FUNDAMENTOS KOTLIN

2021-2022

STA. MARÍA
La nueva
SAN JOSÉ
aRTesano

VARIABLES

- Variables inmutables **(de solo lectura):** Una vez asignado un valor, ya es no es posible modificarlo.
  - **Declaración**-> se usa la palabra reservada **val** seguida de un identificador. En el momento de la asignación se le indica el tipo de datos.

```
val a=2 // constante , en el momento de la asignación se asigna el tipo
// val a="hola" o val a=true
println(a)
```

- Variables mutables **(lectura/escritura):** Son aquellas que pueden cambiar el valor las veces que sea necesario.
  - Para **declarar** este tipo de variables se utiliza la palabra reservada **var**.

```
var b=2 // Variable, en el momento de la declaración asignamos un valor.

// Otra forma de declarar
var x:Int // indicando el tipo de dato que va a almacenar
```

Ejemplo:

```
val a=2 // constante , en el momento de la asignación se asigna el tipo
// val a="hola" o val a=true
println(a) // o podemos poner
println ("a=$a")

var b=2 // Variable, en el momento de la declaración asignamos un valor.

// Otra forma de declarar
var x:Int // indicando el tipo de dato que va a almacenar
println ( " b= $b")
```

El operador **$** en el interior de un String permite insertar el valor de la variable referenciada.

En Kotlin ningún objeto puede ser null, esto hace que se eviten errores en tiempo de ejecución, ya que son detectados en tiempo de compilación. Pero si necesitamos que una variable acepte este valor debemos indicárselo agregando la interrogación en la declaración

```
var objNull:String?
```

TIPOS EN KOTLIN

Números enteros

Tipo	Tamaño en bits	Límite inferior	Límite superior
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2^{-31}	$2^{31}-1$
Long	64	-2^{63}	$2^{63}-1$

Números Reales

Tipo	Tamaño en bits	Bits significativos	Bits del exponente	Dígitos decimales
Float	32	24	8	6-7
Double	64	53	11	15-16

Caracteres Char

Booleanos Boolean

```
val a=true
```

```
var ocurre:Boolean=false;
```

Cualquier tipo de objeto Any

```
// Cualquier objeto ,  se decidirá cuando se le asigne un valor
var objeto:Any
```

FUNCIONES

- Las funciones en Kotlin se declaran usando la palabra reservada **fun**
- Los parámetros se definen utilizando la notación identificados: tipo de datos, cuando hay varios parámetros estos se separan por comas.

```
fun sum(a:Int, b:Int):Int{

   return a+b
}
```

- Si una función no devuelve ningún valor , su tipo de retorno es Unit, equivalente a void en otros lenguajes. La declaración de Unit es opcional.

```
private fun sayHello(datos:String):Unit {
   println(datos)
}
```

INFIX (extensiones de una clase)

Las funciones marcadas con la palabra clave infix también se pueden llamar usando la notación infija (omitiendo el punto y los paréntesis para la llamada). Las funciones de infijo deben cumplir los siguientes requisitos:

- Deben ser funciones miembro o funciones de extensión(*Kotlin brinda la capacidad de extender una clase con nueva funcionalidad sin tener que heredar de la clase o usar patrones de diseño como Decorator . Esto se hace mediante declaraciones especiales llamadas extensiones . .*)
- Deben tener un solo parámetro.
- El parámetro no debe aceptar un número variable de argumentos y no debe tener un valor predeterminado

```
infix fun Int.enableAbs(enable:Boolean)=if (enable)abs(this)else this
```

```
val c=-3
println ( c.enableAbs(false))
println ("$a +$c=${sum(a,c)}")
println ("$a +$c=${sum(a,c.enableAbs(true))}")
```

Otro ejemplo

```
fun main() {
        val a = 20
        print((a >= 0) and (a <= 21))
 }
```

public infix fun and(other: Boolean): Boolean

a and b // es igual a a.and(b)

SOBRECARGA DE MÉTODOS

La sobrecarga es un mismo método con diferentes parámetros.

Argumentos predeterminados

Los parámetros de función pueden tener valores predeterminados, que se utilizan cuando omite el argumento correspondiente. Esto reduce el número de sobrecargas:

```
fun showProduct( name:String, promo:String="Sin promocion", vigencia:String="agotar existencias"){
   println ("$name=$promo hasta $vigencia" )
}
```

Argumentos nombrados

Al llamar a una función, puede nombrar uno o más de sus argumentos. Esto puede ser útil cuando una función tiene muchos argumentos .

