Tutorial

Kotlin Ya

Tutoriales Programación Ya

Tutoriales en PDF .com

1 - Instalación de Kotlin

Kotlin es un lenguaje de programación bastante nuevo desarrollado por la empresa JetBrains (https://www.jetbrains.com/) y que últimamente está tomando vuelo gracias entre otras cosas como ser el segundo lenguaje de programación aceptado oficialmente por Google para la plataforma Android.

Actualmente cuando compilamos con Kotlin un programa se genera código JVM (Java Virtual Machine) que debe ser interpretado por una máquina virtual de Java.

Como genera un código intermedio para la máquina virtual de java los programas en Kotlin pueden interactuar fácilmente con librerías codificadas en Java.

Kotlin al ser un lenguaje nuevo introduce muchas características que no están presentes en Java y facilitan el desarrollo de programas más seguros, concisos y compatibles con la plataforma Java.

El proyecto de Kotlin no se cierra solo al desarrollo de aplicaciones móviles para Android sino para el desarrollar aplicaciones de servidor y otras plataformas.

Para poder trabajar con Kotlin debemos instalar:

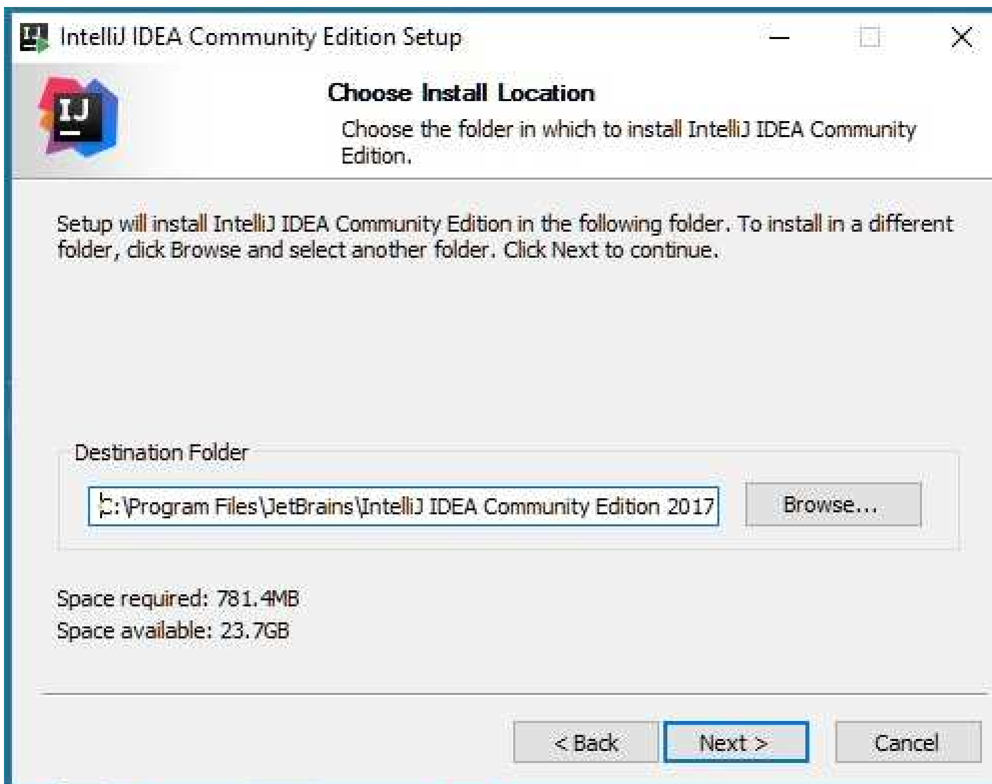
- Primero debemos descargar e instalar el JDK (Java Development Kit), los pasos para descargar e instalar los podemos seguir aquí. (http://tutorialesprogramacionya.com/javaya/detalleconcepto.php?punto=1&codigo=74&inicio=0)

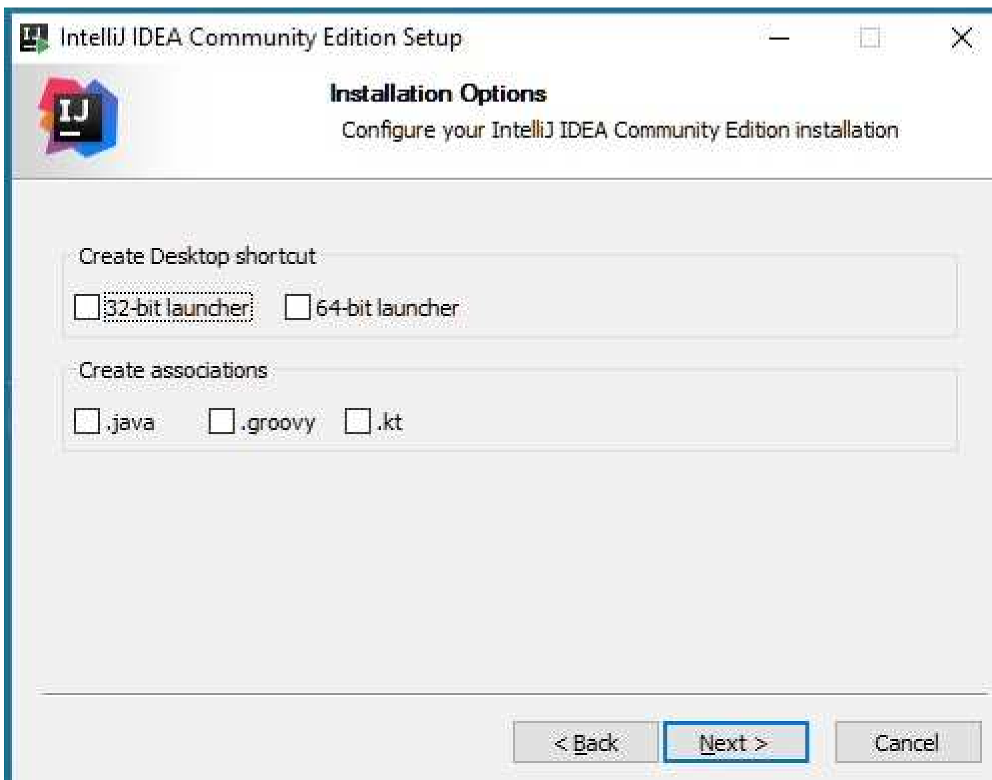
- El entorno de desarrollo más extendido para el desarrollo en Kotlin es el IntelliJ IDEA (https://www.jetbrains.com/idea/?fromMenu#chooseYourEdition)
  Podemos descargar la versión Community que es gratuita.
  Luego de conocer todas las características de Kotlin utilizaremos el Android Studio para desarrollar aplicaciones móviles.

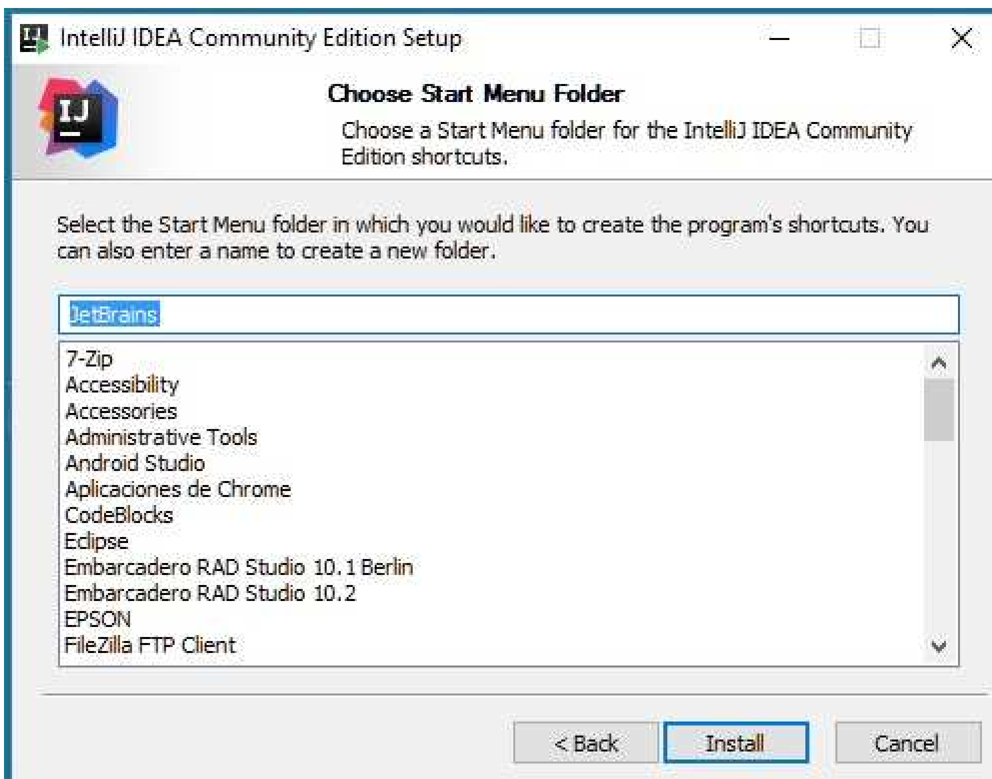
Instalación de IntelliJ IDEA

Una vez descargado el archivo de IntelliJ IDEA procedemos a ejecutarlo para su instalación:

Podemos dejar por defecto la carpeta de instalación y demás datos de configuración:



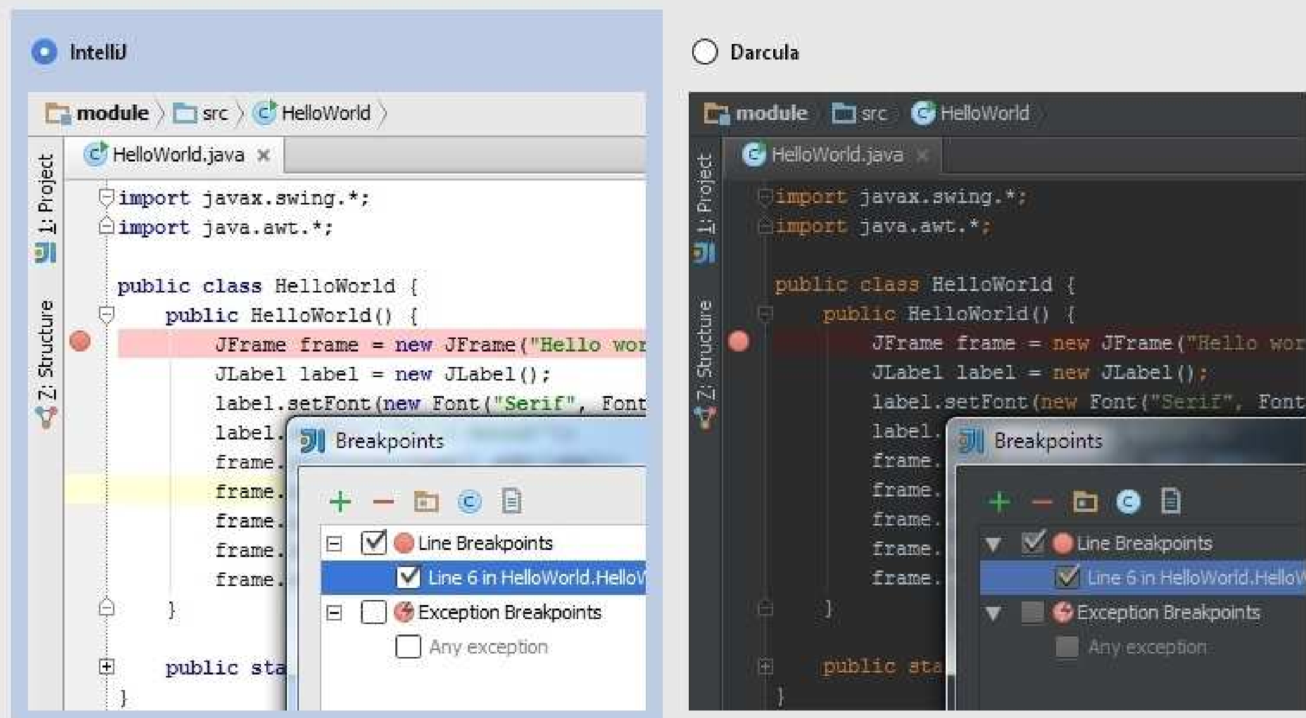




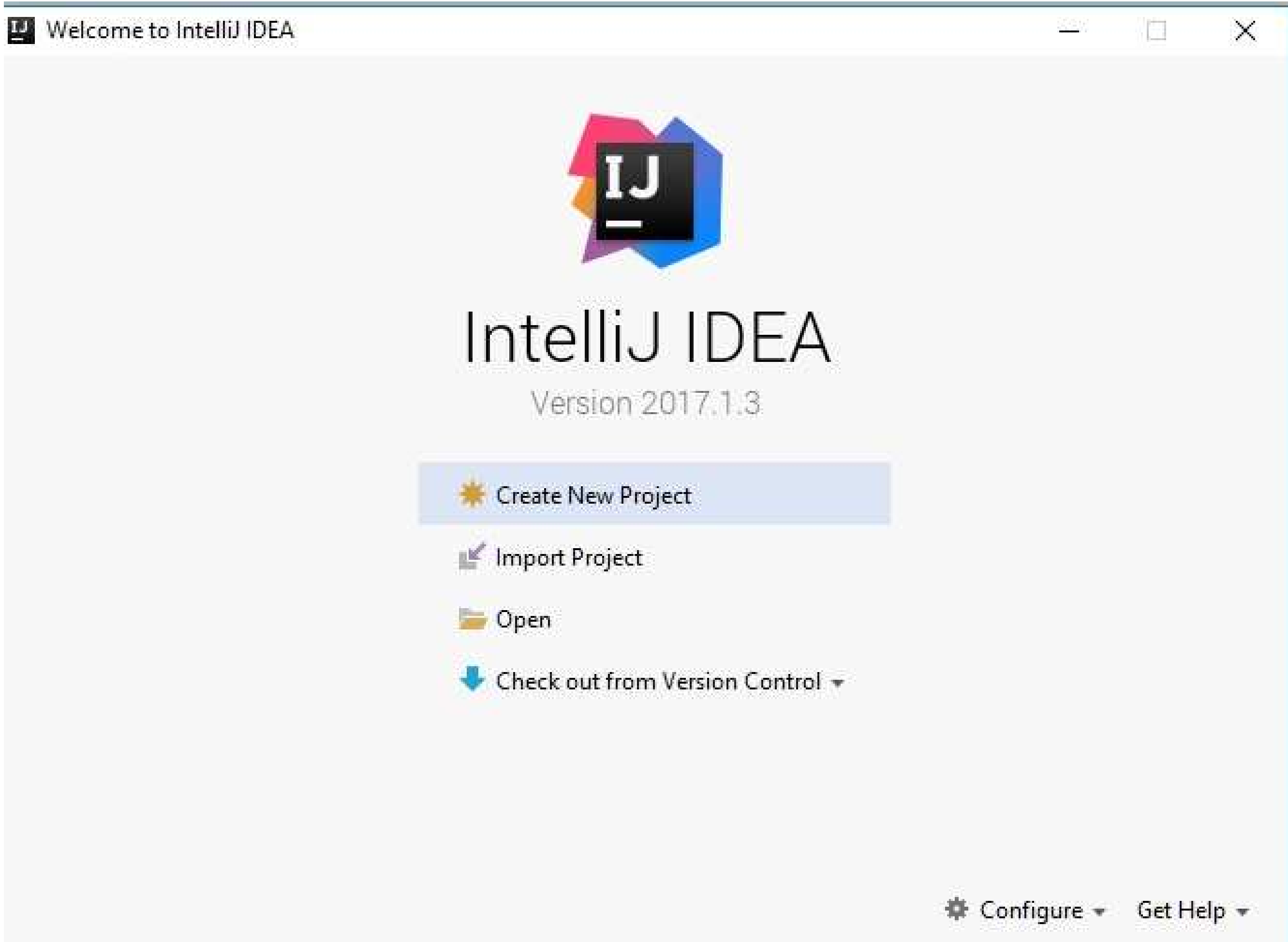
Ahora desde el menú de opciones de Windows podemos iniciar el entorno de desarrollo IntelliJ IDEA:

La primera vez que lo ejecutamos nos pregunta si queremos importar alguna configuración previa (elegimos que no):

También esta primera ejecución nos permite seleccionar el color del entorno de desarrollo (claro u oscuro), aquí presionamos el botón para que el resto de la configuración la haga por defecto:



Ahora si tenemos la pantalla que aparecerá siempre que necesitemos crear un nuevo proyecto o abrir otros existentes:



Retornar (index.php?inicio=0)

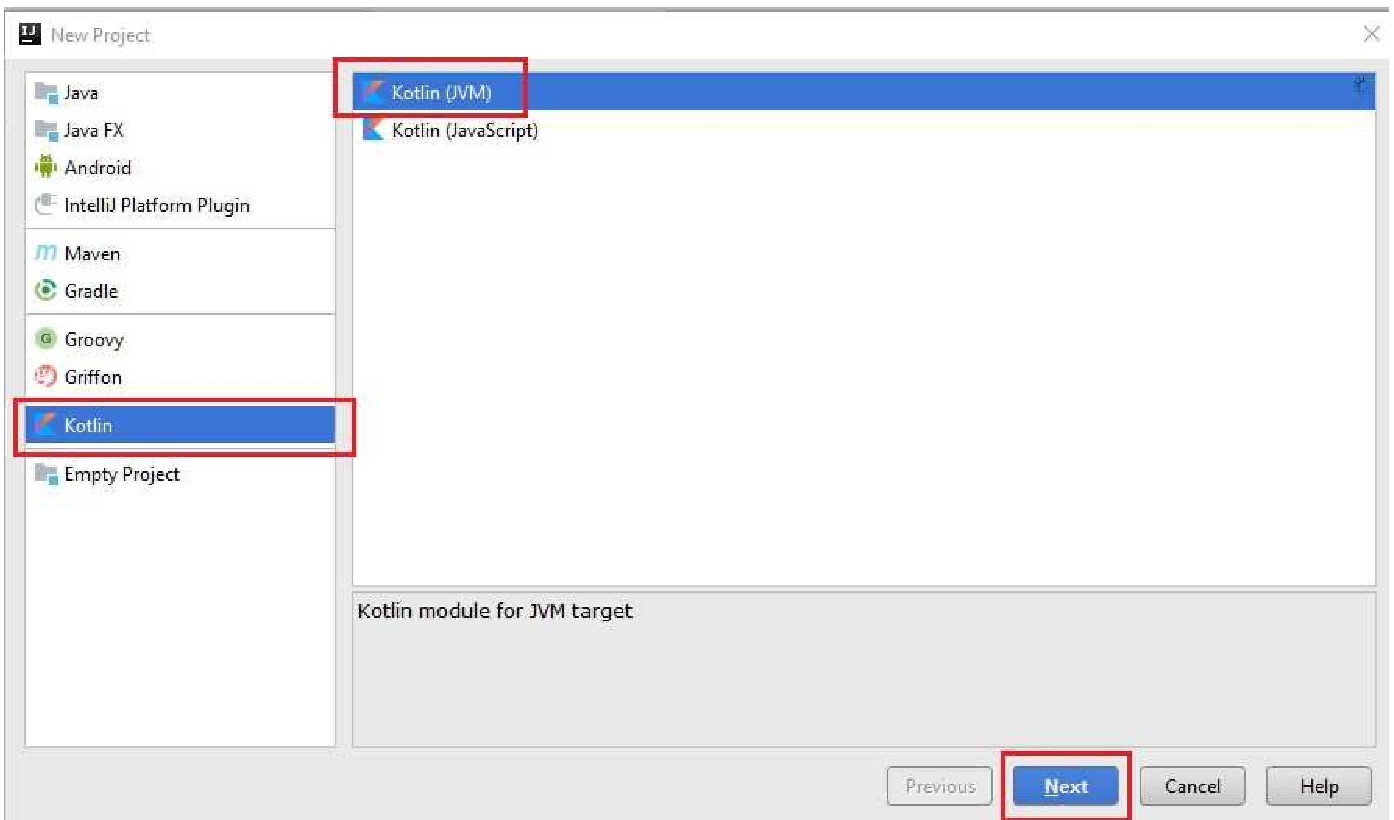
2 - Pasos para crear un programa en Kotlin

Vimos en el concepto anterior que cuando iniciamos IntelliJ IDEA y todavía no se ha creado algún proyecto aparece la siguiente pantalla:

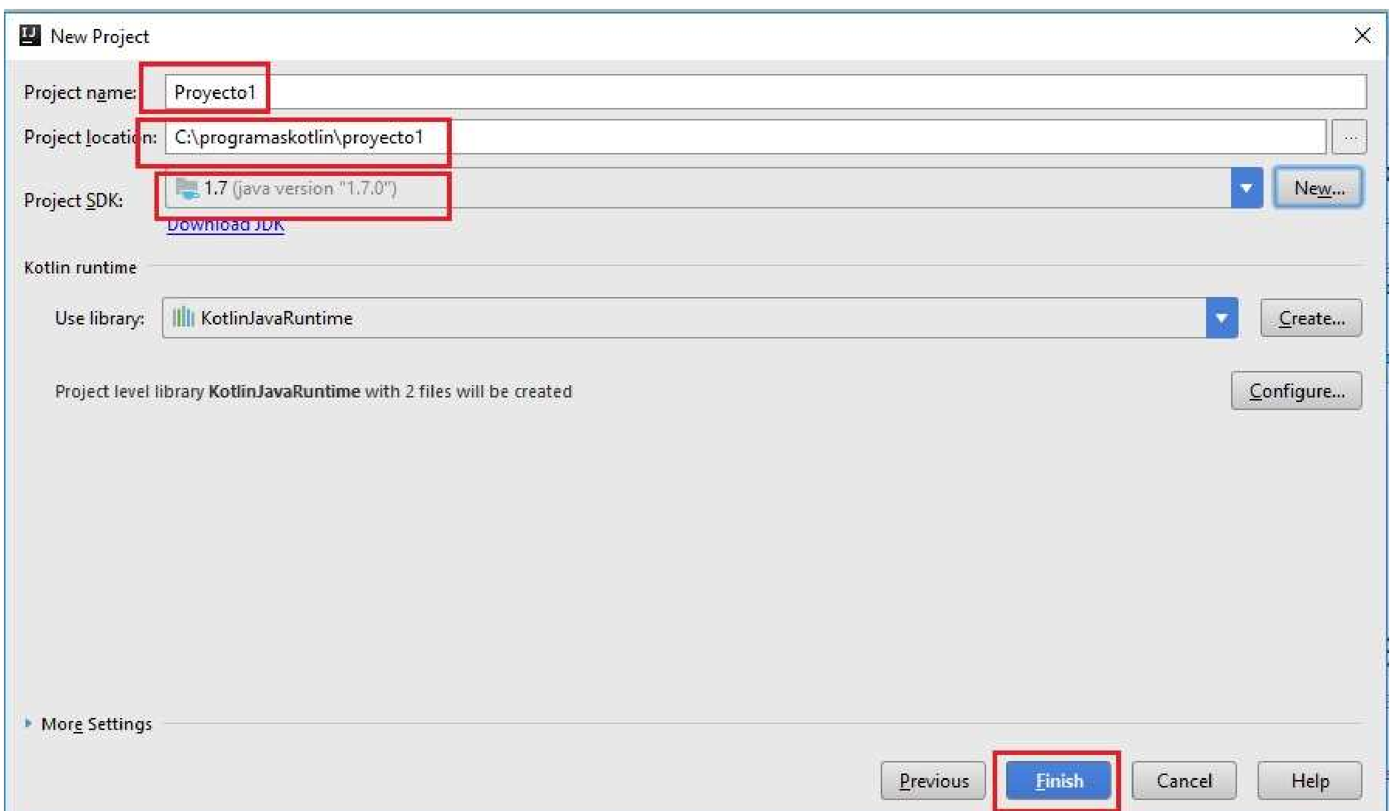
Primero podemos crear una carpeta donde almacenaremos todos los proyectos que desarrollaremos en Kotlin, yo propongo la carpeta:

```
c:\programaskotlin
```

