valor especificado inicial hasta el final, éste último incluido.
* for (number 1..10) {
* }
* **in numberFrom until numberTo**. En este caso, el último valor no estará incluído.
* for (number in 1 until 10) {

}