Cuando usa argumentos con nombre en una llamada de función, puede cambiar libremente el orden en el que se enumeran y, si desea usar sus valores predeterminados, puede dejar estos argumentos por completo.

```
showProduct("soda","10%")
showProduct("PaN")
showProduct("Caramelo", "15%")
showProduct("Jugo",vigencia = "15 de marzo")
```

Número variable de argumentos (varargs)

Puede marcar un parámetro de una función (generalmente el último) con el modificador **vararg**.

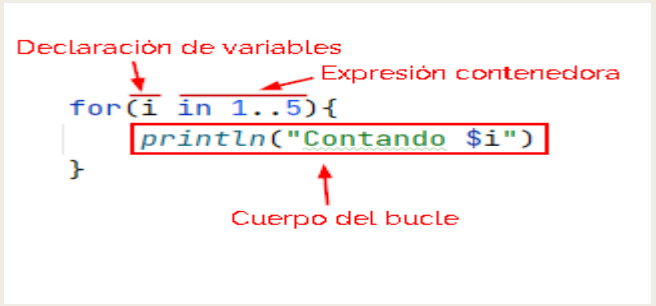
```
fun showPersons(numero:Int , vararg persons: String){

   for (person in persons) println(person)
```

BUCLES

El ciclo **for** itera a través de cualquier elemento que proporcione un iterador. Esto es equivalente al for each en lenguajes como C #. La sintaxis de for es la siguiente:

for (item in collection) print(item)

```
for (person in persons) println(person)
```



Mientras bucles

while y **do-while** son bucles que se ejecutan continuamente mientras se satisface su condición. La diferencia entre ellos es el tiempo de verificación de la condición:

- while comprueba la condición y, si se satisface, ejecuta el cuerpo y luego vuelve a la comprobación de condición.
- do-while ejecuta el cuerpo y luego verifica la condición. Si está satisfecho, el ciclo se repite. Entonces, el cuerpo de se do-while ejecuta al menos una vez independientemente de la condición.

```
var index = 0
while (index < persons.size){
   if (persons[index] == "Mary") println("Es Mary!")
   println(persons[index])
   index++
}
```

WHEN

when define una expresión condicional con múltiples ramas. Es similar a switch en lenguajes similares . Su forma simple se ve así.

```
when (persons[index]) {
   "Angel" -> println(" es Angel!")
   "Mary" -> {
      println("ir a otra pantalla")
      println("2+4")
   }
   else -> println(persons[index])
}
```

when compara su argumento con todas las ramas secuencialmente hasta que se cumpla alguna condición de rama.
when se puede utilizar como expresión o como declaración. Si se usa como expresión, el valor de la primera rama coincidente se convierte en el valor de la expresión general. Si se utiliza como declaración, se ignoran los valores de las ramas individuales. Al igual que con if, cada rama puede ser un bloque y su valor es el valor de la última expresión del bloque.
La rama else se evalúa si no se cumple ninguna de las otras condiciones de la rama. Si when se utiliza como una expresión, la rama else es obligatoria.

```
var x:Int=5
when (x) {
   in 1..10 -> print("x is in the range")
   in validNumbers -> print("x is valid")
   !in 10..20 -> print("x is outside the range")
   else -> print("none of the above")
}
```

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

```
val formResponse: Any = 12

when (formResponse) {
   is Int -> print("Respuesta Entera")
   is String -> print("Respuesta String")
}
```

EXCEPCIONES

Kotlin te permite capturar excepciones con la expresión **try..catch..finally.**

- Dentro del bloque try está el código que puede dar lugar a lanzar excepciones y luego añade bloques catch que verifiquen la aplicabilidad de un subtipo de excepción.
- El bloque finally se ejecuta independientemente de que no he haya ejecutado algún bloque catch. Normalmente aquí liberas los recursos que has tomado del sistema y limpias referencias para evitar fugas de memoria.

```
fun main() {
    println("5.3".toDoubleOrDefault(1.0))
    println("5.".toDoubleOrDefault(1.0))
    println(".3".toDoubleOrDefault(1.0))
    println("dos".toDoubleOrDefault(1.0))
}

fun String.toDoubleOrDefault(defaultValue: Double): Double {
    return try {
        toDouble()
    } catch (e: NumberFormatException) {
        defaultValue
    }
}
```