Pasemos a crear nuestro primer proyecto en Kotlin, elegimos la opción "Create New Proyect" y seguidamente aparece el diálogo:

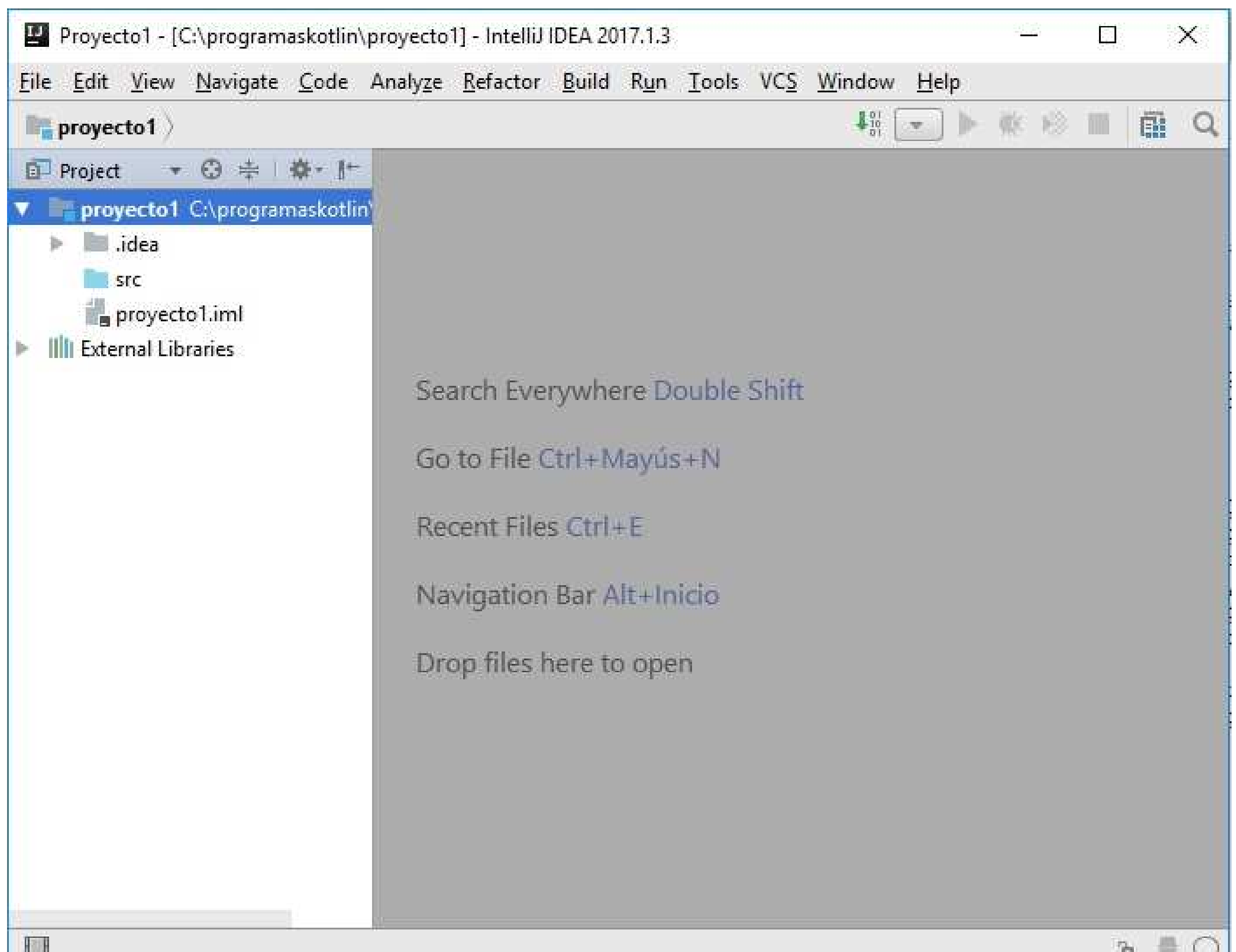


Ahora debemos indicar el nombre del proyecto y la carpeta donde se almacenará (indicamos la carpeta que creamos anteriormente desde Windows) y una subcarpeta que la creará el mismo IntelliJ IDEA llamada proyecto1:



Si no aparece el SDK debemos buscar la carpeta donde se instaló el SDK de Java que instalamos como primer paso en el concepto anterior.

Ahora si tenemos el entorno de IntelliJ IDEA abierto y preparado para nuestro "Proyecto1":



En la ventana "Project" en la carpeta "src" es donde debemos crear los archivos fuentes en Kotlin, presionamos el botón derecho del mouse sobre esta carpeta y seleccionamos la opción "New -> Kotlin File/Class":

Proyecto1 - [C:\programaskotlin\proyecto1] - IntelliJ IDEA 2017.1.3

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

proyecto1 〉 src 〉

Project

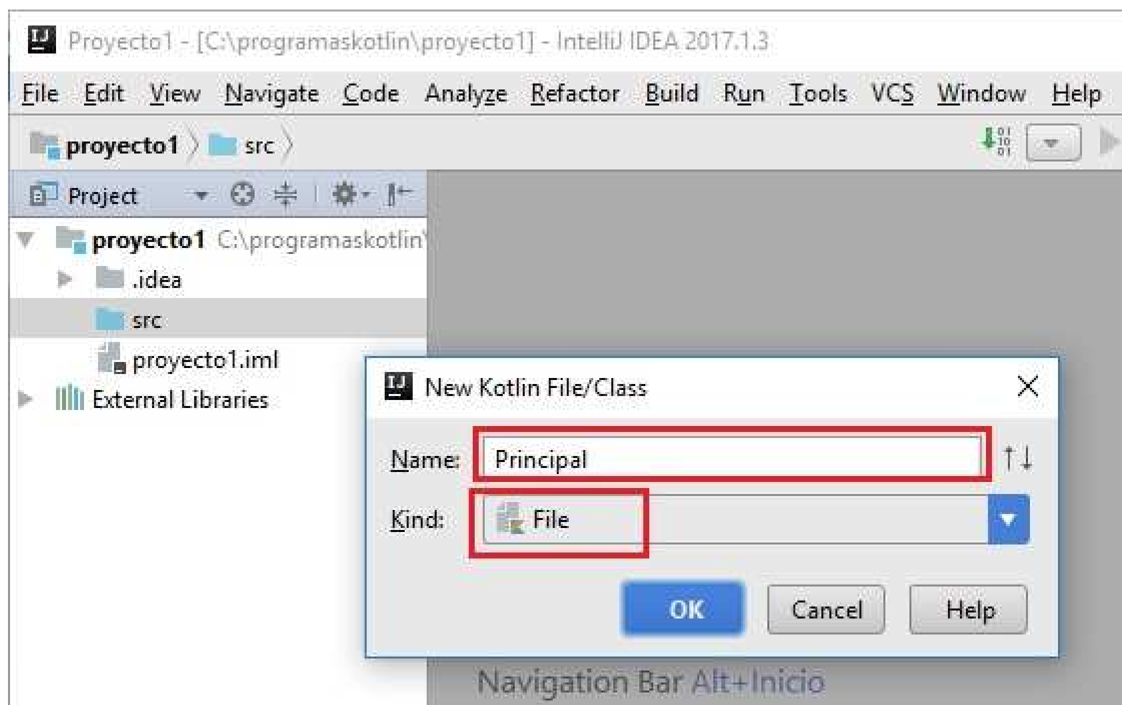
proyecto1 C:\programaskotlin\
.idea
src
Exter

New
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Mayús+C
Copy as Plain Text
Copy Reference Ctrl+Alt+Mayús+C
Paste Ctrl+V
Find Usages Alt+F7
Find in Path... Ctrl+Mayús+F
Replace in Path... Ctrl+Mayús+R
Analyze
Refactor
Add to Favorites
Show Image Thumbnails Ctrl+Mayús+T
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Suprimir

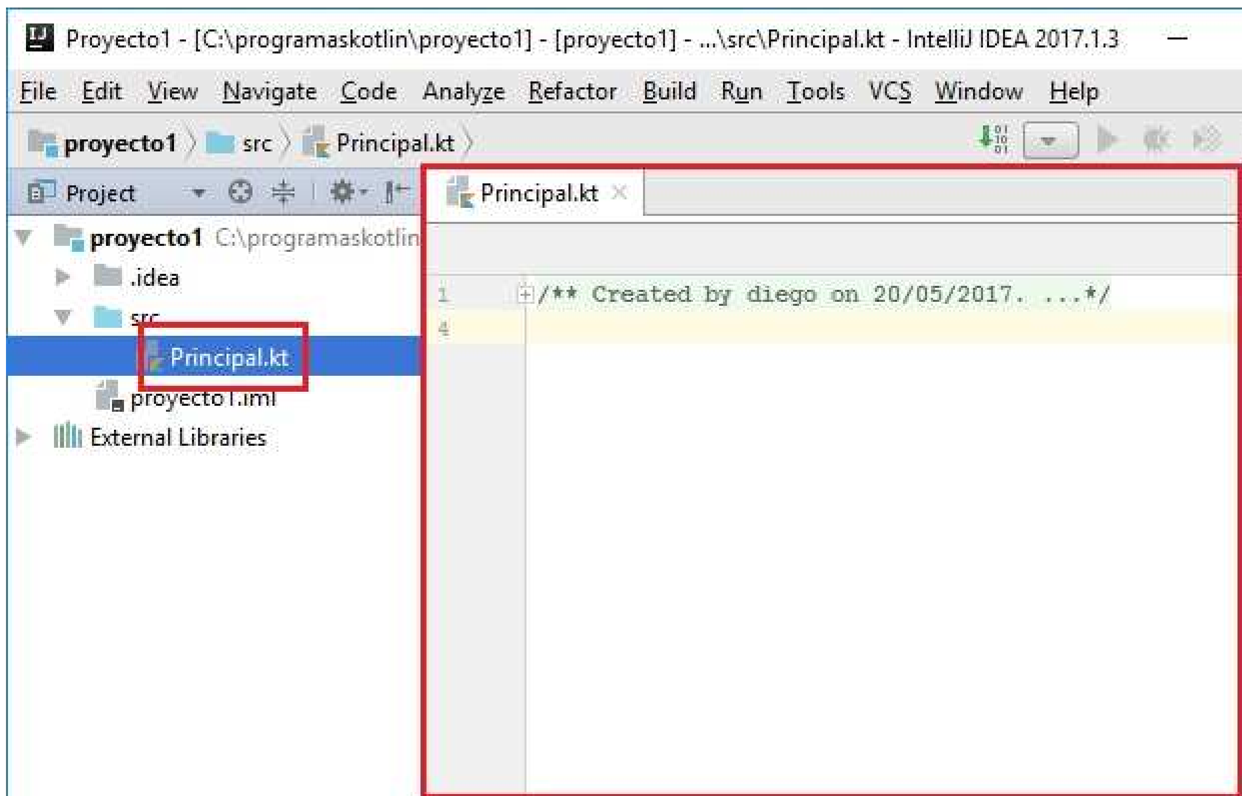
Java Class
Kotlin File/Class
File
Scratch File Ctrl+Alt+Mayús+Insertar
Package
FXML File
package-info.java
HTML File
JavaFXApplication
Singleton
XSLT Stylesheet
Edit File Templates...
GUI Form
Dialog
Form Snapshot
Resource Bundle

Create

Aparece un diálogo donde indicamos que queremos crear un archivo "File" con el nombre "Principal":



Luego de confirmar se ha creado el archivo "Principal.kt" donde almacenaremos nuestro primer programa en Kotlin:



El IntelliJ IDEA genera un comentario en el archivo en forma automática:

```
/**
 * Created by diego on 20/05/2017.
 */
```

Los comentarios luego el compilador de Kotlin no los tiene en cuenta y tienen por objetivo dejar documentado el programa por parte del desarrollador (los comentarios se encierran entre los caracteres /** */)

Problema

Crear un programa mínimo en Kotlin que muestre un mensaje por la consola.

Proyecto1 - Principal.kt

```
/**
 * Created by diego on 20/05/2017.
 */

fun main(parametro: Array<String>) {
    print("Hola Mundo")
}
```

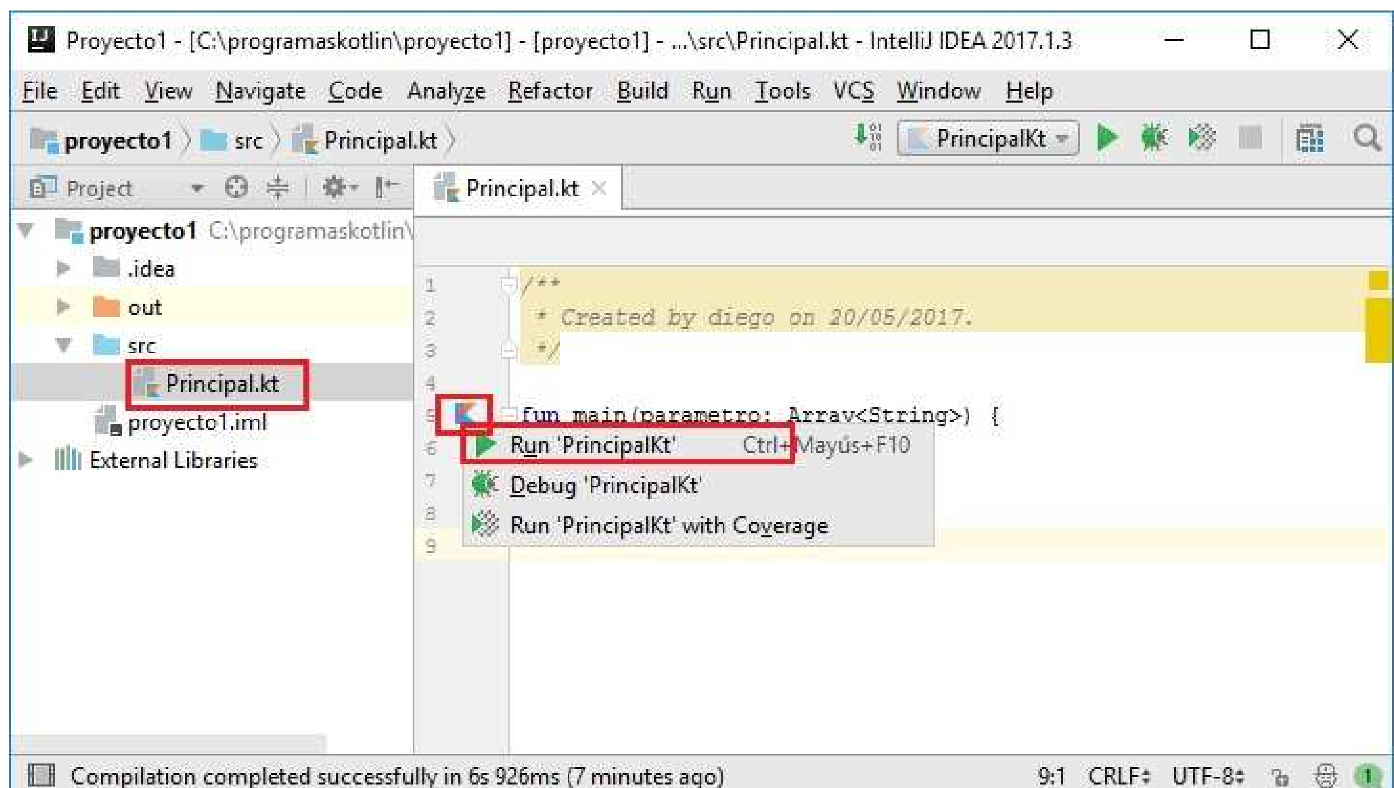
Todo programa en Kotlin comienza en la función main. Luego veremos el objetivo del dato que recibe la función main entre paréntesis (parametro: Array<String>), por ahora lo escribiremos y no lo utilizaremos.

Una función comienza con la palabra clave fun luego entre paréntesis llegan los parámetros y entre llaves disponemos el algoritmo que resuelve nuestro problema.

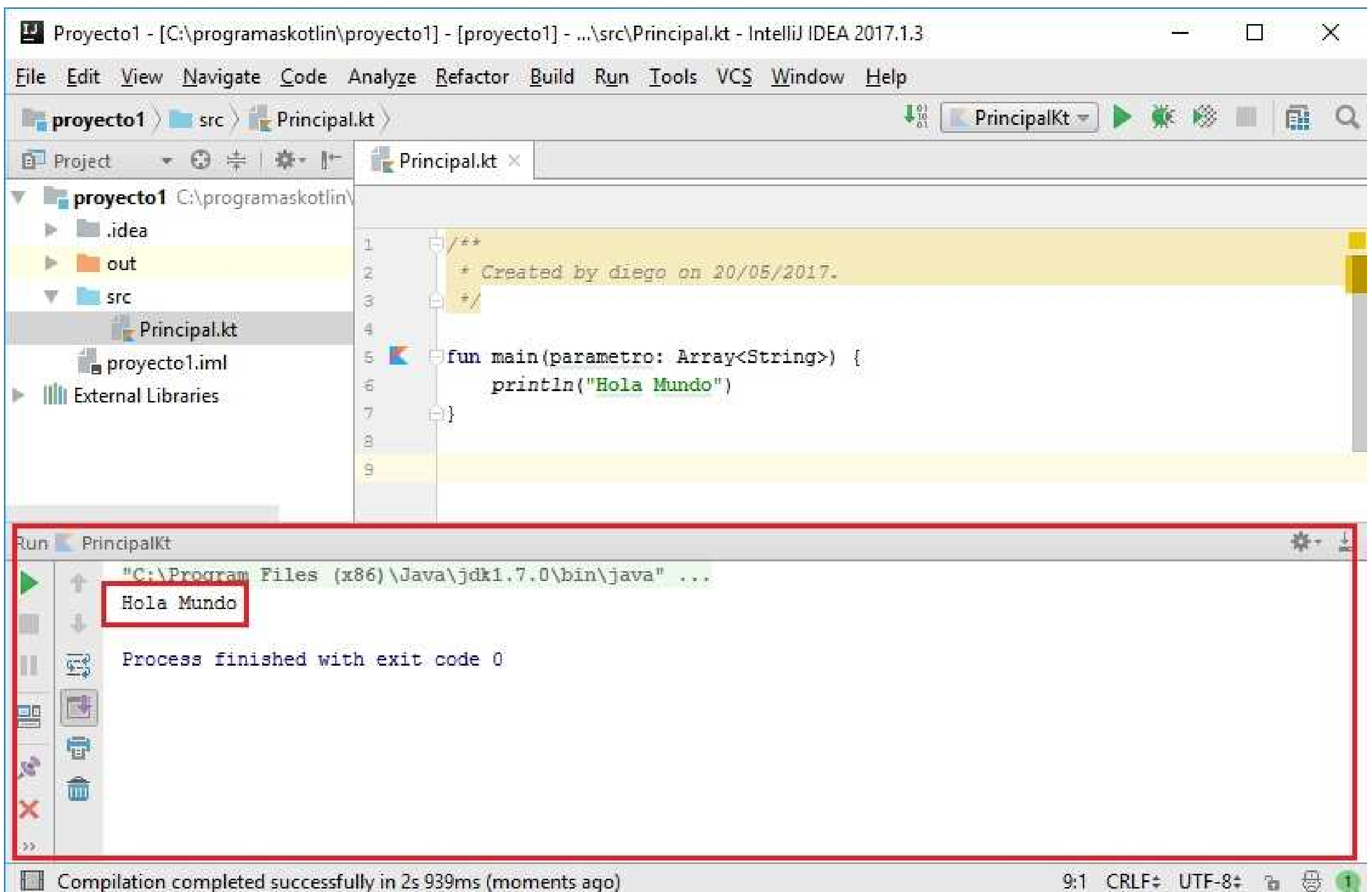
Si queremos mostrar un mensaje por la Consola debemos utilizar la función "print" y entre comillas dobles el mensaje que queremos que aparezca:

```
    print("Hola Mundo")
```

Para ejecutar el programa tenemos varias posibilidades, una es presionar el ícono que aparece al lado de la función main y elegir "Run Principal.kt":



Luego de esto aparece la ventana de la Consola con el resultado de la ejecución del programa:



También podemos ejecutar el programa seleccionando desde el menú de opciones "Run -> Run..."

Cada vez que desarrollemos un programa tenemos que tener en claro todos los pasos que dimos para crear el proyecto, crear el archivo "Principal.kt", codificarlo, compilarlo y ver si los resultados son los deseados.

Retornar (index.php?inicio=0)

3 - Tipos de variables

Una variable es un depósito donde hay un valor. Consta de un nombre y pertenece a un tipo.

En el lenguaje Kotlin si necesitamos almacenar un valor numérico entero podemos definir una variable de tipo:

```
Byte
Short
Int
Long
```

Según el valor máximo a almacenar utilizaremos alguna de estos cuatro tipos de variables enteras. Por ejemplo en una variable de tipo Int podemos almacenar el valor máximo: 2147483647 y en general tenemos:

```
Tipo de variable
                                   mínimo                          máximo
Byte                                 −128                            +127
Short                             −32 768                         +32 767
Int                        −2 147 483 648                  +2 147 483 647
Long           −9 223 372 036 854 775 808      +9 223 372 036 854 775 807
```

Si tenemos que almacenar un valor con parte decimal (es decir con coma como puede ser el 3.14) debemos utilizar una variable de tipo:

```
Double
Float
```

El tipo Double tiene mayor precisión que el tipo Float.

Y otro tipo de variables que utilizaremos en nuestros primeros ejercicios serán las variables de tipo String que permiten almacenar un conjunto de caracteres:

```
String
```

Una variable en Kotlin puede ser inmutable, esto significa que cuando le asignamos un valor no puede cambiar más a lo largo del programa, o puede ser mutable, es decir que puede cambiar el dato almacenado durante la ejecución del programa.

Para definir una variable en Kotlin inmutable utilizamos la palabra clave val, por ejemplo:

```
val edad: Int
edad = 48
val sueldo: Float
sueldo = 1200.55f
val total: Double
total = 70000.24
val titulo: String
titulo = "Sistema de Ventas"
```

Hemos definido cuatro variables y le hemos asignado sus respectivos valores.

Una vez que le asignamos un valor a una variable inmutable su contenido no se puede cambiar, si lo intentamos el compilador nos generará un error:

```
val edad: Int
edad = 48
edad = 78
```

Si compilamos aparece un error ya que estamos tratando de cambiar el contenido de la variable edad que tiene un 48. Como la definimos con la palabra clave val significa que no se cambiará durante toda la ejecución del programa.

En otras situaciones necesitamos que la variable pueda cambiar el valor almacenado, para esto utilizamos la palabra clave var para definir la variable:

```
var mes: Int
mes = 1
// algunas líneas más de código
mes = 2
```

La variable mes es de tipo Int y almacena un 1 y luego en cualquier otro momento del programa le podemos asignar otro valor.

Problema

Crear un programa que defina dos variables inmutables de tipo Int. Luego definir una tercer variable mutable que almacene la suma de las dos primeras variables y las muestre. Seguidamente almacenar en la variable el producto de las dos primeras variables y mostrar el resultado.

Realizar los mismos pasos vistos anteriormente para crear un proyecto y crear el archivo Principal.kt donde codificar el programa respectivo (Si tenemos abierto el IntelliJ IDEA podemos crear un nuevo proyecto desde el menú de opciones: New -> Project)

Proyecto2 - Principal.kt

```
fun main(parametro: Array<String>) {
    val valor1: Int
    val valor2: Int
    valor1 = 100
    valor2 = 400
    var resultado: Int
    resultado = valor1 + valor2
    println("La suma de $valor1 + $valor2 es $resultado")
    resultado = valor1 * valor2
    println("El producto de $valor1 * $valor2 es $resultado")
}
```

Definimos e inicializamos dos variables Int inmutables (utilizamos la palabra clave val):

```
    val valor1: Int
    val valor2: Int
    valor1 = 100
    valor2 = 400
```

Definimos una tercer variable mutable también de tipo Int:

```
    var resultado: Int
```

Primero en la variable resultado almacenamos la suma de los contenidos de las variables valor1 y valor2:

```
    var resultado: Int
    resultado = valor1 + valor2
```

Para mostrar por la Consola el contenido de la variable $resultado utilizamos la función println y dentro del String que muestra donde queremos que aparezca el contenido de la variable le antecedimos el caracter $:

```
    println("La suma de $valor1 + $valor2 es $resultado")
```

Es decir en la Consola aparece:

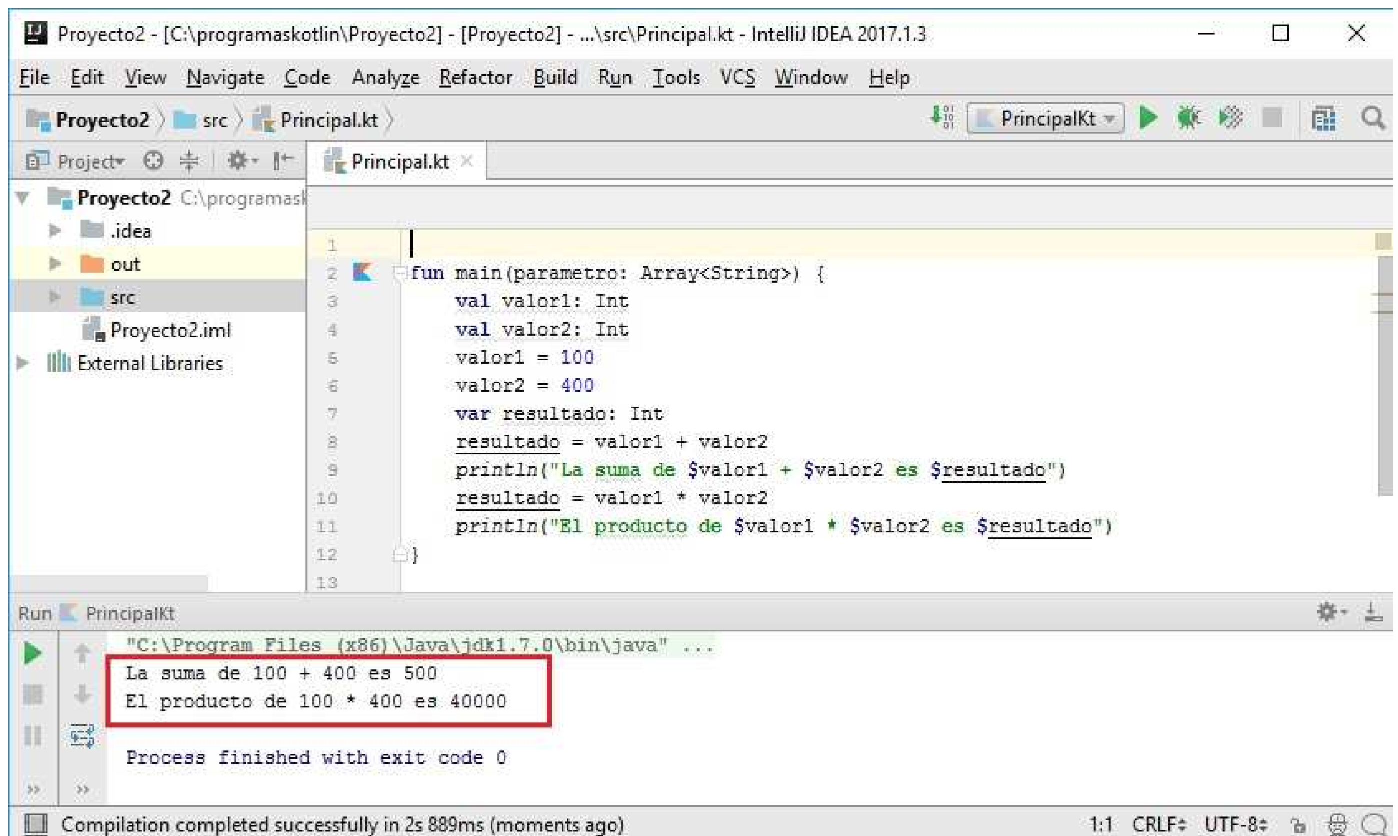
```
La suma de 100 + 400 es 500
```

Como la variable resultado es mutable podemos ahora almacenar el producto de las dos primeras variables:

```
    resultado = valor1 * valor2
    println("El producto de $valor1 * $valor2 es $resultado")
```

kotlin sustituye todas las variables por su contenido en un String.

El resultado de la ejecución de este programa será:



Conciso

Si entramos a la página oficial de Kotlin (https://kotlinlang.org/) podemos ver que una de sus premisas es que un programa en Kotlin sea "CONCISO" (es decir que se exprese un algoritmo en la forma más breve posible)

Haremos un primer cambio al Proyecto2 para que sea más conciso:

```
fun main(parametro: Array<String>) {
    val valor1: Int = 100
    val valor2: Int = 400
    var resultado: Int = valor1 + valor2
    println("La suma de $valor1 + $valor2 es $resultado")
    resultado = valor1 * valor2
    println("El producto de $valor1 * $valor2 es $resultado")
}
```

En este primer cambio podemos observar que en Kotlin podemos definir la variable e inmediatamente asignar su valor. Podemos asignar un valor literal como el 100:

```
    val valor1: Int = 100
```

o el contenido de otras variables:

```
    var resultado: Int = valor1 + valor2
```

Otra paso que podemos dar en Kotlin para que nuestro programa sea más conciso es no indicar el tipo de dato de la variable y hacer que el compilador de Kotlin lo infiera:

```
fun main(parametro: Array<String>) {
    val valor1 = 100
    val valor2 = 400
    var resultado = valor1 + valor2
    println("La suma de $valor1 + $valor2 es $resultado")
    resultado = valor1 * valor2
    println("El producto de $valor1 * $valor2 es $resultado")
}
```

El resultado de compilar este programa es lo mismo que los anteriores. El compilador de Kotlin cuando hacemos:

```
    val valor1 = 100
```

deduce que queremos definir una variable de tipo Int

Si en la variable valor1 almacenamos el número 5000000000, luego el compilador de Kotlin puede inferir que se debe definir una variable de tipo Long

```
    val valor1 = 5000000000
```

Para trabajar con los valores decimales por inferencia debemos utilizar la siguiente sintaxis:

```
    var peso = 4122.23 // infiere que es Double
    val altura = 10.42f // debemos agregarle la f o F al final para que sea un
Float y no un Double
```

Muy fácil es para definir un String:

```
    val titulo = "Sistema de Facturación"
```

Utilizaremos mucho esta sintaxis a lo largo del tutorial.

Problemas propuestos

- Definir una variable inmutable con el valor 50 que representa el lado de un cuadrado, en otras dos variables inmutables almacenar la superficie y el perímetro del cuadrado. Mostrar la superficie y el perímetro por la Consola.

- Definir tres variables inmutables y cargar por asignación los pesos de tres personas con valores Float. Calcular el promedio de pesos de las personas y mostrarlo.

Solución

Retornar (index.php?inicio=0)

4 - Entrada de datos por teclado en la Consola

Cuando utilizamos la Consola para mostrar información por pantalla utilizamos las funciones print y println. Si necesitamos entrar datos por teclado podemos utilizar la función readLine.

Problema 1

Realizar la carga de dos números enteros por teclado e imprimir su suma y su producto.

Proyecto5 - Principal.kt

```
fun main(argumento: Array<String>) {
    print("Ingrese primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese segundo valor:")
    val valor2 = readLine()!!.toInt()
    val suma = valor1 + valor2
    println("La suma de $valor1 y $valor2 es $suma")
    val producto = valor1 * valor2
    println("El producto de $valor1 y $valor2 es $producto")}
```

Para entrada de datos por teclado disponemos una función llamada readLine. Esta función retorna un String con los caracteres escritos por el operador hasta que presiona la tacla "Entrada". Luego llamando al método toInt de la clase String se convierten los datos ingresados por teclado en un Int y se guarda en valor1.

El problema se presenta cuando el operador presiona la tecla "Entrada" sin cargar datos, en ese caso retorna null.

Vimos que una de las premisas de Kotlin es ser conciso, la segunda premisa es que sea seguro, luego cuando una función retorna null no podemos llamar a los métodos que tiene ese objeto:

```
valor1 = readLine().toInt()
```

La línea anterior no compila ya que si readLine retorna un String es correcto llamar al método toInt para que lo convierta a entero, pero readLine puede retornar null y en ese caso no podemos llamar a toInt.

La primer forma de resolver esto es avisarle al compilador de Kotlin que confíe que la función readLine siempre retorna un String, esto lo hacemos agregando el operador !! en la llamada:

```
valor1 = readLine()!!.toInt()
```

No es la mejor forma de validar que se ingrese en forma obligatoria un dato por teclado pero en los próximos conceptos veremos como mejorar la entrada de datos por teclado para que sea más segura y evitar que se generen errores cuando el operador presiona la tecla "Entrada" sin ingresar datos.

Continuando con el problema luego de cargar los dos enteros por tecla procedemos a sumarlos, multiplicarlos y mostrar los resultados:

```
val suma = valor1 + valor2
println("La suma de $valor1 y $valor2 es $suma")
val producto = valor1 * valor2
println("El producto de $valor1 y $valor2 es $producto")
```

Problema 2

Realizar la carga del lado de un cuadrado, mostrar por pantalla el perímetro del mismo (El perímetro de un cuadrado se calcula multiplicando el valor del lado por cuatro)

Proyecto6 - Principal.kt

```
fun main(parametro: Array) {
    print("Ingrese la medida del lado del cuadrado:")
    val lado = readLine()!!.toInt()
    val perimetro = lado * 4
    println("El perímetro del cuadrado es $perimetro")
}
```

La variable lado por inferencia se define de tipo Int:

```
val lado = readLine()!!.toInt()
```

Recordemos que en forma extensa podemos escribir el código anterior con la siguiente sintaxis:

```
val lado: Int
lado = readLine()!!.toInt()
```

En Kolin recordemos que lo que se busca que el código sea lo más conciso posible.

Luego calculamos el perímetro y lo mostramos por la Consola:

```
val perimetro = lado * 4
println("El perímetro del cuadrado es $perimetro")
```

Problema 3

Se debe desarrollar un programa que pida el ingreso del precio de un artículo y la cantidad que lleva el cliente. Mostrar lo que debe abonar el comprador.

Proyecto7 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Ingrese el precio del producto:")
    val precio = readLine()!!.toDouble()
    print("Ingrese la cantidad de productos:")
    val cantidad = readLine()!!.toInt()
    val total = precio * cantidad
    println("El total a pagar es $total")
}
```

Cargamos por teclado un valor de tipo Double y por inferencia se define la variable precio con dicho tipo:

```
    print("Ingrese el precio del producto:")
    val precio = readLine()!!.toDouble()
```

Seguidamente cargamos la cantidad de productos a llevar, el dato que debe ingresar el operador es un entero:

```
    print("Ingrese la cantidad de productos:")
    val cantidad = readLine()!!.toInt()
```

Finalmente multiplicamos la variable Double y la variable Int dando como resultado otro valor Double:

```
    val total = precio * cantidad
    println("El total a pagar es $total")
```

Problemas propuestos

- Escribir un programa en el cual se ingresen cuatro números enteros, calcular e informar la suma de los dos primeros y el producto del tercero y el cuarto.

- Realizar un programa que lea por teclado cuatro valores numéricos enteros e informar su suma y promedio.

Solución

Retornar (index.php?inicio=0)

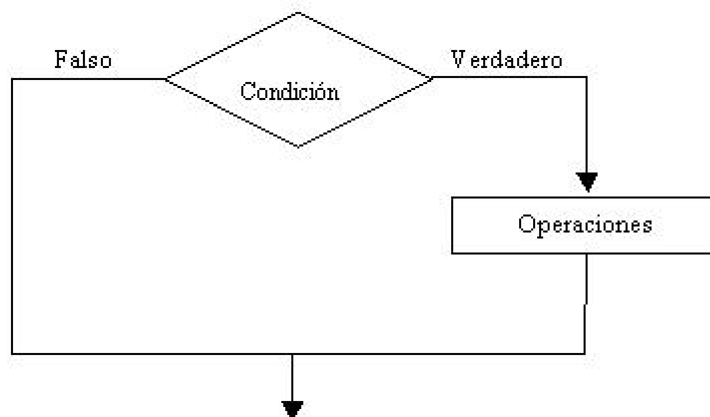
5 - Estructura condicional if

Cuando hay que tomar una decisión aparecen las estructuras condicionales. En nuestra vida diaria se nos presentan situaciones donde debemos decidir.
¿Elijo la carrera A o la carrera B?
¿Me pongo este pantalón?
Para ir al trabajo, ¿elijo el camino A o el camino B?
Al cursar una carrera, ¿elijo el turno mañana, tarde o noche?

Estructura condicional simple

Cuando se presenta la elección tenemos la opción de realizar una actividad o no realizar ninguna.

Representación gráfica:



Podemos observar: El rombo representa la condición. Hay dos opciones que se pueden tomar. Si la condición da verdadera se sigue el camino del verdadero, o sea el de la derecha, si la condición da falsa se sigue el camino de la izquierda.

Se trata de una estructura CONDICIONAL SIMPLE porque por el camino del verdadero hay actividades y por el camino del falso no hay actividades.

Por el camino del verdadero pueden existir varias operaciones, entradas y salidas, inclusive ya veremos que puede haber otras estructuras condicionales.

Problema 1

Ingresar el sueldo de una persona, si supera los 3000 pesos mostrar un mensaje en pantalla indicando que debe abonar impuestos.

Proyecto10 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Ingrese el sueldo del empleado:")
    val sueldo = readLine()!!.toDouble()
    if (sueldo > 3000) {
        println("Debe pagar impuestos")
    }
}
```

La palabra clave "if" indica que estamos en presencia de una estructura condicional; seguidamente disponemos la condición entre paréntesis. Por último encerrada entre llaves las instrucciones de la rama del verdadero.

Es necesario que las instrucciones a ejecutar en caso que la condición sea verdadera estén encerradas entre llaves { }, con ellas marcamos el comienzo y el fin del bloque del verdadero.

Pero hay situaciones donde si tenemos una sola instrucción por la rama del verdadero podemos obviar las llaves y hacer nuestro código más conciso:

```
    if (sueldo > 3000)
        println("Debe pagar impuestos")
```

En los problemas de aquí en adelante no dispondremos las llaves si tenemos una sola instrucción.

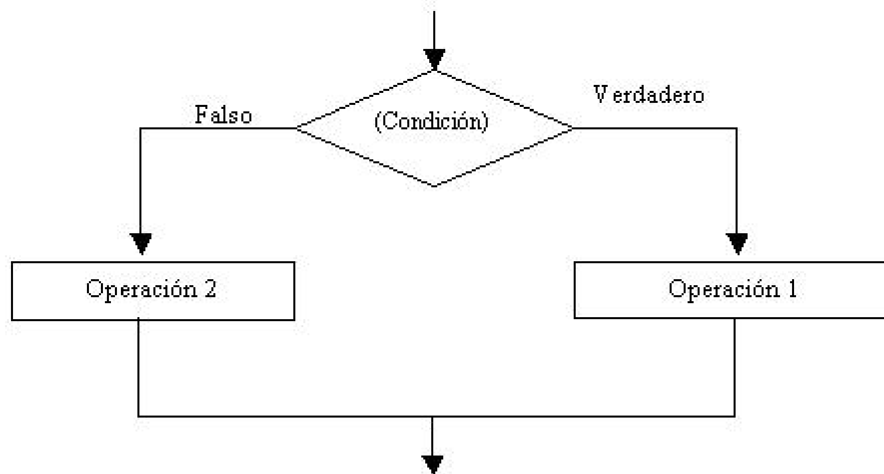
Ejecutando el programa e ingresamos un sueldo superior a 3000 pesos. Podemos observar como aparece en pantalla el mensaje "Debe pagar impuestos", ya que la condición del if es verdadera.

Volvamos a ejecutar el programa y carguemos un sueldo menor o igual a 3000 pesos. No debe aparecer mensaje en pantalla.

Estructura condicional compuesta

Cuando se presenta la elección tenemos la opción de realizar una actividad u otra. Es decir tenemos actividades por el verdadero y por el falso de la condición. Lo más importante que hay que tener en cuenta que se realizan las actividades de la rama del verdadero o las del falso, NUNCA se realizan las actividades de las dos ramas.

Representación gráfica:



En una estructura condicional compuesta tenemos entradas, salidas, operaciones, tanto por la rama del verdadero como por la rama del falso.

Problema 2

Realizar un programa que solicite ingresar dos números enteros distintos y muestre por pantalla el mayor de ellos (suponemos que el operador del programa ingresa valores distintos, no valida nuestro programa dicha situación)

Proyecto11 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Ingrese primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese segundo valor:")
    val valor2 = readLine()!!.toInt()
    if (valor1 > valor2)
        print("El mayor valor es $valor1")
    else
        print("El mayor valor es $valor2")
}
```

Se hace la entrada de valor1 y valor2 por teclado. Para saber cual variable tiene un valor mayor preguntamos si el contenido de valor1 es mayor (>) que el contenido de valor2 en un if, si la respuesta es verdadera imprimimos el contenido de valor1, en caso que la condición sea falsa se ejecuta la instrucción seguida a la palabra clave else donde mostramos el contenido de valor2:

```
    if (valor1 > valor2)
        print("El mayor valor es $valor1")
    else
        print("El mayor valor es $valor2")
```

Como podemos observar nunca se imprimen valor1 y valo2 simultáneamente.

Las llaves son opcionales porque tenemos una sola actividad por cada rama del if, en forma alternativa podemos escribir:

```
    if (valor1 > valor2) {
        print("El mayor valor es $valor1")
    }
    else {
        print("El mayor valor es $valor2")
    }
```

Operadores

En una condición deben disponerse únicamente variables, valores constantes y operadores relacionales.

Operadores Relacionales:

```
>  (mayor)
<  (menor)
>= (mayor o igual)
<= (menor o igual)
== (igual)
!= (distinto)
```

```
+ (más)
- (menos)
* (producto)
/ (división)
% (resto de una división)  Ej.:  x = 13 % 5  {se guarda 3}
```

Hay que tener en cuenta que al disponer una condición debemos seleccionar que operador relacional se adapta a la pregunta.

Ejemplos:

```
        Se ingresa un número multiplicarlo por 10 si es distinto a 0.   (!=)
        Se ingresan dos números mostrar una advertencia si son iguales.  (==)
```

Los problemas que se pueden presentar son infinitos y la correcta elección del operador sólo se alcanza con la práctica intensiva en la resolución de problemas.

Problema 3

Se ingresan por teclado 2 valores enteros. Si el primero es menor al segundo calcular la suma y la resta, luego mostrarlos, sino calcular el producto y la división.

Proyecto12 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Ingrese el primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 = readLine()!!.toInt()
    if (valor1 < valor2) {
        val suma = valor1 + valor2
        val resta = valor1 - valor2
        println("La suma de los dos valores es: $suma")
        println("La resta de los dos valores es: $resta")
    } else {
        val producto = valor1 * valor2
        val division = valor1 / valor2
        println("El producto de los dos valores es: $producto")
        println("La división de los dos valores es: $division")
    }
}
```

En este problema tenemos varias actividades por la rama del verdadero del if por lo que las llaves son obligatorias:

```
    if (valor1 < valor2) {
        val suma = valor1 + valor2
        val resta = valor1 - valor2
        println("La suma de los dos valores es: $suma")
        println("La resta de los dos valores es: $resta")
    } else {
        val producto = valor1 * valor2
        val division = valor1 / valor2
        println("El producto de los dos valores es: $producto")
        println("La división de los dos valores es: $division")
    }
```

La misma situación se produce por la rama del falso, es decir por el else debemos encerrar obligatoriamente con las llaves el bloque.

Problemas propuestos

- Se ingresan tres notas de un alumno, si el promedio es mayor o igual a siete mostrar un mensaje "Promocionado".

- Se ingresa por teclado un número entero comprendido entre 1 y 99, mostrar un mensaje indicando si el número tiene uno o dos dígitos.

(Tener en cuenta que condición debe cumplirse para tener dos dígitos, un número entero)

Solución

Retornar (index.php?inicio=0)

6 - Estructura condicional if como expresión

En Kotlin existe la posibilidad de que la estructura condicional if retorne un valor, característica no común a otros lenguajes de programación.

Veremos con una serie de ejemplos la sintaxis para utilizar el if como expresión.

Problema 1

Cargar dos valores enteros, almacenar el mayor de los mismos en otra variable y mostrarlo.