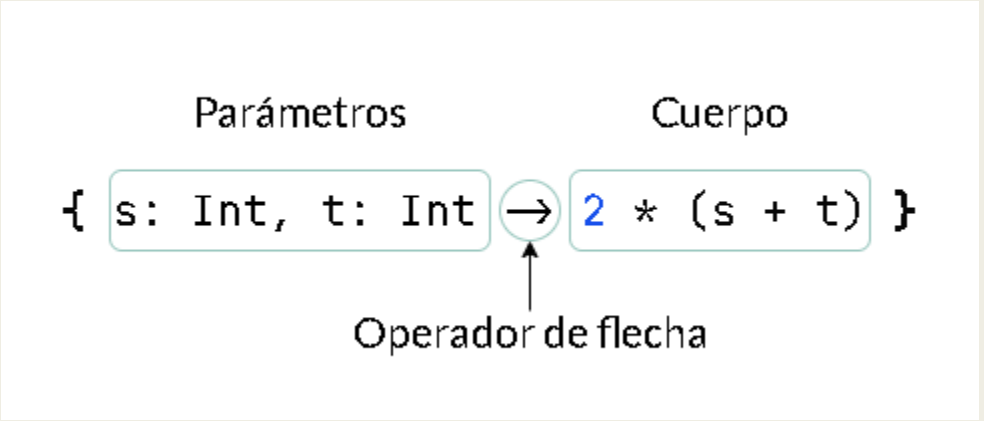
```
val count: Int = try { number!!.count() } catch (e: Exception) { 0 }
```

LAMBDAS

Las expresiones lambda y las funciones anónimas son *literales de función* . Los literales de función son funciones que no se declaran pero que se pasan inmediatamente como una expresión.

La sintaxis de un literal lambda va al interior de dos llaves {}.
Sus componentes son:

- Lista de parámetros — Cada parámetro es una declaración de variable, aunque esta lista es opcional
- Operador de flecha -> — Se omite si no usas lista de parámetros
- Cuerpo del lambda — Son las sentencias que van luego del operador de flecha



CLASES

Las clases en Kotlin se declaran usando la palabra clave class:

```
class Person { /*...*/ }
```

La declaración de clase consta del nombre de la clase, el encabezado de la clase (especificando sus parámetros de tipo, el constructor principal y algunas otras cosas) y el cuerpo de la clase rodeado por llaves. Tanto el encabezado como el cuerpo son opcionales; si la clase no tiene cuerpo, se pueden omitir las llaves.

CONSTRUCTORES

- Una clase en Kotlin puede tener un constructor principal y uno o varios constructores secundarios.
- El constructor principal es parte del encabezado y va a continuación del nombre de la clase
  - ✓ El constructor primario o principal, hace parte de la cabecera de la clase. Este recibe como argumentos, aquellos datos que necesitas explícitamente para inicializar las propiedades al crear el objeto.
  - ✓ En su sintaxis concisa, si declara con **val o var** antes del parámetro de un constructor primario, creas una propiedad automáticamente en la clase.
  - ✓ Usa la palabra reservada **constructor** luego del nombre de la clase:

    class ClaseEjemplo constructor(val propiedad1:Tipo, var propiedad2:Tipo, ...)
  - ✓ Si no tienes anotaciones o modificadores de visibilidad puedes omitirla:

    class ClaseEjemplo(val propiedad1:Tipo, var propiedad2:Tipo, ...)

Bloques De Inicialización

Es posible expandir la inicialización de las propiedades usando la sección ***init*** en la clase para el constructor primario.

En ella incluyes como cuerpo la lógica de asignación.

Por ejemplo, la clase Weapon puede ser escrita en forma expandida de la siguiente forma:

```
class Weapon(attack: Int, speed: Double)
    { val attack: Int
     val speed: Double
    init {
                this.attack = attack
                this.speed = speed }
     }
```

Con la sintaxis completa, las propiedades son declaradas al interior de la clase y los parámetros pasan a solo ser parámetros de constructor sin val ni var.

Si los nombres de los parámetros del constructor se llaman igual a las propiedades, usa la expresión this para acceder a la propiedad de la instancia que intentas generar con el constructor.

Constructores Secundarios

Si la lista de argumentos del constructor primario no satisface la creación de tu objeto en alguna circunstancia, entonces puedes crear un constructor secundario a la medida.

Su declaración se realiza a través de constructor al interior de la clase.

Si tienes un constructor primario es obligatorio usar la expresión this para delegarle los parámetros que requiera.