Proyecto15 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Ingrese primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese segundo valor:")
    val valor2 = readLine()!!.toInt()
    val mayor = if (valor1 > valor2) valor1 else valor2
    println("El mayor entre $valor1 y $valor2 es $mayor")
}
```

Como podemos ver asignamos a una variable llamada mayor el valor que devuelve la expresión if. Si la condición del if es verdadera retorna el contenido de la variable valor1 y sino retorna valor2:

```
    val mayor = if (valor1 > valor2) valor1 else valor2
```

El compilador de Kotlin infiere que la variable mayor debe ser de tipo Int ya que tanto valor1 como valor2 son Int.

Las llaves no las disponemos debido a que hay una solo operación tanto por el verdadero como por el falso.

Lo más común es que se utilice un if como expresión donde se retorna un valor necesitando una sola actividad por el verdadero y el falso, pero el lenguaje nos permite disponer más de una instrucción por cada rama del if.

Problema 2

Ingresar por teclado un valor entero. Almacenar en otra variable el cuadrado de dicho número si el valor ingresado es par, en caso que sea impar guardar el cubo.
Mostrar además un mensaje que indique si se calcula el cuadrado o el cubo.

Proyecto16 - Principal.kt

```
fun main(parametro: Array) {
    print("Ingrese un valor entero:")
    val valor = readLine()!!.toInt()
    val resultado = if (valor % 2 == 0) {
        print("Cuadrado:")
        valor * valor
    } else {
        print("Cubo:")
        valor * valor * valor
    }
    print(resultado)
}
```

En este problema hacemos más de una actividad por la rama del verdadero y el falso del if, por eso van encerradas entre llaves.

Es importante tener en cuenta que la última línea de cada bloque del if es la que se retorna y se almacena en la variable resultado:

```
    val resultado = if (valor % 2 == 0) {
        print("Cuadrado:")
        valor * valor
    } else {
        print("Cubo:")
        valor * valor * valor
    }
```

Utilizamos el operador % (resto de una división) para identificar si una variable es par o impar. El resto de dividir un valor por el número 2 es cero (ej. 10 % 2 es cero)

Tener en cuenta que no es lo mismo hacer:

```
    val resultado = if (valor % 2 == 0) {
        valor * valor
        print("Cuadrado:")
    } else {
```

Problema propuesto

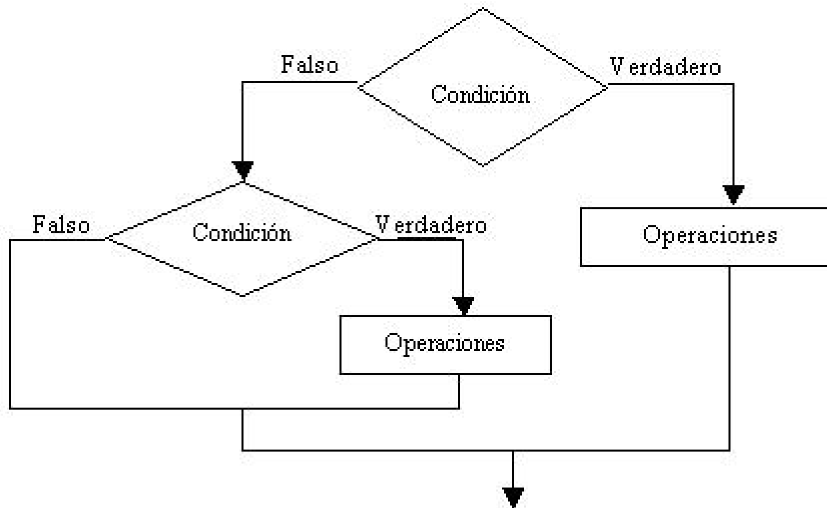
- Cargar un valor entero por teclado comprendido entre 1 y 99. Almacenar en otra variable la cantidad de dígitos que tiene el valor ingresado por teclado.
  Mostrar la cantidad de dígitos del número ingresado por teclado.

Solución

Retornar (index.php?inicio=0)

7 - Estructuras condicionales anidadas

Decimos que una estructura condicional es anidada cuando por la rama del verdadero o el falso de una estructura condicional hay otra estructura condicional.



El diagrama de flujo que se presenta contiene dos estructuras condicionales. La principal se trata de una estructura condicional compuesta y la segunda es una estructura condicional simple y está contenida por la rama del falso de la primer estructura.
Es común que se presenten estructuras condicionales anidadas aún más complejas.

Problema 1

Confeccionar un programa que pida por teclado tres notas de un alumno, calcule el promedio e imprima alguno de estos mensajes:
Si el promedio es >=7 mostrar "Promocionado".
Si el promedio es >=4 y <7 mostrar "Regular".
Si el promedio es <4 mostrar "Reprobado".

Proyecto18 - Principal.kt

```
fun main(parametros: Array<String>) {
    print("Ingrese primer nota:")
    val nota1 = readLine()!!.toInt()
    print("Ingrese segunda nota:")
    val nota2 = readLine()!!.toInt()
    print("Ingrese tercer nota:")
    val nota3 = readLine()!!.toInt()
    val promedio = (nota1 + nota2 + nota3) / 3
    if (promedio >= 7)
        print("Promocionado")
    else
        if (promedio >= 4)
            print("Regular")
        else
            print("Libre")
}
```

Primero preguntamos si el promedio es superior o igual a 7, en caso afirmativo va por la rama del verdadero del if mostrando un mensaje que indica "Promocionado" (con comillas indicamos un texto que debe imprimirse en pantalla).
En caso que la condición nos de falso, por la rama del falso aparece otra estructura condicional if, porque todavía debemos averiguar si el promedio del alumno es superior o igual a cuatro o inferior a cuatro.

Estamos en presencia de dos estructuras condicionales compuestas.

En ninguno de los bloques del verdadero y falso de los dos if hemos dispuesto llaves de apertura y cerrado debido a que hay una sola instrucción en el mismo.

Si utilizamos if como expresiones podemos codificar el mismo programa en forma más concisa con el siguiente código:

Proyecto18 - Principal.kt

```
fun main(parametros: Array<String>) {
    print("Ingrese primer nota:")
    val nota1 = readLine()!!.toInt()
    print("Ingrese segunda nota:")
    val nota2 = readLine()!!.toInt()
    print("Ingrese tercer nota:")
    val nota3 = readLine()!!.toInt()
    val promedio = (nota1 + nota2 + nota3) / 3
    val estado = if (promedio >= 7) "Promocionado" else if (promedio >= 4) "Regu
lar" else "Libre"
    print("Estado del alumno $estado")
}
```

Problemas propuestos

- Se cargan por teclado tres números distintos. Mostrar por pantalla el mayor de ellos.
- Se ingresa por teclado un valor entero, mostrar una leyenda que indique si el número es positivo, nulo o negativo.
- Confeccionar un programa que permita cargar un número entero positivo de hasta tres cifras y muestre un mensaje indicando si tiene 1, 2, o 3 cifras. Mostrar un mensaje de error si el número de cifras es mayor.
- Un postulante a un empleo, realiza un test de capacitación, se obtuvo la siguiente información: cantidad total de preguntas que se le realizaron y la cantidad de preguntas que contestó correctamente. Se pide confeccionar un programa que ingrese los dos datos por teclado e informe el nivel del mismo según el porcentaje de respuestas correctas que ha obtenido, y sabiendo que:

```
        Nivel máximo:   Porcentaje>=90%.
        Nivel medio:    Porcentaje>=75% y <90%.
        Nivel regular:  Porcentaje>=50% y <75%.
        Fuera de nivel: Porcentaje<50%.
```

Solución

Retornar (index.php?inicio=0)

8 - Condiciones compuestas con operadores lógicos

Hasta ahora hemos visto los operadores:

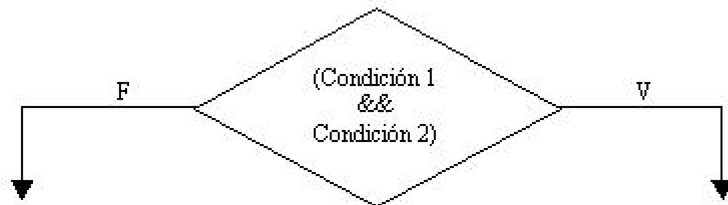
```
relacionales (>, <, >=, <= , ==, !=)
matemáticos (+, -, *, /, %)
```

pero nos están faltando otros operadores imprescindibles:

```
lógicos (&&, ||)
```

Estos dos operadores se emplean fundamentalmente en las estructuras condicionales para agrupar varias condiciones simples.

Operador &&



Traducido se lo lee como "Y". Si la Condición 1 es verdadera Y la condición 2 es verdadera luego ejecutar la rama del verdadero.
Cuando vinculamos dos o más condiciones con el operador "&&", las dos condiciones deben ser verdaderas para que el resultado de la condición compuesta de Verdadero y continúe por la rama del verdadero de la estructura condicional.

La utilización de operadores lógicos permiten en muchos casos plantear algoritmos más cortos y comprensibles.

Problema 1

Confeccionar un programa que lea por teclado tres números y nos muestre el mayor.

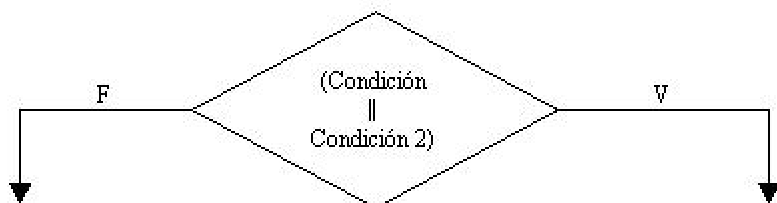
Proyecto23 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Ingrese primer valor:")
    val num1 = readLine()!!.toInt()
    print("Ingrese segundo valor:")
    val num2 = readLine()!!.toInt()
    print("Ingrese tercer valor:")
    val num3 = readLine()!!.toInt()
    if (num1 > num2 && num1 > num3)
        print(num1)
    else
        if (num2 > num3)
            print(num2)
        else
            print(num3);
}
```

Este ejercicio está resuelto sin emplear operadores lógicos en un concepto anterior del tutorial. La primera estructura condicional es una ESTRUCTURA CONDICIONAL COMPUESTA con una CONDICION COMPUESTA.

Podemos leerla de la siguiente forma:
Si el contenido de la variable num1 es mayor al contenido de la variable num2 Y si el contenido de la variable num1 es mayor al contenido de la variable num3 entonces la CONDICION COMPUESTA resulta Verdadera.
Si una de las condiciones simples da falso la CONDICION COMPUESTA da Falso y continua por la rama del falso.
Es decir que se mostrará el contenido de num1 si y sólo si num1 > num2 y num1 > num3.
En caso de ser Falsa la condición, analizamos el contenido de num2 y num3 para ver cual tiene un valor mayor.
En esta segunda estructura condicional no se requieren operadores lógicos al haber una condición simple.

Operador ||



Traducido se lo lee como "O". Si la condición 1 es Verdadera O la condición 2 es Verdadera, luego ejecutar la rama del Verdadero.
Cuando vinculamos dos o más condiciones con el operador "Or", con que una de las dos

condiciones sea Verdadera alcanza para que el resultado de la condición compuesta sea Verdadero.

Problema 2

Se carga una fecha (día, mes y año) por teclado. Mostrar un mensaje si corresponde al primer trimestre del año (enero, febrero o marzo)
Cargar por teclado el valor numérico del día, mes y año. Ejemplo: dia:10 mes:2 año:2017.

Proyecto24 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Ingrese día:")
    val dia = readLine()!!.toInt()
    print("Ingrese mes:")
    val mes = readLine()!!.toInt()
    print("Ingrese Año:")
    val año = readLine()!!.toInt()
    if (mes == 1 || mes == 2 || mes == 3)
        print("Corresponde al primer trimestre");
}
```

La carga de una fecha se hace por partes, ingresamos las variables dia, mes y año.
Mostramos el mensaje "Corresponde al primer trimestre" en caso que el mes ingresado por teclado sea igual a 1, 2 ó 3.

En la condición no participan las variables dia y año.

Problemas propuestos

- Realizar un programa que pida cargar una fecha cualquiera, luego verificar si dicha fecha corresponde a Navidad.
- Se ingresan tres valores por teclado, si todos son iguales calcular el cubo del número y mostrarlo.
- Se ingresan por teclado tres números, si todos los valores ingresados son menores a 10, imprimir en pantalla la leyenda "Todos los números son menores a diez".
- Se ingresan por teclado tres números, si al menos uno de los valores ingresados es menor a 10, imprimir en pantalla la leyenda "Alguno de los números es menor a diez".
- Escribir un programa que pida ingresar la coordenada de un punto en el plano, es decir dos valores enteros x e y (distintos a cero).
  Posteriormente imprimir en pantalla en que cuadrante se ubica dicho punto. (1º Cuadrante si x > 0 Y y > 0 , 2º Cuadrante: x < 0 Y y > 0, etc.)

- Escribir un programa en el cual: dada una lista de tres valores enteros ingresados por teclado se guarde en otras dos variables el menor y el mayor de esa lista. Utilizar el if como expresión para obtener el mayor y el menor.
  Imprimir luego las dos variables.

Solución

Retornar (index.php?inicio=0)

9 - Estructura repetitiva while

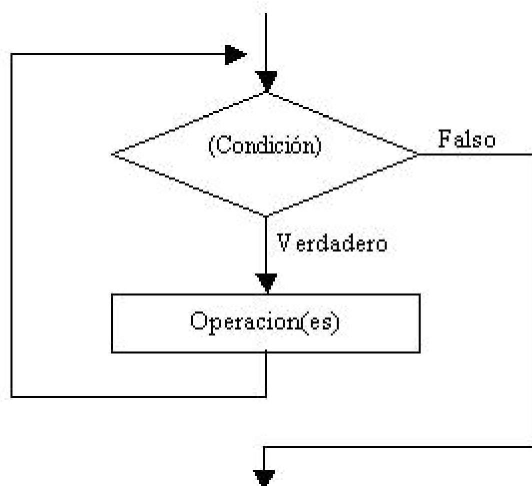
Hasta ahora hemos empleado estructuras SECUENCIALES y CONDICIONALES. Existe otro tipo de estructuras tan importantes como las anteriores que son las estructuras REPETITIVAS.

Una estructura repetitiva permite ejecutar una instrucción o un conjunto de instrucciones varias veces.

Una ejecución repetitiva de sentencias se caracteriza por:
- La o las sentencias que se repiten.
- El test o prueba de condición antes de cada repetición, que motivará que se repitan o no las sentencias.

Estructura repetitiva while.

Representación gráfica de la estructura while:



No debemos confundir la representación gráfica de la estructura repetitiva while (Mientras) con la estructura condicional if (Si)

Funcionamiento: En primer lugar se verifica la condición, si la misma resulta verdadera se ejecutan las operaciones que indicamos por la rama del Verdadero.
A la rama del verdadero la graficamos en la parte inferior de la condición. Una línea al final del bloque de repetición la conecta con la parte superior de la estructura repetitiva.
En caso que la condición sea Falsa continúa por la rama del Falso y sale de la estructura repetitiva para continuar con la ejecución del algoritmo.

El bloque se repite MIENTRAS la condición sea Verdadera.

Importante: Si la condición siempre retorna verdadero estamos en presencia de un ciclo repetitivo infinito. Dicha situación es un error de programación, nunca finalizará el programa.

Problema 1

Realizar un programa que imprima en pantalla los números del 1 al 100.

Proyecto31 - Principal.kt

```
fun main(parametro: Array<String>) {
    var x = 1
    while (x <= 100) {
        println(x)
        x = x + 1
    }
}
```

Lo primero que podemos hacer notar que definimos una variable mutable de tipo Int con el valor 1:

```
    var x = 1
```

Esta variable x se irá modificando dentro del while por eso la necesidad de definirla con la palabra clave var.

La estructura repetitiva while siempre tiene una condición, si se verifica verdadera pasa a ejecutar el bloque de instrucciones encerradas entre llaves (si no hay llaves solo ejecuta la siguiente instrucción del while a modo similar del if)

Cada vez que se ejecuta el bloque de instrucciones del while vuelve a verificarse la condición para ver si repite nuevamente o corta el ciclo repetitivo.

En nuestro problema x comienza con el valor 1, al ejecutarse la condición retorna VERDADERO porque el contenido de x (1) es menor o igual a 100. Al ser la condición verdadera se ejecuta el bloque de instrucciones que contiene la estructura while. El bloque de instrucciones contiene una salida y una operación.

Se imprime el contenido de x, y seguidamente se incrementa la variable x en uno.

La operación x = x + 1 se lee como "en la variable x se guarda el contenido de x más 1". Es decir, si x contiene 1 luego de ejecutarse esta operación se almacenará en x un 2.

Al finalizar el bloque de instrucciones que contiene la estructura repetitiva se verifica nuevamente la condición de la estructura repetitiva y se repite el proceso explicado anteriormente.

Mientras la condición retorne verdadero se ejecuta el bloque de instrucciones; al retornar falso la verificación de la condición se sale de la estructura repetitiva y continua el algoritmo, en este caso finaliza el programa.

Lo más difícil es la definición de la condición de la estructura while y qué bloque de instrucciones se van a repetir. Observar que si, por ejemplo, disponemos la condición x >= 100 (si x es mayor o igual a 100) no provoca ningún error sintáctico pero estamos en presencia de un error lógico porque al evaluarse por primera vez la condición retorna falso y no se ejecuta el bloque de instrucciones que queríamos repetir 100 veces.

No existe una RECETA para definir una condición de una estructura repetitiva, sino que se logra con una práctica continua solucionando problemas.

Problema 2

Escribir un programa que solicite la carga de un valor positivo y nos muestre desde 1 hasta el valor ingresado de uno en uno.
Ejemplo: Si ingresamos 30 se debe mostrar en pantalla los números del 1 al 30.

Proyecto32 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Ingrese un valor:")
    val valor = readLine()!!.toInt()
    var x = 1
    while (x <= valor) {
        println(x)
        x = x + 1
    }
}
```

La variable x recibe el nombre de CONTADOR. Un contador es un tipo especial de variable que se incrementa o disminuye con valores constantes durante la ejecución del programa.

El contador x nos indica en cada momento la cantidad de valores impresos en pantalla.

Problema 3

Desarrollar un programa que permita la carga de 10 valores por teclado y nos muestre posteriormente la suma de los valores ingresados y su promedio.

Proyecto33 - Principal.kt

```
fun main(parametro: Array<String>) {
    var x = 1
    var suma = 0
    while (x <= 10) {
        print("Ingrese un valor:")
        val valor=readLine()!!.toInt()
        suma = suma + valor
        x = x + 1
    }
    println("La suma de los 10 valores ingresados es $suma")
    val promedio = suma / 10
    println("El promedio es $promedio")
}
```

En este problema, a semejanza de los anteriores, llevamos un CONTADOR llamado x que nos sirve para contar las vueltas que debe repetir el while.
También aparece el concepto de ACUMULADOR (un acumulador es un tipo especial de variable que se incrementa o disminuye con valores variables durante la ejecución del programa)

Hemos dado el nombre de suma a nuestro acumulador. Cada ciclo que se repita la estructura repetitiva, la variable suma se incrementa con el contenido ingresado en la variable valor.

Problema 4

Una planta que fabrica perfiles de hierro posee un lote de n piezas.
Confeccionar un programa que pida ingresar por teclado la cantidad de piezas a procesar y luego ingrese la longitud de cada perfil; sabiendo que la pieza cuya longitud esté comprendida en el rango de 1.20 y 1.30 son aptas. Imprimir por pantalla la cantidad de piezas aptas que hay en el lote.

Proyecto34 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Cuantas piezas procesará:")
    val n = readLine()!!.toInt()
    var x = 1
    var cantidad = 0
    while (x <= n) {
        print("Ingrese la medida de la pieza:")
        val largo = readLine()!!.toDouble()
        if (largo >= 1.20 && largo <= 1.30)
            cantidad = cantidad +1
        x = x + 1;
    }
    print("La cantidad de piezas aptas son: $cantidad")
}
```

Podemos observar que dentro de una estructura repetitiva puede haber estructuras condicionales (inclusive puede haber otras estructuras repetitivas que veremos más adelante)

En este problema hay que cargar inicialmente la cantidad de piezas a ingresar (n), seguidamente se cargan n valores de largos de piezas dentro del while.

Cada vez que ingresamos un largo de pieza (largo) verificamos si es una medida correcta (debe estar entre 1.20 y 1.30 el largo para que sea correcta), en caso de ser correcta la CONTAMOS (incrementamos la variable cantidad en 1)

Al contador cantidad lo inicializamos en cero porque inicialmente no se ha cargado ningún largo de medida.

Cuando salimos de la estructura repetitiva porque se han cargado n largos de piezas mostramos por pantalla el contador cantidad (que representa la cantidad de piezas aptas)

En este problema tenemos dos CONTADORES:

```
x                (Cuenta la cantidad de piezas cargadas hasta el momento)
cantidad         (Cuenta los perfiles de hierro aptos)
```

Problemas propuestos

- Escribir un programa que solicite ingresar 10 notas de alumnos y nos informe cuántos tienen notas mayores o iguales a 7 y cuántos menores.

- Se ingresan un conjunto de n alturas de personas por teclado (n se ingresa por teclado). Mostrar la altura promedio de las personas.

- En una empresa trabajan n empleados cuyos sueldos oscilan entre $100 y $500, realizar un programa que lea los sueldos que cobra cada empleado e informe cuántos empleados cobran entre $100 y $300 y cuántos cobran más de $300. Además el programa deberá informar el importe que gasta la empresa en sueldos al personal.

- Realizar un programa que imprima 25 términos de la serie 11 - 22 - 33 - 44, etc. (No se ingresan valores por teclado)

- Mostrar los múltiplos de 8 hasta el valor 500. Debe aparecer en pantalla 8 - 16 - 24, etc.

- Realizar un programa que permita cargar dos listas de 5 valores cada una. Informar con un mensaje cual de las dos listas tiene un valor acumulado mayor (mensajes "Lista 1 mayor", "Lista 2 mayor", "Listas iguales")
  Tener en cuenta que puede haber dos o más estructuras repetitivas en un algoritmo.

- Desarrollar un programa que permita cargar n números enteros y luego nos informe cuántos valores fueron pares y cuántos impares.
  Emplear el operador "%" en la condición de la estructura condicional:

  ```
          if (valor % 2 == 0)         //Si el if se verifica verdadero luego
  es par.
  ```

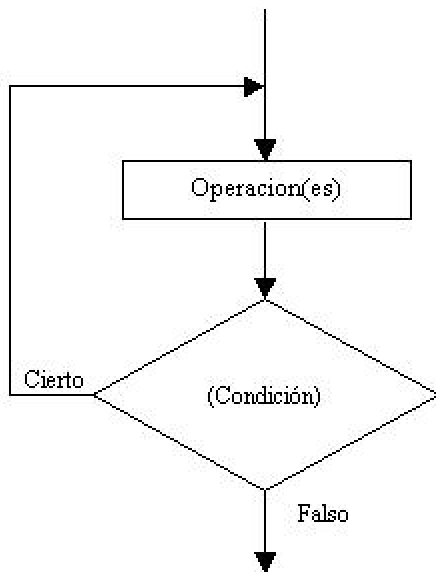
Solución

Retornar (index.php?inicio=0)

10 - Estructura repetitiva do/while

La estructura do/while es otra estructura repetitiva, la cual ejecuta al menos una vez su bloque repetitivo, a diferencia del while que podría no ejecutar el bloque.
Esta estructura repetitiva se utiliza cuando conocemos de antemano que por lo menos una vez se ejecutará el bloque repetitivo.
La condición de la estructura está abajo del bloque a repetir, a diferencia del while que está en la parte superior.

Representación gráfica:



El bloque de operaciones se repite MIENTRAS que la condición sea Verdadera.
Si la condición retorna Falso el ciclo se detiene.

Es importante analizar y ver que las operaciones se ejecutan como mínimo una vez.

Problema 1

Escribir un programa que solicite la carga de un número entre 0 y 999, y nos muestre un mensaje de cuántos dígitos tiene el mismo. Finalizar el programa cuando se cargue el valor 0.

Proyecto42 - Principal.kt

```
fun main(parametro: Array<String>) {
    do {
        print("Ingrese un valor comprendido entre 0 y 999:")
        val valor = readLine()!!.toInt()
        if (valor < 10)
            println("El valor ingresado tiene un dígito")
        else
            if (valor < 100)
                println("El valor ingresado tiene dos dígitos")
            else
                println("El valor ingresado tiene tres dígitos")
    } while (valor != 0)
}
```

En este problema por lo menos se carga un valor. Si se carga un valor menor a 10 se trata de un número de una cifra, si es menor a 100 se trata de un valor de dos dígitos, en caso contrario se trata de un valor de tres dígitos (no se controla que el operador cargue valores negativos o valores con más de tres dígitos). Este bloque se repite hasta que se ingresa en la variable valor el número 0 con lo que la condición de la estructura do while retorna falso y sale del bloque repetitivo finalizando el programa.

Problema 2

Escribir un programa que solicite la carga de números por teclado, obtener su promedio. Finalizar la carga de valores cuando se cargue el valor 0.

Cuando la finalización depende de algún valor ingresado por el operador conviene el empleo de la estructura do/while, por lo menos se cargará un valor (en el caso más extremo se carga 0, que indica la finalización de la carga de valores)

Proyecto43 - Principal.kt

```
fun main(parametro: Array<String>) {
    var cant = 0
    var suma = 0
    do {
        print("Ingrese un valor (0 para finalizar):")
        val valor = readLine()!!.toInt()
        if (valor != 0) {
            suma += valor
            cant++
        }
    } while (valor !=0)
    if (cant != 0) {
        val promedio = suma / cant
        print("El promedio de los valores ingresados es: $promedio")
    } else
        print("No se ingresaron valores.")
}
```

Acotaciones

Lo más común para incrementar en uno una variable Int es utilizar el operador ++:

```
cant++
```

Es lo mismo que escribir:

```
cant = cant + 1
```

También existe el operador -- que disminuye en uno la variable.

También en Kotlin se utiliza el operador += para la acumulación:

```
suma += valor
```

En lugar de escribir:

```
suma = suma + valor
```

En el problema 2 cada vez que se ingresa mediante un if verificamos si es distinto a cero para acumularlo y contarlo:

```
if (valor != 0) {
    suma += valor
    cant++
}
```

El ciclo do/while se repite mientras se ingrese un valor distinto a cero, cuando ingresamos el cero finaliza el ciclo y mediante un if verificamos si se cargó al menos un valor de información para obtener el promedio:

```
    if (cant != 0) {
        val promedio = suma / cant
        print("El promedio de los valores ingresados es: $promedio")
    } else
        print("No se ingresaron valores.")
```

Problema 3

Realizar un programa que permita ingresar el peso (en kilogramos) de piezas. El proceso termina cuando ingresamos el valor 0.
Se debe informar:

a) Cuántas piezas tienen un peso entre 9.8 Kg. y 10.2 Kg.?, cuántas con más de 10.2 Kg.? y cuántas con menos de 9.8 Kg.?

b) La cantidad total de piezas procesadas.

Proyecto44 - Principal.kt

```
fun main(parametro: Array<String>) {
    var cant1 = 0
    var cant2 = 0
    var cant3 = 0
    do {
        print("Ingrese el peso de la pieza (0 pera finalizar):")
        val peso = readLine()!!.toDouble()
        if (peso > 10.2)
            cant1++
        else
            if (peso >= 9.8)
                cant2++
            else
                if (peso > 0)
                    cant3++
    } while(peso != 0.0)
    println("Piezas aptas: $cant2")
    println("Piezas con un peso superior a 10.2: $cant1")
    println("Piezas con un peso inferior a 9.8: $cant3");
    val suma = cant1 + cant2 + cant3
    println("Cantidad total de piezas procesadas: $suma")
}
```

Los tres contadores cont1, cont2, y cont3 se inicializan en 0 antes de entrar a la estructura repetitiva.

A la variable suma no se la inicializa en 0 porque no es un acumulador, sino que guarda la suma del contenido de las variables cont1, cont2 y cont3.

La estructura se repite mientras no se ingrese el valor 0 en la variable peso. Este valor no se lo considera un peso menor a 9.8 Kg., sino que indica que ha finalizado la carga de valores por teclado.

Debemos en la condición del do/while comparar el contenido de la variable peso que es de tipo Double con el valor 0.0 (de esta forma indicamos que el valor es Double, no podemos poner solo 0)

Problemas propuestos

- Realizar un programa que acumule (sume) valores ingresados por teclado hasta ingresar el 9999 (no sumar dicho valor, indica que ha finalizado la carga). Imprimir el valor acumulado e informar si dicho valor es cero, mayor a cero o menor a cero.

- En un banco se procesan datos de las cuentas corrientes de sus clientes. De cada cuenta corriente se conoce: número de cuenta y saldo actual. El ingreso de datos debe finalizar al ingresar un valor negativo en el número de cuenta.
  Se pide confeccionar un programa que lea los datos de las cuentas corrientes e informe:
  a)De cada cuenta: número de cuenta y estado de la cuenta según su saldo, sabiendo que:

  ```
  Estado de la cuenta     'Acreedor' si el saldo es >0.
                          'Deudor' si el saldo es <0.
                          'Nulo' si el saldo es =0.
  ```

  b) La suma total de los saldos acreedores.

Solución

Retornar (index.php?inicio=0)

11 - Estructura repetitiva for y expresiones de rango

La estructura for tiene algunas variantes en Kotlin, en este concepto veremos la estructura for con expresiones de rango.

Veamos primero como se define y crea un rango.

Un rango define un intervalo que tiene un valor inicial y un valor final, se lo define utilizando el operador ..

Ejemplos de definición de rangos:

```
val unDigito = 1..9
val docena = 1..12
var letras = "a".."z"
```

Si necesitamos conocer si un valor se encuentra dentro de un rango debemos emplear el operador in o el !in:

```
    val docena = 1..12

    if (5 in docena)
        println("el 5 está en el rango docena")

    if (18 !in docena)
        println("el 18 no está en el rango docena")
```

Los dos if se verifican como verdadero.

Veamos ahora la estructura repetitiva for empleando un rango para repetir un bloque de comandos.

Problema 1

Realizar un programa que imprima en pantalla los números del 1 al 100.

Proyecto47 - Principal.kt

```
fun main(parametro: Array) {
    for(i in 1..100)
        println(i)
}
```

La variable i se define de tipo Int por inferencia ya que el rango es de 1..100

En la primer ejecución del ciclo repetitivo i almacena el valor inicial del rango es decir el 1. Luego de ejecutar el bloque la variable i toma el valor 2 y así sucesivamente.

Problema 2

Desarrollar un programa que permita la carga de 10 valores por teclado y nos muestre posteriormente la suma de los valores ingresados y su promedio. Este problema ya lo desarrollamos empleando el while, lo resolveremos empleando la estructura repetitiva for.

Proyecto48 - Principal.kt

```
fun main(parametro: Array<String>) {
    var suma = 0
    for(i in 1..10) {
        print("Ingrese un valor:")
        val valor = readLine()!!.toInt()
        suma += valor
    }
    println("La suma de los valores ingresados es $suma")
    val promedio = suma / 10
    println("Su promedio es $promedio")
}
```

Como podemos ver la variable i dentro del ciclo for no se la utiliza dentro del bloque repetitivo y solo nos sirve para que el bloque contenido en el for se repita 10 veces.

Cuando sabemos cuantas veces se debe repetir un bloque de instrucciones es más conveniente utilizar el for que un while donde nosotros debemos definir, inicializar e incrementar un contador.

Problema 3

Escribir un programa que lea 10 notas de alumnos y nos informe cuántos tienen notas mayores o iguales a 7 y cuántos menores.

Proyecto49 - Principal.kt

```
fun main(parametro: Array<String>) {
    var aprobados = 0
    var reprobados = 0
    for(i in 1..10) {
        print("Ingrese nota:")
        val nota = readLine()!!.toInt()
        if (nota >= 7)
            aprobados++
        else
            reprobados++
    }
    println("Cantidad de alumnos con notas mayores o iguales a 7: $aprobados")
    println("Cantidad de alumnos con notas menores a 7: $reprobados")
}
```

Nuevamente como necesitamos cargar 10 valores por teclado disponemos un for:

```
    for(i in 1..10) {
```

Problema 4

Desarrollar un programa que cuente cuantos múltiplos de 3, 5 y 9 hay en el rango de 1 a 10000 (No se deben cargar valores por teclado)

Proyecto50 - Principal.kt

```
fun main(parametro: Array<String>) {
    var mult3 = 0
    var mult5 = 0
    var mult9 = 0
    for(i in 1..10000) {
        if (i % 3 == 0)
            mult3++
        if (i % 5 == 0)
            mult5++
        if (i % 8 == 0)
            mult9++
    }
    println("Cantidad de múltiplos de 3: $mult3")
    println("Cantidad de múltiplos de 5: $mult5")
    println("Cantidad de múltiplos de 9: $mult9")
}
```

En este problema recordemos que el contador i toma la primer vuelta el valor 1, luego el valor 2 y así sucesivamente hasta el valor 10000.

Si queremos averiguar si un valor es múltiplo de 3 obtenemos el resto de dividirlo por 3 y si dicho resultado es cero luego podemos inferir que el número es múltiplo de 3.

Problema 5

Escribir un programa que lea n números enteros y calcule la cantidad de valores pares ingresados.

Este tipo de problemas también se puede resolver empleando la estructura repetitiva for ya que cuando expresamos el rango podemos disponer un nombre de variable.

Proyecto51 - Principal.kt

```
fun main(parametros: Array<String>) {
    var cant = 0
    print("Cuantos valores ingresará para analizar:")
    val cantidad = readLine()!!.toInt()
    for(i in 1..cantidad) {
        print("Ingrese valor:")
        val valor = readLine()!!.toInt()
        if (valor % 2 ==0)
            cant++
    }
    println("Cantidad de pares: $cant")
}
```

Como vemos en la estructura repetitiva for el valor final del rango dispusimos la variable cantidad en lugar de un valor fijo:

```
    for(i in 1..cantidad) {
```

Hasta que no se ejecuta el programa no podemos saber cuantas veces se repetirá el for. La cantidad de repeticiones dependerá del número que cargue el operador.

Variantes del for

Si necesitamos que la variable del for no reciba todos los valores comprendidos en el rango sino que avance de 2 en 2 podemos utilizar la siguiente sintaxis:

```
    for(i in 1..10 step 2)
        println(i)
```

Se imprimen los valores 1, 3, 5, 7, 9

Si necesitamos que la variable tome el valor 10, luego el 9 y así sucesivamente, es decir en forma inversa debemos utilizar la siguiente sintaxis:

```
    for(i in 10 downTo 1)
        println(i)
```

Se imprimen los valores 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

También podemos utilizar el step con el downTo:

```
    for(i in 10 downTo 1 step 2)
        println(i)
```

Se imprimen los valores 10, 8, 6, 4, 2

Problemas propuestos

- Confeccionar un programa que lea n pares de datos, cada par de datos corresponde a la medida de la base y la altura de un triángulo. El programa deberá informar:
  a) De cada triángulo la medida de su base, su altura y su superficie (la superficie se calcula multiplicando la base por la altura y dividiendo por dos).
  b) La cantidad de triángulos cuya superficie es mayor a 12.

- Desarrollar un programa que solicite la carga de 10 números e imprima la suma de los últimos 5 valores ingresados.

- Desarrollar un programa que muestre la tabla de multiplicar del 5 (del 5 al 50)

- Confeccionar un programa que permita ingresar un valor del 1 al 10 y nos muestre la tabla de multiplicar del mismo (los primeros 12 términos)
  Ejemplo: Si ingresó 3 deberá aparecer en pantalla los valores 3, 6, 9, hasta el 36.

- Realizar un programa que lea los lados de n triángulos, e informar:
  a) De cada uno de ellos, qué tipo de triángulo es: equilátero (tres lados iguales), isósceles (dos lados iguales), o escaleno (ningún lado igual)
  b) Cantidad de triángulos de cada tipo.

- Escribir un programa que pida ingresar coordenadas (x,y) que representan puntos en el plano.
  Informar cuántos puntos se han ingresado en el primer, segundo, tercer y cuarto cuadrante. Al comenzar el programa se pide que se ingrese la cantidad de puntos a procesar.

- Se realiza la carga de 10 valores enteros por teclado. Se desea conocer:
  a) La cantidad de valores ingresados negativos.
  b) La cantidad de valores ingresados positivos.
  c) La cantidad de múltiplos de 15.
  d) El valor acumulado de los números ingresados que son pares.

Solución

Retornar (index.php?inicio=0)

12 - Estructura condicional when

Además de la estructura condicional if Kotlin nos proporciona una estructura condicional para situaciones que tenemos que verificar múltiples condiciones que se resuelven con if anidados.

Mediante una serie de ejemplos veremos la sintaxis de la estructura when.

Problema 1

Escribir un programa que pida ingresar la coordenada de un punto en el plano, es decir dos valores enteros x e y.
Posteriormente imprimir en pantalla en que cuadrante se ubica dicho punto. (1º Cuadrante si x > 0 Y y > 0 , 2º Cuadrante: x < 0 Y y > 0, 3º Cuadrante: x < 0 Y y < 0, 4º Cuadrante: x > 0 Y y < 0)
Si alguno o los dos valores son cero luego el punto se encuentra en un eje.

Proyecto59 - Principal.kt

```
fun main(parametro: Array<String>) {
    print("Ingrese coordenada x del punto:")
    val x = readLine()!!.toInt()
    print("Ingrese coordenada y del punto:")
    val y = readLine()!!.toInt()
    when {
        x > 0 && y > 0 -> println("Primer cuadrate")
        x < 0 && y > 0 -> println("Segundo cuadrante")
        x < 0 && y < 0 -> println("Tercer cuadrante")
        x > 0 && y < 0 -> println("Cuarto cuadrante")
        else -> println("El punto se encuentra en un eje")
    }
}
```

Disponemos la palabra clave when y entre llaves las distintas condiciones y luego del operador -> la o las instrucciones a ejecutar si se cumple la condición.:

```
    when {
        x > 0 && y > 0 -> println("Primer cuadrate")
        x < 0 && y > 0 -> println("Segundo cuadrante")
        x < 0 && y < 0 -> println("Tercer cuadrante")
        x > 0 && y < 0 -> println("Cuarto cuadrante")
        else -> println("El punto se encuentra en un eje")
    }
```

Si alguna de las condiciones se verifica verdadera no se analizan las siguientes.

Si ninguna de las cuatro condiciones dispuestas en el when se verifica verdadera se ejecutan las instrucciones que disponemos luego del else.

Podemos comparar el mismo mismo problema resuelto con if anidados y ver que queda más conciso y claro con when:

```
    if (x > 0 && y > 0)
        print("Se encuentra en el primer cuadrante")
    else
        if (x < 0 && y > 0)
            print("Se encuentra en el segundo cuadrante")
        else
            if (x < 0 && y < 0)
                print("Se encuentra en el tercer cuadrante")
            else
                if (x > 0 && y < 0)
                    print("Se encuentra en el cuarto cuadrante")
                else
                    print("Se encuentra en un eje")
```

Problema 2

Confeccionar un programa que pida por teclado tres notas de un alumno, calcule el promedio e imprima alguno de estos mensajes:
Si el promedio es >=7 mostrar "Promocionado".
Si el promedio es >=4 y <7 mostrar "Regular".
Si el promedio es <4 mostrar "Reprobado".

Proyecto60 - Principal.kt

```
fun main(parametros: Array<String>) {
    print("Ingrese primer nota:")
    val nota1 = readLine()!!.toInt()
    print("Ingrese segunda nota:")
    val nota2 = readLine()!!.toInt()
    print("Ingrese tercer nota:")
    val nota3 = readLine()!!.toInt()
    val promedio = (nota1 + nota2 + nota3) / 3
    when {
        promedio >= 7 -> print("Promocionado")
        promedio >= 4 -> print("Regular")
        else -> print("Libre")
    }
}
```

Problema 3

Realizar un programa que permita ingresar el peso (en kilogramos) de piezas. El proceso termina cuando ingresamos el valor 0.
Se debe informar:

a) Cuántas piezas tienen un peso entre 9.8 Kg. y 10.2 Kg.?, cuántas con más de 10.2 Kg.? y cuántas con menos de 9.8 Kg.?

b) La cantidad total de piezas procesadas.

Proyecto61 - Principal.kt

```
fun main(parametro: Array<String>) {
    var cant1 = 0
    var cant2 = 0
    var cant3 = 0
    do {
        print("Ingrese el peso de la pieza (0 pera finalizar):")
        val peso = readLine()!!.toDouble()
        when {
            peso > 10.2 -> cant1++
            peso >= 9.8 -> cant2++
            peso > 0 -> cant3++
        }
    } while(peso != 0.0)
    println("Piezas aptas: $cant2")
    println("Piezas con un peso superior a 10.2: $cant1")
    println("Piezas con un peso inferior a 9.8: $cant3");
    val suma = cant1 + cant2 + cant3
    println("Cantidad total de piezas procesadas: $suma")
}
```

La sección del else es opcional como lo podemos comprobar en este problema.

Estructura when como expresión

Vimos que en Kotlin existe la posibilidad de que la estructura condicional if retorne un valor, la misma posibilidad se presenta con la estructura when.

Problema 4

Ingresar los sueldos de 10 empleados por teclado. Mostrar un mensaje según el valor del sueldo:

```
"sueldo alto" si es > 5000
"sueldo medio" si es <=5000 y > 2000
"sueldo bajo" si es <= 2000
```

Además mostrar el total acumulado de gastos en sueldos altos.

Proyecto62 - Principal.kt

```
fun main(parametro: Array<String>) {
    var total = 0
    for(i in 1..10) {
        print("ingrese sueldo del operario:")
        val sueldo = readLine()!!.toInt()
        total += when {
            sueldo > 5000 -> {
                println("Sueldo alto")
                sueldo
            }
            sueldo > 2000 -> {
                println("Sueldo medio")
                0
            }
            else -> {
                println("Sueldo bajo")
                0
            }
        }
    }
    println("Gastos totales en sueldos altos: $total")
}
```

La estructura when retorna un valor entero que acumulamos en la variable total. Si entra por la primera condición del when mostramos por pantalla el mensaje "Sueldo alto" y retornamos el valor del sueldo.

Como solo debemos acumular los sueldos altos cuando es un sueldo medio o bajo retornamos el valor cero que no afecta en la acumulación.

Tengamos en cuenta que cuando tenemos dos o más instrucciones luego del operador -> debemos disponer las llaves de apertura y cerrado.

Problemas propuestos

- Se ingresa por teclado un valor entero, mostrar una leyenda por pantalla que indique si el número es positivo, nulo o negativo.

- Plantear una estructura que se repita 5 veces y dentro de la misma cargar 3 valores enteros. Acumular solo el mayor del cada lista de tres valores.

- Realizar un programa que lea los lados de n triángulos, e informar:
  a) De cada uno de ellos, qué tipo de triángulo es: equilátero (tres lados iguales), isósceles (dos lados iguales), o escaleno (ningún lado igual)
  b) Cantidad de triángulos de cada tipo.

Solución

Retornar (index.php?inicio=0)

13 - Estructura condicional when con argumento

Tenemos una segunda forma de utilizar la sentencia when en el lenguaje Kotlin pasando un argumento inmediatamente después de la palabra clave when.

Problema 1

Ingresar un valor entero comprendido entre 1 y 5. Mostrar el mismo en castellano.

Proyecto66 - Principal.kt

```
fun main(parameto: Array<String>) {
    print("Ingrese un valor entero entre 1 y 5:")
    val valor = readLine()!!.toInt()
    when (valor) {
        1 -> print("uno")
        2 -> print("dos")
        3 -> print("tres")
        4 -> print("cuatro")
        5 -> print("cinco")
        else -> print("valor fuera de rango")
    }
}
```

Como vemos disponemos luego de la palabra clave when entre paréntesis una variable. Se verifica el contenido de la variable "valor" con cada uno de los datos indicados.
Por ejemplo si cargamos por teclado el 3 luego son falsos los dos primeros caminos:

```
        1 -> print("uno")
        2 -> print("dos")
```

Pero el tercer camino se verifica verdadero y pasa a ejecutar los comandos dispuestos después del operador ->

```
        3 -> print("tres")
```

Problema 2

Ingresar un valor entero positivo comprendido entre 1 y 10000. Imprimir un mensaje indicando cuantos dígitos tiene.

Proyecto67 - Principal.kt

```
fun main(parametro: Array<String>){
    print("Ingrese un valor entero positivo comprendido entre 1 y 99999:")
    val valor = readLine()!!.toInt()
    when (valor){
        in 1..9 -> print("Tiene 1 dígito")
        in 10..99 -> print("Tiene 2 dígitos")
        in 100..999 -> print("Tiene 3 dígitos")
        in 1000..9999 -> print("Tiene 4 dígitos")
        in 10000..99999 -> print("Tiene 5 dígitos")
        else -> print("No se encuentra comprendido en el rango indicado")
    }
}
```

También en Kotlin podemos comprobar si una variable se encuentra comprendida en un rango determinado utilizando la palabra in y el rango respectivo.

Problema 3

Ingresar 10 valores enteros por teclado. Contar cuantos de dichos valores ingresados fueron cero y cuantos 1,5 o 10.

Proyecto68 - Principal.kt

```
fun main(parametro: Array<String>){
    var cant1 = 0
    var cant2 = 0
    for(i in 1..10) {
        print("Ingrese un valor entero:")
        val valor = readLine()!!.toInt()
        when (valor){
            0 -> cant1++
            1, 5, 10 -> cant2++
        }
    }
    println("Cantidad de números 0 ingresados: $cant1")
    println("Cantidad de números 1,2 o 3 ingresados: $cant2")
}
```

En este problema disponemos en uno de los caminos del when una lista de valores separados por coma, si alguno de ellos coincide con la variable "valor" luego se incrementa "cant2":

```
            1, 5, 10 -> cant2++
```

Problema propuesto

- Realizar la carga de la cantidad de hijos de 10 familias. Contar cuantos tienen 0,1,2 o más hijos. Imprimir dichos contadores.

Solución

Retornar (index.php?inicio=0)

14 - Concepto de funciones

Hasta ahora hemos trabajado resolviendo todo el problema en la función main propuesta en Kotlin.

Esta forma de organizar un programa solo puede ser llevado a cabo si el mismo es muy pequeño (decenas de líneas)

Ahora buscaremos dividir o descomponer un problema complejo en pequeños problemas. La solución de cada uno de esos pequeños problemas nos trae la solución del problema complejo.

En Kotlin el planteo de esas pequeñas soluciones al problema complejo se hace dividiendo el programa en funciones.
Una función es un conjunto de instrucciones en Kotlin que resuelven un problema específico.

Veamos ahora como crear nuestras propias funciones.

Los primeros problemas que presentaremos nos puede parecer que sea más conveniente resolver todo en la función main en vez de dividirlo en pequeña funciones. A medida que avancemos veremos que si un programa empieza a ser más complejo (cientos de líneas, miles de líneas o más) la división en pequeñas funciones nos permitirá tener un programa más ordenado y fácil de entender y por lo tanto en mantener.

Problema 1

Confeccionar una aplicación que muestre una presentación en pantalla del programa. Solicite la carga de dos valores y nos muestre la suma. Mostrar finalmente un mensaje de despedida del programa.
Implementar estas actividades en tres funciones.

Proyecto70 - Principal.kt

```
fun presentacion() {
    println("Programa que permite cargar dos valores por teclado.")
    println("Efectua la suma de los valores")
    println("Muestra el resultado de la suma")
    println("*******************************")
}

fun cargarSumar() {
    print("Ingrese el primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 = readLine()!!.toInt()
    val suma = valor1 + valor2
    println("La suma de los dos valores es: $suma")
}

fun finalizacion() {
    println("*******************************")
    println("Gracias por utilizar este programa")
}

fun main(parametro: Array<String>) {
    presentacion()
    cargarSumar()
    finalizacion()
}
```

La forma de organizar nuestro programa cambia en forma radical.
El programa en Kotlin siempre comienza en la función main.

Es decir que el programa comienza en:

```
fun main(parametro: Array<String>) {
    presentacion()
    cargarSumar()
    finalizacion()
}
```

Como podemos ver en la función main llamamos a la tres funciones que declaramos previamente.

La sintaxis para declarar una función es mediante la palabra clave fun seguida por el nombre de la función (el nombre de la función no puede tener espacios en blanco ni comenzar con un número)

Luego del nombre de la función deben ir entre paréntesis los datos que llegan, si no llegan datos como es el caso de nuestras tres funciones solo se disponen paréntesis abierto y cerrado.

Todo el bloque de la función se encierra entre llaves y se indenta cuatro espacios como venimos trabajando con la función main.

Dentro de una función implementamos el algoritmo que pretendemos que resuelva esa función, por ejemplo la función presentacion tiene por objetivo mostrar en pantalla el objetivo del programa:

```
fun presentacion() {
    println("Programa que permite cargar dos valores por teclado.")
    println("Efectua la suma de los valores")
    println("Muestra el resultado de la suma")
    println("*******************************")
}
```

La función cargarSumar permite ingresar dos enteros por teclado, sumarlos y mostrarlos en pantalla:

```
fun cargarSumar() {
    print("Ingrese el primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 = readLine()!!.toInt()
    val suma = valor1 + valor2
    println("La suma de los dos valores es: $suma")
}
```

La función finalizacion() tiene por objetivo mostrar un mensaje que informe al operador que el programa finalizó:

```
fun finalizacion() {
    println("*******************************")
    println("Gracias por utilizar este programa")
}
```

Luego de definir las funciones tenemos al final de nuestro archivo Principal.kt las llamadas de las funciones dentro de la función main:

```
fun main(parametro: Array<String>) {
    presentacion()
    cargarSumar()
    finalizacion()
}
```

Si no hacemos las llamadas a las funciones los algoritmos que implementan las funciones nunca se ejecutarán.

Cuando en el bloque del programa principal se llama una función hasta que no finalice no continua con la llamada a la siguiente función:

```
Principal.kt

fun presentacion() {
    println("Programa que permite cargar dos valores por teclado.")
    println("Efectua la suma de los valores")
    println("Muestra el resultado de la suma")
    println("********************************")
}

fun cargarSumar() {
    print("Ingrese el primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 = readLine()!!.toInt()
    val suma = valor1 + valor2
    println("La suma de los dos valores es: $suma")
}

fun finalizacion() {
    println("********************************")
    println("Gracias por utilizar este programa")
}

fun main(parametro: Array<String>) {
    presentacion()
    cargarSumar()
    finalizacion()
}
```

En Kotlin la función main en realidad puede estar definida al principio del archivo y luego las otras funciones, lo que es importante decir que siempre un programa en Kotlin comienza a ejecutarse en la función main.

Problema 2

Confeccionar una aplicación que solicite la carga de dos valores enteros y muestre su suma.
Repetir la carga e impresión de la suma 5 veces.
Mostrar una línea separadora después de cada vez que cargamos dos valores y su suma.

Proyecto71 - Principal.kt

```
fun cargarSuma() {
    print("Ingrese el primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 = readLine()!!.toInt()
    val suma = valor1 + valor2
    println("La suma de los dos valores es: $suma")
}

fun separacion() {
    println("*******************************")
}

fun main(parametro: Array<String>) {
    for (i in 1..5) {
        cargarSuma()
        separacion()
    }
}
```

Proyecto71 - [C:\programaskotlin\Proyecto71] - [Proyecto71] - ...\src\Principal.kt - IntelliJ IDEA 2017....

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

Proyecto71 〉 src 〉 Principal.kt 〉 PrincipalKt

Project

- Proyecto71 C:\programasl
  - .idea
  - out
  - src
  - Proyecto71.iml
- External Libraries

Principal.kt

```kotlin
fun cargarSuma() {
    print("Ingrese el primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 = readLine()!!.toInt()
    val suma = valor1 + valor2
    println("La suma de los dos valores es: $suma")
}

fun separacion() {
    println("*******************************")
}

fun main(parametro: Array<String>) {
    for (i in 1..5) {
        cargarSuma()
        separacion()
    }
}
```

Run PrincipalKt

```
"C:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
Ingrese el primer valor:3
Ingrese el segundo valor:4
La suma de los dos valores es: 7
*******************************
Ingrese el primer valor:33
Ingrese el segundo valor:2
La suma de los dos valores es: 35
*******************************
Ingrese el primer valor:4
Ingrese el segundo valor:5
La suma de los dos valores es: 9
*******************************
Ingrese el primer valor:6
Ingrese el segundo valor:3
La suma de los dos valores es: 9
*******************************
Ingrese el primer valor:1
Ingrese el segundo valor:2
La suma de los dos valores es: 3
*******************************

Process finished with exit code 0
```

Compilation completed successfully in 2s 667ms (a minute ago) 24:1 CRLF UTF-8

Hemos declarado dos funciones, una que permite cargar dos enteros sumarlos y mostrar el resultado:

```
fun cargarSuma() {
    print("Ingrese el primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 = readLine()!!.toInt()
    val suma = valor1 + valor2
    println("La suma de los dos valores es: $suma")
}
```

Y otra función que tiene por objetivo mostrar una línea separadora con asteriscos:

```
fun separacion() {
    println("******************************")
}
```

Desde la función main llamamos a las funciones de cargarSumar y separación 5 veces:

```
fun main(parametro: Array<String>) {
    for (i in 1..5) {
        cargarSuma()
        separacion()
    }
}
```

Lo nuevo que debe quedar claro es que la llamada a las funciones desde la función main de nuestro programa puede hacerse múltiples veces.

Problemas propuestos

- Desarrollar un programa con dos funciones. La primer solicite el ingreso de un entero y muestre el cuadrado de dicho valor. La segunda que solicite la carga de dos valores y muestre el producto de los mismos. LLamar desde la main a ambas funciones.

- Desarrollar una función que solicite la carga de tres valores y muestre el menor. Desde la función main del programa llamar 2 veces a dicha función (sin utilizar una estructura repetitiva)

Solución

Retornar (index.php?inicio=0)

15 - Funciones: parámetros

Vimos en el concepto anterior que una función resuelve una parte de nuestro algoritmo.

Tenemos por un lado la declaración de la función por medio de un nombre y el algoritmo de la función seguidamente. Luego para que se ejecute la función la llamamos desde la función main.

Ahora veremos que una función puede tener parámetros para recibir datos. Los parámetros nos permiten comunicarle algo a la función y la hace más flexible.

Problema 1

Confeccionar una aplicación que muestre una presentación en pantalla del programa.
Solicite la carga de dos valores y nos muestre la suma.
Mostrar finalmente un mensaje de despedida del programa.

Proyecto74 - Principal.kt

```
fun mmostrarMensaje(mensaje: String) {
    println("**************************************************")
    println(mensaje)
    println("**************************************************")
}

fun cargarSumar() {
    print("Ingrese el primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 = readLine()!!.toInt()
    val suma = valor1 + valor2
    println("La suma de los dos valores es: $suma")
}

fun main(parametro: Array<String>) {
    mmostrarMensaje("El programa calcula la suma de dos valores ingresados por t
eclado.")
    cargarSumar()
    mmostrarMensaje("Gracias por utilizar este programa")
}
```

Ahora para resolver este pequeño problema hemos planteado una función llamada mmostrarMensaje que recibe como parámetro un String (cadena de caracteres) y lo muestra en pantalla.

Los parámetros van seguidos del nombre de la función encerrados entre paréntesis (y en el caso de tener más de un parámetro los mismos deben ir separados por coma):

```
fun mmostrarMensaje(mensaje: String) {
    println("*************************************************")
    println(mensaje)
    println("*************************************************")
}
```

Un parámetro podemos imaginarlo como una variable que solo se puede utilizar dentro de la función.

Ahora cuando llamamos a la función mmostrarMensaje desde la main de nuestro programa debemos pasar una variable String o un valor de tipo String:

```
    mmostrarMensaje("El programa calcula la suma de dos valores ingresados por
teclado.")
```

El String que le pasamos: "El programa calcula la suma de dos valores ingresados por teclado." lo recibe el parámetro de la función.

Una función con parámetros nos hace más flexible la misma para utilizarla en distintas circunstancias. En nuestro problema la función mostrarMensaje la utilizamos tanto para la presentación inicial de nuestro programa como para mostrar el mensaje de despedida. Si no existieran los parámetros estaríamos obligados a implementar dos funciones como el concepto anterior.

Problema 2

Confeccionar una función que reciba tres enteros y nos muestre el mayor de ellos. La carga de los valores hacerlo por teclado en la función main.

Proyecto75 - Principal.kt

```
fun mostrarMayor(v1: Int, v2: Int, v3: Int) {
    print("Mayor:")
    if (v1 > v2 && v1 > v3)
        println(v1)
    else
        if (v2 > v3)
            print(v2)
        else
            print(v3)
}

fun main(parametro: Array<String>) {
    print("Ingrese primer valor:")
    val valor1 = readLine()!!.toInt()
    print("Ingrese segundo valor:")
    val valor2 = readLine()!!.toInt()
    print("Ingrese tercer valor:")
    val valor3 = readLine()!!.toInt()
    mostrarMayor(valor1, valor2, valor3)
}
```

Es importante notar que un programa en Kotlin no se ejecuta en forma lineal las funciones definidas en el archivo *.kt sino que arranca en la función main.

En la función main se solicita el ingreso de tres enteros por teclado y llama a la función mostrarMayor y le pasa a sus parámetros las tres variable enteras valor1, valor2 y valor3.

La función mostrarMayor recibe en sus parámetros v1, v2 y v3 los valores cargados en las variables valor1, valor2 y valor3.

Los parámetros son la forma que nos permite comunicar la función main con la función mostrarMayor.

Dentro de la función mostrarMayor no podemos acceder a las variable valor1, valor2 y valor3 ya que son variables locales de la función main.

Problema 3

Desarrollar un programa que permita ingresar el lado de un cuadrado. Luego preguntar si quiere calcular y mostrar su perímetro o su superficie.

Proyecto76 - Principal.kt

```
fun mostrarPerimetro(lado: Int) {
    val perimetro = lado *4
    println("El perímetro es $perimetro")
}

fun mostrarSuperficie(lado: Int) {
    val superficie = lado * lado
    println("La superficie es $superficie")
}

fun main(parametro: Array<String>) {
    print("Ingrese el valor del lado de un cuadrado:")
    val la = readLine()!!.toInt()
    print("Quiere calcular el perimetro o la superficie[ingresar texto: perimetr
o/superficie]")
    var respuesta = readLine()!!
    when (respuesta){
        "perimetro" -> mostrarPerimetro(la)
        "superficie" -> mostrarSuperficie(la)
    }
}
```

Definimos dos funciones que calculan y muestran el perimetro por un lado y por otro la superficie:

```
fun mostrarPerimetro(lado: Int) {
    val perimetro = lado *4
    println("El perímetro es $perimetro")
}

fun mostrarSuperficie(lado: Int) {
    val superficie = lado * lado
    println("La superficie es $superficie")
}
```

En la función main cargamos el lado del cuadrado e ingresamos un String que indica que cálculo deseamos realizar si obtener el perímetro o la superficie. Una vez que se ingreso la variable respuesta procedemos a llamar a la función que efectúa el calculo respectivo pasando como dato la variable local "la" que almacena el valor del lado del cuadrado.

Los parámetros son la herramienta fundamental para pasar datos cuando hacemos la llamada a una función.

Problemas propuestos

- En la función main solicitar que se ingrese una clave dos veces por teclado. Desarrollar una función que reciba dos String como parametros y muestre un mensaje

si las dos claves ingresadas son iguales o distintas.

- Confeccionar una función que reciba tres enteros y los muestre ordenados de menor a mayor. En la función main solicitar la carga de 3 enteros por teclado y proceder a llamar a la primer función definida.

Solución

Retornar (index.php?inicio=0)

16 - Funciones: con retorno de datos

Vimos que una función la definimos mediante un nombre y que puede recibir datos por medio de sus parámetros.

Los parámetros son la forma para que una función reciba datos para ser procesados. Ahora veremos otra característica de las funciones que es la de devolver un dato a quien invocó la función (recordemos que una función la podemos llamar desde la función main o desde otra función que desarrollemos)

Problema 1

Confeccionar una función que le enviemos como parámetro el valor del lado de un cuadrado y nos retorne su superficie.

Proyecto79 - Principal.kt

```
fun retornarSuperficie(lado: Int): Int {
    val sup = lado * lado
    return sup
}

fun main(parametro: Array<String>) {
    print("Ingrese el valor del lado del cuafrado:")
    val la = readLine()!!.toInt()
    val superficie = retornarSuperficie(la)
    println("La superficie del cuadrado es $superficie")
}
```

Aparece una nueva palabra clave en Kotlin para indicar el valor devuelto por la función: return

La función retornarSuperficie recibe un parámetro llamado lado de tipo Int. Al final de la declaración de la función disponemos dos puntos y el tipo de dato que retorna la función, en este caso un Int:

```
fun retornarSuperficie(lado: Int): Int {
```

definimos una variable local llamada sup donde almacenamos el producto del parámetro lado por sí mismo.

La variable local sup es la que retorna la función mediante la palabra clave return:

```
fun retornarSuperficie(lado: Int): Int {
    val sup = lado * lado
    return sup
}
```

Hay que tener en cuenta que las variables locales (en este caso sup) solo se puede consultar dentro de la función donde se las define, no se tienen acceso a las mismas en la función main o dentro de otra función.

Hay un cambio importante cuando llamamos o invocamos a una función que devuelve un dato:

```
    val superficie = retornarSuperficie(la)
```

Es decir el valor devuelto por la función retornarSuperficie se almacena en la variable superficie.

Es un error lógico llamar a la función retornarSuperficie y no asignar el valor a una variable:

```
retornarSuperficie(la)
```

El dato devuelto (en nuestro caso la superficie del cuadrado) no se almacena.

Si podemos utilizar el valor devuelto para pasarlo a otra función:

```
    print("La superficie del cuadrado es ")
    println(retornarSuperficie(la))
```

La función retornarSuperficie devuelve un entero y se lo pasamos a la función println para que lo muestre.

En Kodlin podemos llamar dentro de un String a una función:

```
    print("La superficie del cuadrado es ${retornarSuperficie(la)}")
```

Debemos encerrarlo entre llaves y anteceder el caracter $ (luego esto es sustituido por el valor devuelto por la función)

Problema 2

Confeccionar una función que le enviemos como parámetros dos enteros y nos retorne el mayor.

Proyecto80 - Principal.kt

```
fun retornarMayor(v1: Int, v2: Int): Int {
    if (v1 > v2)
        return v1
    else
        return v2
}

fun main(parametro: Array<String>) {
    print("Ingrese el primer valor:")
    val valor1 =readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 =readLine()!!.toInt()
    println("El mayor entre $valor1 y $valor2 es ${retornarMayor(valor1, valor
2)}")
}
```

Nuevamente tenemos una función que recibe dos parámetros y retorna el mayor de ellos:

```
fun retornarMayor(v1: Int, v2: Int): Int {
    if (v1 > v2)
        return v1
    else
        return v2
}
```

Cuando una función encuentra la palabra return no sigue ejecutando el resto de la función sino que sale a la línea del programa desde donde llamamos a dicha función.

Problema 3

Confeccionar una función que le enviemos como parámetro un String y nos retorne la cantidad de caracteres que tiene. En la función main solicitar la carga de dos nombres por teclado y llamar a la función dos veces. Imprimir en la main cual de las dos palabras tiene más caracteres.

Proyecto81 - Principal.kt

```
fun largo(nombre: String): Int {
    return nombre.length
}

fun main(parametro: Array<String>) {
    print("Ingrese un nombre:")
    val nombre1 = readLine()!!
    print("Ingrese otro nombre:")
    val nombre2 = readLine()!!
    if (largo(nombre1) == largo(nombre2))
        print("Los nombres: $nombre1 y $nombre2 tienen la misma cantidad de cara
cteres")
    else
        if (largo(nombre1) > largo(nombre2))
            print("$nombre1 es mas largo")
        else
            print("$nombre2 es mas largo")
}
```

Hemos definido una función llamada largo que recibe un parámetro llamado nombre y retorna la cantidad de caracteres que tiene dicha cadena (accedemos a la propiedad length que tiene la clase String para obtener la cantidad de caracteres)

Desde la función main de nuestro programa llamamos a la función largo pasando las variables nombre1 y nombre2:

```
    if (largo(nombre1) == largo(nombre2))
        print("Los nombres: $nombre1 y $nombre2 tienen la misma cantidad de car
acteres")
    else
        if (largo(nombre1) > largo(nombre2))
            print("$nombre1 es mas largo")
        else
            print("$nombre2 es mas largo")
```

Problemas propuestos

- Elaborar una función que reciba tres enteros y nos retorne el valor promedio de los mismos.

- Elaborar una función que nos retorne el perímetro de un cuadrado pasando como parámetros el valor del lado.

- Confeccionar una función que calcule la superficie de un rectángulo y la retorne, la función recibe como parámetros los valores de dos de sus lados:

  ```
  fun retornarSuperficie(lado1: Int,lado2: Int): Int
  ```

En la función main del programa cargar los lados de dos rectángulos y luego mostrar cual de los dos tiene una superficie mayor.

Solución

Retornar (index.php?inicio=15)

17 - Funciones: con una única expresión

Las funciones de una única expresión se pueden expresar en Kotlin sin el bloque de llaves y mediante una asignación indicar el valor que retorna.

Recordemos que uno de los objetivos en Kotlin es permitirnos implementar los algoritmos en la forma más concisa posible.

Resolveremos algunas de las funciones ya planteadas utilizando esta nueva sintaxis.

Problema 1

Confeccionar una función que le enviemos como parámetro el valor del lado de un cuadrado y nos retorne su superficie.

Proyecto85 - Principal.kt

```
fun retornarSuperficie(lado: Int) = lado * lado

fun main(parametro: Array<String>) {
    print("Ingrese el valor del lado del cuafrado:")
    val la = readLine()!!.toInt()
    println("La superficie del cuadrado es ${retornarSuperficie(la)}")
}
```

Como podemos ver la implementación completa de la función es una sola línea:

```
fun retornarSuperficie(lado: Int) = lado * lado
```

Disponemos el operador = y seguidamente la expresión, en este caso el producto del parámetro lado por si mismo.

No hace falta indicar el tipo de dato que retorna la función ya que el compilador puede inferir que del producto lado * lado se genera un tipo de dato Int.

No hay problema de indicar el tipo de dato a retornar, pero en muchas situaciones el compilador lo puede inferir como es este caso:

```
fun retornarSuperficie(lado: Int): Int = lado * lado
```

La llamada a una función que contiene una única expresión no varía:

```
    println("La superficie del cuadrado es ${retornarSuperficie(la)}")
```

Problema 2

Confeccionar una función que le enviemos como parámetros dos enteros y nos retorne el mayor.

Proyecto86 - Principal.kt

```
fun retornarMayor(v1: Int, v2: Int) = if (v1 > v2) v1 else v2

fun main(parametro: Array<String>) {
    print("Ingrese el primer valor:")
    val valor1 =readLine()!!.toInt()
    print("Ingrese el segundo valor:")
    val valor2 =readLine()!!.toInt()
    println("El mayor entre $valor1 y $valor2 es ${retornarMayor(valor1, valor
2)}")
}
```

Teniendo en cuenta que la instrucción if puede disponerse como una expresión como lo vimos anteriormente está permitido su uso en las funciones con una única expresión:

```
fun retornarMayor(v1: Int, v2: Int) = if (v1 > v2) v1 else v2
```

Recordemos que el objetivo de codificar el algoritmo con esta sintaxis es hacer el código lo mas conciso, recordemos que la otra forma de expresar esta función es:

```
fun retornarMayor(v1: Int, v2: Int): Int {
    if (v1 > v2)
        return v1
    else
        return v2
}
```

Problema 3

Confeccionar una función reciba un entero comprendido entre 1 y 5 y nos retorne en castellano dicho número o un String con la cadena "error" si no está comprendido entre 1 y 5.

Proyecto87 - Principal.kt

```
fun convertirCastelano(valor: Int) = when (valor) {
    1 -> "uno"
    2 -> "dos"
    3 -> "tres"
    4 -> "cuatro"
    5 -> "cinco"
    else -> "error"
}

fun main(parametro: Array<String>) {
    for(i in 1..6)
        println(convertirCastelano(i))
}
```

En este problema mostramos que podemos utilizar la sentencia when como expresión de retorno de la función.

Problemas propuestos

Utilizar una única expresión en las funciones pedidas en estos problemas

- Elaborar una función que reciba tres enteros y nos retorne el valor promedio de los mismos.

- Elaborar una función que nos retorne el perímetro de un cuadrado pasando como parámetros el valor del lado.

- Confeccionar una función que calcule la superficie de un rectángulo y la retorne, la función recibe como parámetros los valores de dos de sus lados:

  ```
  fun retornarSuperficie(lado1: Int,lado2: Int): Int
  ```

  En la función main del programa cargar los lados de dos rectángulos y luego mostrar cual de los dos tiene una superficie mayor.

- Confeccionar una función que le enviemos como parámetro un String y nos retorne la cantidad de caracteres que tiene. En la función main solicitar la carga de dos nombres por teclado y llamar a la función dos veces. Imprimir en la main cual de las dos palabras tiene más caracteres.

Solución

Retornar (index.php?inicio=15)

18 - Funciones: con parámetros con valor por defecto

En Kotlin se pueden definir parámetros y asignarles un dato en la misma cabecera de la función. Luego cuando llamamos a la función podemos o no enviarle un valor al parámetro.

Los parámetros por defecto nos permiten crear funciones más flexibles y que se pueden emplear en distintas circunstancias.

Problema 1

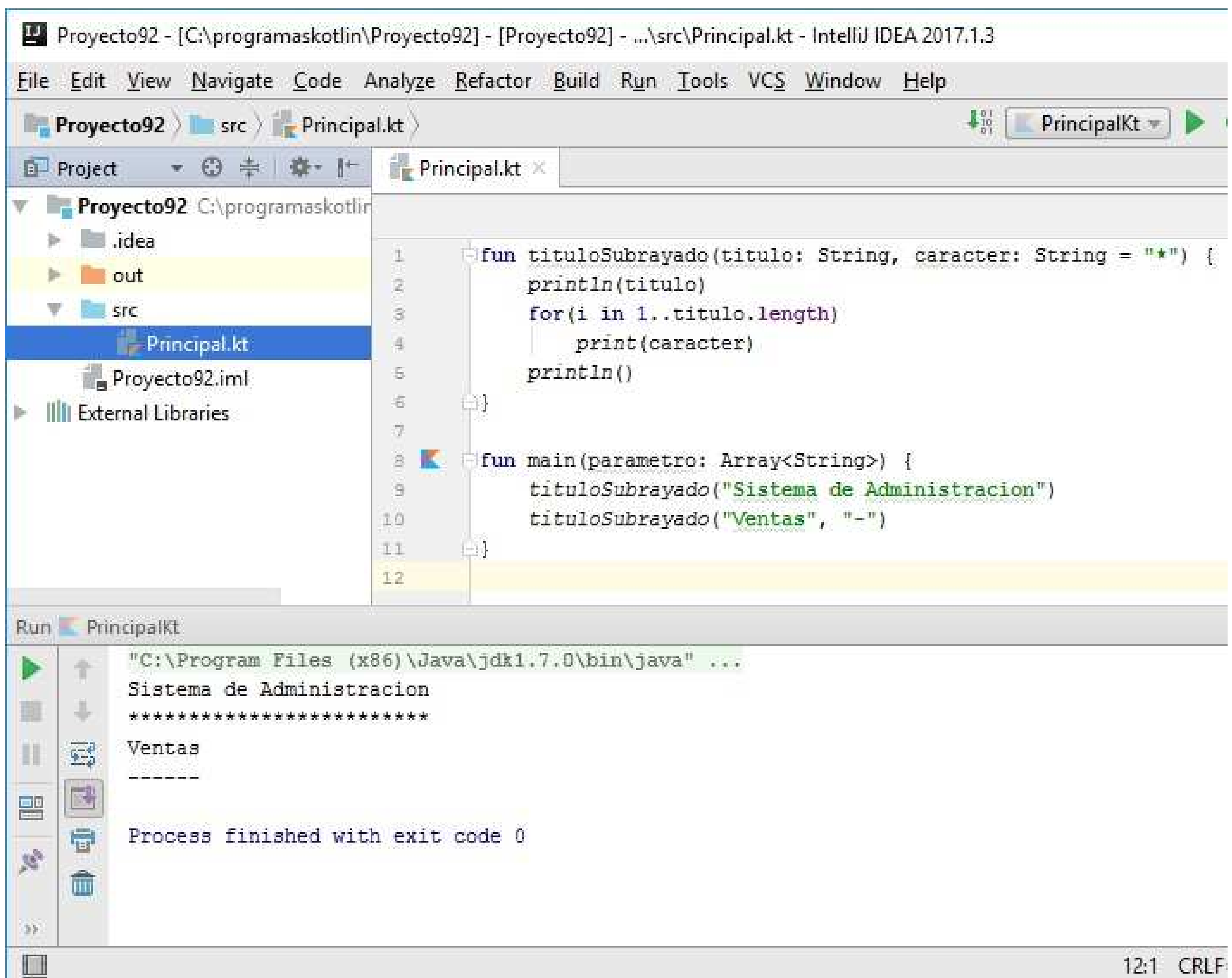
Confeccionar una función que reciba un String como parámetro y en forma opcional un segundo String con un caracter. La función debe mostrar el String subrayado con el caracter que indica el segundo parámetro

Proyecto92 - Principal.kt