Luego escribe la lógica de inicialización en el bloque.

```
class Clase{
        constructor(parametro1:Tipo, parametro2:Tipo /*,..*/): this(/*parametros*/){
        /* Cuerpo de constructor */
        }
}
```

HERENCIA

Para aplicar herencia en Kotlin, añade a la sintaxis de declaración de la clase así:

- Añade el modificador ***open*** para habilitar su capacidad de herencia
- Añade dos puntos para expresar en la cabecera, que extiende de otra
- Añade los paréntesis al final del nombre de la superclase para especificar la llamada a su constructor

```
open class Phone( protected val number:Int) {
```

Constructor Primario En Herencia

Si la superclase tiene constructor primario, debes inicializarlo pasando los parámetros en la llamada de la sintaxis de herencia.

open class Ancestro(val propiedad:Boolean)
class Descendiente(propiedad: Boolean) : Ancestro(propiedad)

```
class SmartPhone(number:Int ,val isPrivate:Boolean):Phone(number)
```

Sobrescribir Métodos

Aplicar polimorfismo con la sobreescritura de métodos en Kotlin, requiere habilitar el método con el modificador open.

Luego usa el modificador override desde el método polifórmico la subclase.

```
open fun showNumber(){
    println(" Mi numero es : $number")
 }
```

```
override fun showNumber() {
   if (isPrivate)println("Desconocido")
   else{
            println( super.showNumber())}
}
```

DATA CLASS

- El modificador data facilita la creación de clases cuyo propósito es solo almacenar valores. A estas clases se les llama, clases de datos o data classes en Kotlin.

```
data class User( val id:Long, var name:String,var lastName:String, var group:Int) {
```

Al marcar la clase User con data, este le dice al compilador Kotlin que le autogenere los siguientes métodos, tomando como base sus propiedades en el constructor primario:

- equals
- hashCode
- copy
- toString
- N métodos componentN() que corresponden a cada propiedad en orden de declaración

COLECCIONES

- *Una **lista de solo lectura*** (*read-only list*) puede ser consultada después de ser inicializada, pero no permite el uso de comandos para cambiar su estado.

```
val frutaList=listOf("Manzana","Banana","Uva","Lima")
```

Para acceder al estado de la lista puedes usar los siguientes miembros básicos:

- ***size** para obtener la cantidad de elementos de la lista*
- *Para acceder a un elemnto de la posición index **get(index)***
- ***indexOf(element)** para obtener el índice de la primera ocurrencia de element*
- ***lastIndexOf(element)** para obtener el índice de la última ocurrencia del element*
- ***subList(fromIndex, toIndex)** para obtener una porción de la lista en el rango (fromIndex, toIndex)*

```
println(frutaList.get((0..frutaList.size-1).random()))
```

```
println("Index de Uva es ${frutaList.indexOf("Uva")}")
```

- Una lista mutable es representada por la interfaz **MutableList<E>**. Este tipo de listas pueden ser consultadas como List<E>,pero permiten añadir, cambiar y remover elementos.

```
val myUser = User(0, "marai", "pardo", Group.FAMILY.ordinal)

 val bro2 = myUser.copy(1, "pepe")

val friend=bro2.copy(id=2,group= Group.FRIENDS.ordinal)
val userList = mutableListOf<User>( myUser,bro2)
```

Los métodos para modificar el contenido son:

- **add(element)** para añadir un nuevo ítem en la parte superior de la lista
- **add(index, element)** para insertar a ítem en un índice
- **removeAt(index)** para eliminar ítem en un índice
- **[index]=element**, para reemplazar un ítem en el índice. Esta construcción es equivalente al operador **set(index, element)**

```
userList.add(friend)
```

```
userList.remove(bro2)
```

```
userSelectedList.set(0,bro2)
```

Un mapa es una colección que almacena sus elementos (entradas) en forma de pares clave-valor. Esto quiere decir que a cada clave le corresponde un solo valor y será única como si se tratase de un identificador.

```
val userMap=mutableMapOf<Int, User>()
```

Algunos métodos son:

- **mapa[clave]:** Esta sintaxis permite obtener el valor a partir de la clave en el corchete. Es la construcción equivalente al operador get(clave)
- **getOrDefault(key, defaultValue):** Obtiene el valor correspondiente a la clave, de lo contrario retorna a defaultValue
- **isEmpty():** Retorna true si el mapa no contiene entradas y false en caso contrario
- **containsKey(key):** Retorna true si key existe en el mapa. Esto es equivalente a usar el operador in al comprar la clave frente al mapa
- **containsValue(value):** Retorna true si una o varias claves se relacionan con value

```
userMap.put( myUser.id.toInt(), myUser)
```

```
println(userMap.keys)
println(userMap.values)
```

```
println(userMap.containsKey(0))
```

```
userMap.remove(2)
```