```
fun tituloSubrayado(titulo: String, caracter: String = "*") {
    println(titulo)
    for(i in 1..titulo.length)
        print(caracter)
    println()
}

fun main(parametro: Array<String>) {
    tituloSubrayado("Sistema de Administracion")
    tituloSubrayado("Ventas", "-")
}
```

Cuando ejecutamos esta aplicación podemos observar el siguiente resultado:



Lo primero importante en notar que la llamada a la función tituloSubrayado la podemos hacer enviándole un dato o dos datos:

```
    tituloSubrayado("Sistema de Administracion")
    tituloSubrayado("Ventas", "-")
```

Esto no podría ser correcto si no utilizamos una sintaxis especial cuando declaramos los parámetros de la función:

```
fun tituloSubrayado(titulo: String, caracter: String = "*") {
    println(titulo)
    for(i in 1..titulo.length)
        print(caracter)
    println()
}
```

Como vemos el parámetro caracter tiene una asignación de un valor por defecto para los casos que llamamos a la función con un solo parámetro.

Cuando la llamamos a la función tituloSubrayado con un solo parámetro luego el parámetro caracter almacena el valor "*". Si llamamos a la función y le pasamos dos parámetros en nuestro ejemplo el parámetro caracter almacena el string "-"

El algoritmo de la función imprimir el primer parámetro y mediante un for que se repite tantas veces como el largo del "titulo" imprimimos el segundo parámetro

Problema propuesto

- Confeccionar una función que reciba entre 2 y 5 enteros. La misma nos debe retornar la suma de dichos valores. Debe tener tres parámetros por defecto.

Solución

Retornar (index.php?inicio=15)

19 - Funciones: llamada a la función con argumentos nombrados

Esta característica de Kotlin nos permite llamar a la función indicando en cualquier orden los parámetros de la misma, pero debemos especificar en la llamada el nombre del parámetro y el valor a enviarle.

Problema 1

Confeccionar una función que reciba el nombre de un operario, el pago por hora y la cantidad de horas trabajadas. Debe mostrar su sueldo y el nombre. Hacer la llamada de la función mediante argumentos nombrados.

Proyecto94 - Principal.kt

```
fun calcularSueldo(nombre: String, costoHora: Double, cantidadHoras: Int) {
    val sueldo = costoHora * cantidadHoras
    println("$nombre trabajó $cantidadHoras horas, se le paga por hora $costoHor
a por lo tanto le corresponde un sueldo de $sueldo")
}

fun main(parametro: Array<String>) {
    calcularSueldo("juan", 10.5, 120)
    calcularSueldo(costoHora = 12.0, cantidadHoras = 40, nombre="ana")
    calcularSueldo(cantidadHoras = 90, nombre = "luis", costoHora = 7.25)
}
```

Como podemos ver no hay ningún cambio cuando definimos la función:

```
fun calcularSueldo(nombre: String, costoHora: Double, cantidadHoras: Int) {
    val sueldo = costoHora * cantidadHoras
    println("$nombre trabajó $cantidadHoras horas, se le paga por hora $costoHo
ra por lo tanto le corresponde un sueldo de $sueldo")
}
```

Podemos llamarla como ya conocemos indicando los valores directamente:

```
    calcularSueldo("juan", 10.5, 120)
```

Pero también podemos indicar los datos en cualquier orden pero con la obligación de anteceder el nombre del parámetro:

```
calcularSueldo(costoHora = 12.0, cantidadHoras = 40, nombre="ana")
calcularSueldo(cantidadHoras = 90, nombre = "luis", costoHora = 7.25)
```

Problema propuesto

- Elaborar una función que muestre la tabla de multiplicar del valor que le enviemos como parámetro. Definir un segundo parámetro llamado termino que por defecto almacene el valor 10. Se deben mostrar tantos términos de la tabla de multiplicar como lo indica el segundo parámetro.
  Llamar a la función desde la main con argumentos nombrados.

Solución

Retornar (index.php?inicio=15)

20 - Funciones: internas o locales

Kotlin soporta funciones locales o internas, es decir, una función dentro de otra función.

Problema 1

Confeccionar una función que permita ingresar 10 valores por teclado y contar cuantos son múltiplos de 2 y cuantos son múltiplos de 5.

Proyecto96 - Principal.kt

```
fun multiplos2y5() {
    fun multiplo(numero: Int, valor: Int) = numero % valor == 0

    var mult2 = 0
    var mult5 = 0
    for(i in 1..10) {
        print("Ingrese valor:")
        val valor = readLine()!!.toInt()
        if (multiplo(valor, 2))
            mult2++
        if (multiplo(valor, 5))
            mult5++
    }
    println("Cantidad de múltiplos de 2 : $mult2")
    println("Cantidad de múltiplos de 5 : $mult5")
}

fun main(parametro: Array<String>) {
    multiplos2y5()
}
```

En este problema hemos definido una función llamada multiplos2y5 que tiene por objetivo cargar 10 enteros por teclado y verificar cuantos son múltiplos de 2 y cuantos múltiplos de 5.

Para verificar si un número es múltiplo de otro definimos una función local llamada "multiplo", la misma retorna true si el resto de dividir el primer parámetro con respecto al segundo es cero (Ej. 10 % 2 == 0 retorna true ya que el resto de dividir 10 con respecto a 2 es 0)

A una función interna solo la podemos llamar desde la misma función donde se la define, es decir la función multiplo solo puede ser llamada dentro de la función multiplos2y5. Si desde la función main tratamos de llamar a la función multiplo se genera un error en tiempo de compilación.

Problema propuesto

- Confeccionar una función que permita cargar dos enteros y nos muestre el mayor de ellos. Realizar esta actividad con 5 pares de valores.
  Implementar una función interna que retorne el mayor de dos enteros.

Solución

Retornar (index.php?inicio=15)

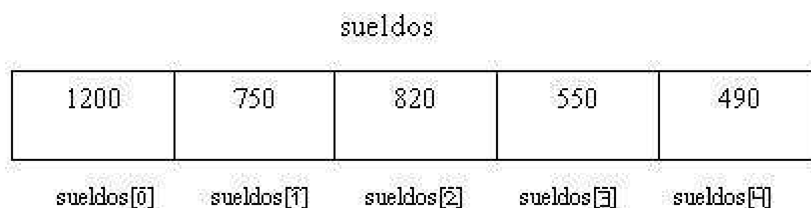
21 - Arreglos: conceptos

Hemos empleado variables de distinto tipo para el almacenamiento de datos (variables Int, Float, Double, Byte, Short, Long, Char, Boolean) En esta sección veremos otros tipos de variables que permiten almacenar un conjunto de datos en una única variable.

Un arreglo es una estructura de datos que permite almacenar un CONJUNTO de datos del MISMO tipo.
Con un único nombre se define un arreglo y por medio de un subíndice hacemos referencia a cada elemento del mismo (componente)

Problema 1

Se desea guardar los sueldos de 5 operarios.
Según lo conocido deberíamos definir 5 variables si queremos tener en un cierto momento los 5 sueldos almacenados en memoria.
Empleando un arreglo solo se requiere definir un único nombre y accedemos a cada elemento por medio del subíndice.



Proyecto98 - Principal.kt

```
fun main(parametro: Array<String>) {
    val sueldos: IntArray
    sueldos = IntArray(5)
    //carga de sus elementos por teclado
    for(i in 0..4) {
        print("Ingrese sueldo:")
        sueldos[i] = readLine()!!.toInt()
    }
    //impresion de sus elementos
    for(i in 0..4) {
        println(sueldos[i])
    }
}
```

Para declarar un arreglo de enteros definimos una variable de tipo IntArray:

```
val sueldos: IntArray
```

Para crearlo al arreglo y que se reserve espacio para 5 componentes debemos hacer:

```
sueldos = IntArray(5)
```

Para acceder a cada componente del arreglo utilizamos los corchetes y mediante un subíndice indicamos que componente estamos procesando:

```
for(i in 0..4) {
    print("Ingrese sueldo:")
    sueldos[i] = readLine()!!.toInt()
}
```

Cuando i valore 0 estamos cargando la primer componente del arreglo.

Las componentes comienzan a numerarse a partir de cero y llegan hasta el tamaño que le indicamos menos 1.

Una vez que cargamos todas sus componentes podemos imprimirlas una a una dentro de otro for:

```
for(i in 0..4) {
    println(sueldos[i])
}
```

Si queremos conocer el tamaño de un arreglo luego de haberse creado podemos acceder a la propiedad size:

```
val sueldos: IntArray
sueldos = IntArray(5)
println(sueldos.size) // se imprime un 5
```

Es más común crear un arreglo de enteros en una sola línea con la sintaxis:

```
val sueldos = IntArray(5)
```

Acotaciones

La biblioteca estándar de Kotlin (https://kotlinlang.org/api/latest/jvm/stdlib/index.html) contiene todas las clases básicas que se requieren para programar con este lenguaje organizado en paquetes.

En el paquete kotlin (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/index.html) podemos identificar que se encuentra declarada la clase IntArray.

Las otras clases de arreglos que suministra Kotlin son:

```
ByteArray
ShortArray
LongArray
FloatArray
DoubleArray
BooleanArray
CharArray
```

No hay uno para manejar String, en los próximos conceptos veremos como trabajar con este tipo de dato.

Problema 2

Definir un arreglo de 5 componentes de tipo Float que representen las alturas de 5 personas.
Obtener el promedio de las mismas. Contar cuántas personas son más altas que el promedio y cuántas más bajas.

Proyecto99 - Principal.kt

```
fun main(parametro: Array<String>) {
    val alturas = FloatArray(5)
    var suma = 0f
    for(i in 0..alturas.size-1){
        print("Ingrese la altura:")
        alturas[i] = readLine()!!.toFloat()
        suma += alturas[i]
    }
    val promedio = suma / alturas.size
    println("Altura promedio: $promedio")
    var altos = 0
    var bajos = 0
    for(i in 0..alturas.size-1)
        if (alturas[i] > promedio)
            altos++
        else
            bajos++
    println("Cantidad de personas más altas que el promedio: $altos")
    println("Cantidad de personas más bajas que el promedio: $bajos")
}
```

Creamos un arreglo con datos de tipo flotante de 5 elementos utilizando la clase FloatArray:

```
    val alturas = FloatArray(5)
```

En el primer for cargamos cada altura y la acumulamos en la variable suma. La variable suma se define por inferencia de tipo Float si le agregamos el caracter "f" o "F":

```
var suma = 0f
```

Dentro del for cargamos y acumulamos:

```
for(i in 0..alturas.size-1){
    print("Ingrese la altura:")
    alturas[i] = readLine()!!.toFloat()
    suma += alturas[i]
}
```

Luego cuando salimos del for obtenemos la altura promedio de las personas y la mostramos:

```
val promedio = suma / alturas.size
println("Altura promedio: $promedio")
```

Para contar la cantidad de personas más altas que el promedio y más bajas debemos definir dos contadores y dentro de otro for controlar cada altura con respecto al promedio:

```
var altos = 0
var bajos = 0
for(i in 0..alturas.size-1)
    if (alturas[i] > promedio)
        altos++
    else
        bajos++
```

Como el for tiene una sola sentencia no son obligatorias las llaves.

Fuera del for mostramos los dos contadores:

```
println("Cantidad de personas más altas que el promedio: $altos")
println("Cantidad de personas más bajas que el promedio: $bajos")
```

Problema 3

Cargar un arreglo de 10 elementos de tipo entero y verificar posteriormente si el mismo está ordenado de menor a mayor.

Proyecto100 - Principal.kt

```
fun main(parametro: Array<String>) {
    val arreglo = IntArray(10)
    for(i in 0..arreglo.size-1) {
        print("Ingrese elemento:")
        arreglo[i] = readLine()!!.toInt()
    }
    var ordenado = true
    for(i in 0..arreglo.size-2)
        if (arreglo[i+1] < arreglo[i])
            ordenado = false
    if (ordenado)
        print("Los elementos están ordenados de menor a mayor")
    else
        print("Los elementos no están ordenados de menor a mayor")
}
```

Definimos un arreglo de 10 elementos de tipo entero y procedemos a cargarlo por teclado:

```
    val arreglo = IntArray(10)
    for(i in 0..arreglo.size-1) {
        print("Ingrese elemento:")
        arreglo[i] = readLine()!!.toInt()
    }
```

Definimos una variable de tipo Boolean con el valor true (suponiendo que el arreglo está ordenado de menor a mayor antes de analizarlo.

```
    var ordenado = true
```

Por inferencia sabe el compilador que la variable ordenado debe ser de tipo Boolean, la otra forma de definir es:

```
    var ordenado: Boolean = true
```

Ahora mediante otro for procedemos a comparar un elemento de la posición i+1 con el de la posición i, si se cumple que sea mayor podemos ya inferir que el arreglo no está ordenado:

```
    for(i in 0..arreglo.size-2)
        if (arreglo[i+1] < arreglo[i])
            ordenado = false
```

Fuera del for preguntamos por el contenido de la variable "ordenado" si tiene almacenado el valor true significa que el vector está ordenado de menor a mayor:

```
    if (ordenado)
        print("Los elementos están ordenados de menor a mayor")
    else
        print("Los elementos no están ordenados de menor a mayor")
```

Si queremos hacer un poco más eficiente la verificación de si el array está ordenado podemos cortar las comparaciones en cuanto aparezca un elemento no ordenado mediante la palabra clave break:

```
    for(i in 0..arreglo.size-2)
        if (arreglo[i+1] < arreglo[i]){
            ordenado = false
            break
        }
```

El comando break sale de la estructura repetitiva que lo contiene en forma inmediata sin continuar el ciclo.

Propiedad indices de la clases IntArray, ByteArray, LongArray etc.

La clase IntArray tiene una propiedad IntRange llamada indices que almacena el rango mínimo y máximo del arreglo.
La propiedad indices podemos utilizarla en el for para recorrer las componentes:

Problema 4

Cargar un arreglo de 10 elementos de tipo entero. Imprimir luego el primer y último elemento.

Proyecto101 - Principal.kt

```
fun main(parametro: Array<String>) {
    val arreglo = IntArray(10)
    for(i in arreglo.indices) {
        print("Ingrese elemento:")
        arreglo[i] = readLine()!!.toInt()
    }
    println("Primera componente del arreglo ${arreglo[0]}")
    println("Ultima componente del arreglo ${arreglo[arreglo.lastIndex]}")
}
```

Es más conveniente utilizar la propiedad indices en lugar de disponer el rango: 0..arreglo.size-1 si vamos a recorrer todo el arreglo.

Para acceder a la última componente del arreglo utilizamos la propiedad lastIndex que devuelve el último índice válido:

```
    println("Ultima componente del arreglo ${arreglo[arreglo.lastIndex]}")
```

Iterar con un for un objeto array

Además de la forma que hemos visto para acceder a los elementos de un arreglo mediante un subíndice podemos utilizar la estructura repetitiva for con iteradores.

Problema 5

Cargar un arreglo de 5 elementos de tipo entero. Imprimir luego todo el arreglo.

Proyecto102 - Principal.kt

```
fun main(parametro: Array<String>) {
    val arreglo = IntArray(10)
    for(i in arreglo.indices) {
        print("Ingrese elemento:")
        arreglo[i] = readLine()!!.toInt()
    }
    for(elemento in arreglo)
        println(elemento)
}
```

Para iterar un arreglo completo de un array con un for utilizamos la siguiente sintaxis:

```
    for(elemento in arreglo)
        println(elemento)
```

Cada vuelta del for se almacena en la variable elemento un valor almacenado en la variable "arreglo".

Como vemos es una sintaxis muy simple para recorrer un arreglo y acceder a cada elementos para consultarlo.

Otras características de los arreglos que pueden ser útiles.

Imprimir cada elemento y su índice iterando el arreglo llamando al método withIndex de la clase IntArray:

```
    for((indice, elemento) in arreglo.withIndex())
        println("En el índice $indice se almacena el elemento $elemento")
```

Carga de los elementos utilizando el for como iterador;

```
for((indice, elemento) in arreglo.withIndex())
{
    print("Ingrese elemento:")
    arreglo[indice] = readLine()!!.toInt()
}
```

Problemas propuestos

- Desarrollar un programa que permita ingresar un arreglo de 8 elementos enteros, e informe:
  El valor acumulado de todos los elementos.
  El valor acumulado de los elementos que sean mayores a 36.
  Cantidad de valores mayores a 50.
  (Definir dos for, en el primero realizar la carga y en el segundo proceder a analizar cada elemento)

- Realizar un programa que pida la carga de dos arreglos numéricos enteros de 4 elementos. Obtener la suma de los dos arreglos elemento a elemento, dicho resultado guardarlo en un tercer arreglo del mismo tamaño.

Solución

Retornar (index.php?inicio=15)

22 - Funciones: parámetros y retorno de datos tipo arreglo

Hemos visto el objetivo de plantear funciones en un programa y que las mismas pueden recibir datos por medio de parámetros y retornar un dato.

Los parámetros de una función pueden ser de tipo Int, Char, Float etc. como hemos visto en conceptos anteriores pero también pueden ser de tipo arreglo como veremos en este concepto.

Problema 1

Definir en la función main un arreglo de enteros de 5 elementos. Declarar dos funciones, en una efectuar la carga de sus elementos y en la otra su impresión.

Proyecto105 - Principal.kt

```
fun cargar(arreglo: IntArray) {
    for(i in arreglo.indices) {
        print("Ingrese elemento:")
        arreglo[i] = readLine()!!.toInt()
    }
}

fun imprimir(arreglo: IntArray) {
    for(elemento in arreglo)
        println(elemento)
}

fun main(parametro: Array<String>) {
    val arre = IntArray(5)
    cargar(arre)
    imprimir(arre)
}
```

En la función main creamos un arreglo de 5 elementos de tipo entero mediante la clase IntArray:

```
fun main(parametro: Array<String>) {
    val arre = IntArray(5)
```

Llamamos seguidamente a la función cargar y le pasamos la referencia a nuestro arreglo:

```
    cargar(arre)
```

En la función cargar podemos acceder a los 5 elementos del arreglo para cargarlos:

```
fun cargar(arreglo: IntArray) {
    for(i in arreglo.indices) {
        print("Ingrese elemento:")
        arreglo[i] = readLine()!!.toInt()
    }
}
```

Como podemos observar el parámetro se llama arreglo y la variable que le pasamos de la main se llama arre. No hay problema que tengan nombres distintos pero si es obligatorio que los dos sean de tipo IntArray.

La función imprimir recibe el arreglo y muestra su contenido:

```
fun imprimir(arreglo: IntArray) {
    for(elemento in arreglo)
        println(elemento)
}
```

Problema 2

Se desea almacenar los sueldos de operarios. Cuando se ejecuta el programa se debe pedir la cantidad de sueldos a ingresar. Luego crear un arreglo con dicho tamaño.
Definir una función de carga y otra de impresión.

Proyecto106 - Principal.kt

```
fun cargar(): IntArray {
    print("Cuantos sueldos quiere cargar:")
    val cantidad = readLine()!!.toInt()
    val sueldos = IntArray(cantidad)
    for(i in sueldos.indices) {
        print("Ingrese elemento:")
        sueldos[i] = readLine()!!.toInt()
    }
    return sueldos
}

fun imprimir(sueldos: IntArray) {
    println("Listado de todos los sueldos")
    for(sueldo in sueldos)
        println(sueldo)
}

fun main(parametro: Array<String>) {
    val sueldos = cargar()
    imprimir(sueldos)
}
```

Este problema muestra como podemos crear un arreglo en una función y retornarla. La función cargar retorna la referencia de un objeto de tipo IntArray:

```
fun cargar(): IntArray {
```

Dentro de la función creamos un arreglo, cargamos su contenido y finalmente lo retornamos:

```
    print("Cuantos sueldos quiere cargar:")
    val cantidad = readLine()!!.toInt()
    val sueldos = IntArray(cantidad)
    for(i in sueldos.indices) {
        print("Ingrese elemento:")
        sueldos[i] = readLine()!!.toInt()
    }
    return sueldos
```

En la función main llamamos a la función cargar y le asignamos a una variable que por inferencia se detecta que es de tipo IntArray:

```
fun main(parametro: Array<String>) {
    val sueldos = cargar()
    imprimir(sueldos)
}
```

Problemas propuestos

- Desarrollar un programa que permita ingresar un arreglo de n elementos, ingresar n por teclado.
  Elaborar dos funciones una donde se lo cree y cargue al arreglo y otra que sume todos sus elementos y retorne dicho valor a la main donde se lo imprima.

- Cargar un arreglo de n elementos. Imprimir el menor elemento y un mensaje si se repite dentro del arreglo.

Solución

Retornar (index.php?inicio=15)

23 - POO - Conceptos de programación orientada a objetos

Kotlin nos permite utilizar la metodología de programación orientada a objetos.

Con la metodología de programación orientada a objetos (POO) se irán introduciendo conceptos de objeto, clase, propiedad, campo, método, constructor, herencia etc. y de todos estos temas se irán planteando problemas resueltos.

Prácticamente todos los lenguajes desarrollados en los últimos 25 años implementan la posibilidad de trabajar con POO (Programación Orientada a Objetos)

Conceptos básicos de Objetos

Un objeto es una entidad independiente con sus propios datos y programación. Las ventanas, menúes, carpetas de archivos pueden ser identificados como objetos; el motor de un auto también es considerado un objeto, en este caso, sus datos (campos y propiedades) describen sus características físicas y su programación (métodos) describen el funcionamiento interno y su interrelación con otras partes del automóvil (también objetos).

El concepto renovador de la tecnología Orientación a Objetos es la suma de funciones a elementos de datos, a esta unión se le llama encapsulamiento. Por ejemplo, un objeto Auto contiene ruedas, motor, velocidad, color, etc, llamados atributos. Encapsulados con estos datos se encuentran los métodos para arrancar, detenerse, dobla, frenar etc. La responsabilidad de un objeto auto consiste en realizar las acciones apropiadas y mantener actualizados sus datos internos. Cuando otra parte del programa (otros objetos) necesitan que el auto realice alguna de estas tareas (por ejemplo, arrancar) le envía un mensaje. A estos objetos que envían mensajes no les interesa la manera en que el objeto auto lleva a cabo sus tareas ni las estructuras de datos que maneja, por ello, están ocultos. Entonces, un objeto contiene información pública, lo que necesitan los otros objetos para interactuar con él e información privada, interna, lo que necesita el objeto para operar y que es irrelevante para los otros objetos de la aplicación.

Concepto de Clase y definición de Objetos

La programación orientada a objetos se basa en la programación de clases; a diferencia de la programación estructurada, que está centrada en las funciones.

Una clase es un molde del que luego se pueden crear múltiples objetos, con similares características.

Una clase es una plantilla (molde), que define propiedades (variables) y métodos (funciones)

La clase define las propiedades y métodos comunes a los objetos de ese tipo, pero luego, cada objeto tendrá sus propios valores y compartirán las mismas funciones.

Debemos crear una clase antes de poder crear objetos (instancias) de esa clase. Al crear un objeto de una clase, se dice que se crea una instancia de la clase o un objeto propiamente dicho.

La estructura básica en Kotlin de una clase es:

```
class [nombre de la clase] {
  [propiedades de la clase]
  [métodos o funciones de la clase]
}
```

Problema 1

Implementaremos una clase llamada Persona que tendrá como propiedades (variables) su nombre y edad, y tres métodos (funciones), uno de dichos métodos inicializará las propiedades del nombre y la edad, otro método mostrará en la pantalla el contenido de las propiedades y por último uno que imprima si es mayor de edad.

Definir dos objetos de la clase Persona.

Proyecto109 - Principal.kt

```
class Persona {
    var nombre: String = ""
    var edad: Int = 0

    fun inicializar(nombre: String, edad: Int) {
        this.nombre = nombre
        this.edad = edad
    }

    fun imprimir() {
        println("Nombre: $nombre y tiene una edad de $edad")
    }

    fun esMayorEdad() {
        if (edad >= 18)
            println("Es mayor de edad $nombre")
        else
            println("No es mayor de edad $nombre")
    }
}

fun main(parametro: Array<String>) {
    val persona1: Persona
    persona1 = Persona()
    persona1.inicializar("Juan", 12)
    persona1.imprimir()
    persona1.esMayorEdad()
    val persona2: Persona
    persona2 = Persona()
    persona2.inicializar("Ana", 50)
    persona2.imprimir()
    persona2.esMayorEdad()
}
```

El nombre de la clase debe hacer referencia al concepto (en este caso la hemos llamado Persona). La palabra clave para declarar la clase es class, seguidamente el nombre de la clase y luego una llave de apertura que debe cerrarse al final de la declaración de la clase:

```
class Persona {
```

Definimos dos propiedades llamadas nombre y edad, las inicializamos con un String vacío a una y con 0 a la otra:

```
    var nombre: String = ""
    var edad: Int = 0
```

Una de las premisas del lenguaje Kotlin es que sea seguro y no permite definir una propiedad y no asignarle un valor y que quede con el valor null

Luego definimos sus tres métodos (es lo que conocemos como funciones hasta ahora, pero al estar dentro de una clase se las llama métodos)

El método inicializar recibe como parámetros un String y un Int con el objetivo de cargar las propiedades nombre y edad:

```
    fun inicializar(nombre: String, edad: Int) {
        this.nombre = nombre
        this.edad = edad
    }
```

Como los parámetros se llaman en este caso igual que las propiedades las diferenciamos antecediendo la palabra clave this al nombre de la propiedad:

```
        this.nombre = nombre
        this.edad = edad
```

El método imprimir tiene por objetivo mostrar el contenido de las dos propiedades:

```
    fun imprimir() {
        println("Nombre: $nombre y tiene una edad de $edad")
    }
```

Como en este método no hay parámetros que se llamen igual a las propiedades podemos acceder a las propiedades directamente por su nombre y no estar obligados a anteceder el operador this, no habría problema de anteceder el this y escribir esto:

```
    fun imprimir() {
        println("Nombre: ${this.nombre} y tiene una edad de ${this.edad}")
    }
```

El tercer y último método tiene por objetivo mostrar un mensaje si la persona es mayor de edad o no:

```
    fun esMayorEdad() {
        if (edad >= 18)
            println("Es mayor de edad $nombre")
        else
            println("No es mayor de edad $nombre")
    }
```

Decíamos que una clase es un molde que nos permite crear objetos. Ahora veamos cual es la sintaxis para la creación de objetos de la clase Persona:

```
fun main(parametro: Array<String>) {
    val persona1: Persona
    persona1 = Persona()
    persona1.inicializar("Juan", 12)
    persona1.imprimir()
    persona1.esMayorEdad()
```

Primero debemos definir una variable de tipo Persona:

```
    val persona1: Persona
```

Para crear el objeto debemos asignar a la variable persona1 el nombre de la clase y unos paréntesis abiertos y cerrados:

```
    persona1 = Persona()
```

Una vez que hemos creado el objeto podemos llamar a sus métodos antecediendo primero el nombre del objeto (persona1):

```
    persona1.inicializar("Juan", 12)
    persona1.imprimir()
    persona1.esMayorEdad()
```

Es importante el orden que llamamos a los métodos, por ejemplo si primero llamamos a imprimir antes de inicializar, veremos que muestra una edad de cero y un String vacío como nombre.

Una clase es un molde que nos permite crear tantos objetos como necesitemos, en nuestro problema debemos crear dos objetos de la clase Persona, el segundo lo creamos en forma similar al primero:

```
    val persona2: Persona
    persona2 = Persona()
    persona2.inicializar("Ana", 50)
    persona2.imprimir()
    persona2.esMayorEdad()
```

Acotación.

Hemos visto que Kotlin busca ser conciso, podemos en una sola línea definir el objeto y crearlo con la siguiente sintaxis:

```
    val persona1 = Persona()
```

En lugar de:

```
val persona1: Persona
persona1 = Persona()
```

Esto nos debe recordar a conceptos anteriores cuando definimos un objeto de la clase Int:

```
val peso: Int
peso = 40
```

Y en forma concisa escribimos:

```
val peso = 40
```

Problema 2

Implementar una clase que cargue los lados de un triángulo e implemente los siguientes métodos: inicializar las propiedades, imprimir el valor del lado mayor y otro método que muestre si es equilátero o no.

Proyecto110 - Principal.kt

```
class Triangulo {
    var lado1: Int = 0
    var lado2: Int = 0
    var lado3: Int =0

    fun inicializar() {
        print("Ingrese lado 1:")
        lado1 = readLine()!!.toInt()
        print("Ingrese lado 2:")
        lado2 = readLine()!!.toInt()
        print("Ingrese lado 3:")
        lado3 = readLine()!!.toInt()
    }

    fun ladoMayor() {
        print("Lado mayor:")
        when {
            lado1 > lado2 && lado1 > lado3 -> println(lado1)
            lado2 > lado3 -> println(lado2)
            else -> println(lado3)
        }
    }

    fun esEquilatero() {
        if (lado1 == lado2 && lado1 == lado3)
            print("Es un triángulo equilátero")
        else
            print("No es un triángulo equilátero")
    }
}


fun main(parametro: Array<String>) {
    val triangulo1 = Triangulo()
    triangulo1.inicializar()
    triangulo1.ladoMayor()
    triangulo1.esEquilatero()
}
```

La clase Triangulo define tres propiedades donde almacenamos los lados del triángulo:

```
class Triangulo {
    var lado1: Int = 0
    var lado2: Int = 0
    var lado3: Int =0
```

El método inicializar, que debemos recordar que sea el primero en ser llamado procede a cargar por teclado los tres lados del triángulo:

```
    fun inicializar() {
        print("Ingrese lado 1:")
        lado1 = readLine()!!.toInt()
        print("Ingrese lado 2:")
        lado2 = readLine()!!.toInt()
        print("Ingrese lado 3:")
        lado3 = readLine()!!.toInt()
    }
```

El segundo método verifica cual de los tres lados tiene almacenado un valor mayor utilizando la sentencia when (en lugar de una serie de if anidados):

```
    fun ladoMayor() {
        print("Lado mayor:")
        when {
            lado1 > lado2 && lado1 > lado3 -> println(lado1)
            lado2 > lado3 -> println(lado2)
            else -> println(lado3)
        }
    }
```

El tercer método verifica si se trata de un triángulo equilátero:

```
    fun esEquilatero() {
        if (lado1 == lado2 && lado1 == lado3)
            print("Es un triángulo equilátero")
        else
            print("No es un triángulo equilátero")
    }
```

En la función main definimos un objeto de la clase Triangulo y llamamos a sus métodos:

```
fun main(parametro: Array<String>) {
    val triangulo1 = Triangulo()
    triangulo1.inicializar()
    triangulo1.ladoMayor()
    triangulo1.esEquilatero()
}
```

Problema propuesto

- Implementar una clase llamada Alumno que tenga como propiedades su nombre y su nota. Definir los métodos para inicializar sus propiedades por teclado, imprimirlos y mostrar un mensaje si está regular (nota mayor o igual a 4)
  Definir dos objetos de la clase Alumno.

Solución

Retornar (index.php?inicio=15)

24 - POO - Constructor de la clase

En Kotlin podemos definir un método que se ejecute inicialmente y en forma automática. Este método se lo llama constructor.

El constructor tiene las siguientes características:

- Es el primer método que se ejecuta.
- Se ejecuta en forma automática.
- No puede retornar datos.
- Se ejecuta una única vez.
- Un constructor tiene por objetivo inicializar atributos.
- Una clase puede tener varios constructores pero solo uno es el principal.

Problema 1

Implementar una clase llamada Persona que tendrá como propiedades su nombre y edad.
Plantear un constructor donde debe llegar como parámetros el nombre y la edad.
Definir además dos métodos, uno que imprima las propiedades y otro muestre si es mayor de edad.

Proyecto112 - Principal.kt

```
class Persona constructor(nombre: String, edad: Int) {
    var nombre: String = nombre
    var edad: Int = edad

    fun imprimir() {
        println("Nombre: $nombre y tiene una edad de $edad")
    }

    fun esMayorEdad() {
        if (edad >= 18)
            println("Es mayor de edad $nombre")
        else
            println("No es mayor de edad $nombre")
    }
}

fun main(parametro: Array<String>) {
    val persona1 = Persona("Juan", 12)
    persona1.imprimir()
    persona1.esMayorEdad()
}
```

El constructor principal de la clase se lo declara inmediatamente luego de definir el nombre de la clase:

```
class Persona constructor(nombre: String, edad: Int) {
```

Podemos ver que no tiene un bloque de llaves con código y podemos asignar los parámetros a propiedades que define la clase:

```
    var nombre: String = nombre
    var edad: Int = edad
```

En la función main donde definimos un objeto de la clase Persona debemos pasar en forma obligatoria los datos que recibe el constructor:

```
    val persona1 = Persona("Juan", 12)
```

Por eso decimos que el constructor se ejecuta en forma automática y tiene por objetivo inicializar propiedades del objeto que se crea.

La llamada a los otros métodos de la clase no varía en nada a lo visto en el concepto anterior:

```
    persona1.imprimir()
    persona1.esMayorEdad()
```

Acotaciones

Palabra clave constructor opcional.

En muchas situaciones como esta la palabra clave constructor es opcional y en forma más concisa podemos escribir la declaración de la clase y su constructor principal con la sintaxis:

```
class Persona (nombre: String, edad: Int) {
```

Veremos en conceptos futuros que es obligatorio la palabra constructor cuando agregamos un modificador de acceso (private, protected, public, internal) previo al constructor o una anotación.

Definición de propiedades en el mismo constructor.

Esto también lo implementa Kotlin para favorecer que el programa sea lo más conciso posible (reducir la cantidad de líneas de código)

El programa completo queda ahora con la sintaxis:

```
class Persona (var nombre: String, var edad: Int) {

    fun imprimir() {
        println("Nombre: $nombre y tiene una edad de $edad")
    }

    fun esMayorEdad() {
        if (edad >= 18)
            println("Es mayor de edad $nombre")
        else
            println("No es mayor de edad $nombre")
    }
}

fun main(parametro: Array<String>) {
    val persona1 = Persona("Juan", 12)
    persona1.imprimir()
    persona1.esMayorEdad()
}
```

Es importante ver que cuando declaramos el constructor hemos definido las dos propiedades:

```
class Persona (var nombre: String, var edad: Int) {
```

No hace falta declararlas dentro de la clase y si lo hacemos nos genera un error sintáctico ya que estaremos definiendo dos veces con el mismo nombre una propiedad:

```
class Persona (var nombre: String, var edad: Int) {
    var nombre: String = ""
    var edad: Int = 0
```

Esto no significa que no se vayan a definir otras propiedades en la clase, solo las que se inicializan en el constructor las definimos en el mismo.

Bloque init

Si en algunas situaciones queremos ejecutar un algoritmo inmediatamente después del constructor debemos implementar un bloque llamado init.

En este bloque podemos por ejemplo validar los datos que llegan al constructor e inicializar otras propiedades de la clase.

Modificaremos nuevamente el programa para verificar si en el parámetro de la edad llega un valor menor a cero:

```
class Persona (var nombre: String, var edad: Int) {

    init {
        if (edad < 0)
            edad = 0
    }

    fun imprimir() {
        println("Nombre: $nombre y tiene una edad de $edad")
    }

    fun esMayorEdad() {
        if (edad >= 18)
            println("Es mayor de edad $nombre")
        else
            println("No es mayor de edad $nombre")
    }
}

fun main(parametro: Array<String>) {
    val persona1 = Persona("Juan", -12)
    persona1.imprimir()
    persona1.esMayorEdad()
}
```

El bloque init debe ir encerrado entre llaves e implementamos un algoritmo que se ejecutará inmediatamente después del constructor. En nuestro ejemplo si la propiedad edad se carga un valor negativo procedemos a asignarle un cero:

```
    init {
        if (edad < 0)
            edad = 0
    }
```

Problema 2

Implementar una clase que cargue los lados de un triángulo e implemente los siguientes métodos: inicializar las propiedades, imprimir el valor del lado mayor y otro método que muestre si es equilátero o no.

Proyecto113 - Principal.kt

```
class Triangulo (var lado1: Int, var lado2: Int, var lado3: Int){

    fun ladoMayor() {
        print("Lado mayor:")
        when {
            lado1 > lado2 && lado1 > lado3 -> println(lado1)
            lado2 > lado3 -> println(lado2)
            else -> println(lado3)
        }
    }

    fun esEquilatero() {
        if (lado1 == lado2 && lado1 == lado3)
            print("Es un triángulo equilátero")
        else
            print("No es un triángulo equilátero")
    }
}

fun main(parametro: Array<String>) {
    val triangulo1 = Triangulo(12, 45, 24)
    triangulo1.ladoMayor()
    triangulo1.esEquilatero()
}
```

Definimos tres propiedades en el constructor principal de la clase:

```
class Triangulo (var lado1: Int, var lado2: Int, var lado3: Int){
```

Cuando creamos un objeto de la clase Triangulo en la función main le pasamos los valores de los tres lados del triángulo:

```
fun main(parametro: Array<String>) {
    val triangulo1 = Triangulo(12, 45, 24)
    triangulo1.ladoMayor()
    triangulo1.esEquilatero()
}
```

Definición de varios constructores

Cuando se define otro constructor aparte del principal de la clase debe implementarse obligatoriamente con la palabra clave constructor, sus parámetros y la llamada obligatoria al constructor principal de la clase. En un bloque de llaves desarrollamos el algoritmo del mismo.

Problema 3

Implementar una clase que cargue los lados de un triángulo e implemente los siguientes métodos: inicializar las propiedades, imprimir el valor del lado mayor y otro método que muestre si es equilátero o no.

Plantear el constructor principal que reciba los valores de los lados y un segundo constructor que permita ingresar por teclado los tres lados.

Proyecto114 - Principal.kt

```
class Triangulo (var lado1: Int, var lado2: Int, var lado3: Int){

    constructor():this(0, 0, 0) {
        print("Ingrese primer lado:")
        lado1 = readLine()!!.toInt()
        print("Ingrese segundo lado:")
        lado2 = readLine()!!.toInt()
        print("Ingrese tercer lado:")
        lado3 = readLine()!!.toInt()
    }

    fun ladoMayor() {
        print("Lado mayor:")
        when {
            lado1 > lado2 && lado1 > lado3 -> println(lado1)
            lado2 > lado3 -> println(lado2)
            else -> println(lado3)
        }
    }

    fun esEquilatero() {
        if (lado1 == lado2 && lado1 == lado3)
            println("Es un triángulo equilátero")
        else
            println("No es un triángulo equilátero")
    }
}

fun main(parametro: Array<String>) {
    val triangulo1 = Triangulo()
    triangulo1.ladoMayor()
    triangulo1.esEquilatero()
    val triangulo2 = Triangulo(6, 6, 6)
    triangulo2.ladoMayor()
    triangulo2.esEquilatero()
}
```

El constructor principal con la definición de tres propiedades es:

```
class Triangulo (var lado1: Int, var lado2: Int, var lado3: Int){
```

El segundo constructor debe llamar obligatoriamente al constructor principal antecediendo la palabra clave this y entre paréntesis los datos a enviar:

```
    constructor():this(0, 0, 0) {
```

Luego entre llaves el algoritmo propiamente dicho de ese constructor:

```
    constructor():this(0, 0, 0) {
        print("Ingrese primer lado:")
        lado1 = readLine()!!.toInt()
        print("Ingrese segundo lado:")
        lado2 = readLine()!!.toInt()
        print("Ingrese tercer lado:")
        lado3 = readLine()!!.toInt()
    }
```

Ahora cuando creamos un objeto de la clase Triangulo podemos llamar a uno u otro constructor según nuestras necesidades.

Si queremos cargar por teclado los tres lados del triángulo debemos llamar al constructor que no tiene parámetros:

```
    val triangulo1 = Triangulo()
    triangulo1.ladoMayor()
    triangulo1.esEquilatero()
```

Si ya sabemos los valores de cada lado del triángulo se los pasamos en la llamada al constructor:

```
    val triangulo2 = Triangulo(6, 6, 6)
    triangulo2.ladoMayor()
    triangulo2.esEquilatero()
```

Problemas propuestos

- Implementar una clase llamada Alumno que tenga como propiedades su nombre y su nota. Al constructor llega su nombre y nota.
  Imprimir el nombre y su nota. Mostrar un mensaje si está regular (nota mayor o igual a 4)
  Definir dos objetos de la clase Alumno.

- Cofeccionar una clase que represente un punto en el plano, la coordenada de un punto en el plano está dado por dos valores enteros x e y.
  Al constructor llega la ubicación del punto en x e y.
  Implementar un método que retorne un String que indique en que cuadrante se ubica dicho punto. (1º Cuadrante si x > 0 Y y > 0 , 2º Cuadrante: x < 0 Y y > 0, 3º Cuadrante: x < 0 Y y < 0, 4º Cuadrante: x > 0 Y y < 0)
  Si alguno o los dos valores son cero luego el punto se encuentra en un eje.
  Definir luego 5 objetos de la clase Punto en cada uno de los cuadrantes y uno en un eje.

Solución

Retornar (index.php?inicio=15)

25 - POO - Llamada de métodos desde otro método de la misma clase

Hasta ahora todos los problemas planteados hemos llamado a los métodos desde donde definimos un objeto de dicha clase, por ejemplo:

```
    val persona1 = Persona("Juan", 12)
    persona1.imprimir()
    persona1.esMayorEdad()
```

Utilizamos la sintaxis:

```
[nombre del objeto].[nombre del método]
```

Es decir antecedemos al nombre del método el nombre del objeto y el operador punto.

Ahora bien que pasa si queremos llamar dentro de la clase a otro método que pertenece a la misma clase, la sintaxis es la siguiente:

```
[nombre del método]
```

O en forma larga:

```
this.[nombre del método]
```

Es importante tener en cuenta que esto solo se puede hacer cuando estamos dentro de la misma clase.

Problema 1

Plantear una clase Operaciones que en un método solicite la carga de 2 enteros y posteriormente llame a otros dos métodos que calculen su suma y producto.

Proyecto117 - Principal.kt

```
class Operaciones {
    var valor1: Int = 0
    var valor2: Int = 0

    fun cargar() {
        print("Ingrese primer valor:")
        valor1 = readLine()!!.toInt()
        print("Ingrese segundo valor:")
        valor2 = readLine()!!.toInt()
        sumar()
        restar()
    }

    fun sumar() {
        val suma = valor1 + valor2
        println("La suma de $valor1 y $valor2 es $suma")
    }

    fun restar() {
        val resta = valor1 - valor2
        println("La resta de $valor1 y $valor2 es $resta")
    }
}

fun main(parametro: Array<String>) {
    val operaciones1 = Operaciones()
    operaciones1.cargar()
}
```

Nuestro método cargar además de cargar los dos enteros en las propiedades procede a llamar a los métodos que calculan la suma y resta de los dos valores ingresados.

La llamada de los métodos de la misma clase se hace indicando el nombre del método.

Desde donde definimos un objeto de la clase Operaciones llamamos a sus métodos antecediendo el nombre del objeto:

```
    val operaciones1 = Operaciones()
    operaciones1.cargar()
```

Problema propuesto

- Declarar una clase llamada Hijos. Definir dentro de la misma un arreglo para almacenar las edades de 5 personas.
  Definir un método cargar donde se ingrese por teclado el arreglo de las edades y llame a otros dos método que impriman la mayor edad y el promedio de edades.

Solución

Retornar (index.php?inicio=15)

26 - POO - Colaboración de clases

Normalmente un problema resuelto con la metodología de programación orientada a objetos no interviene una sola clase, sino que hay muchas clases que interactúan y se comunican.

Plantearemos problemas separando las actividades en dos clases.

Problema 1

Un banco tiene 3 clientes que pueden hacer depósitos y extracciones. También el banco requiere que al final del día calcule la cantidad de dinero que hay depositado.

Lo primero que hacemos es identificar las clases:
Podemos identificar la clase Cliente y la clase Banco.

Luego debemos definir las propiedades y los métodos de cada clase:

```
Cliente
    propiedades
        nombre
        monto
    métodos
        depositar
        extraer
        imprimir

Banco
    propiedades
        3 Cliente (3 objetos de la clase Cliente)
    métodos
        operar
        depositosTotales
```

Proyecto119 - Principal.kt

```
class Cliente(var nombre: String, var monto: Float) {

    fun depositar(monto: Float) {
        this.monto += monto
    }

    fun extraer(monto: Float) {
        this.monto -= monto
    }

    fun imprimir() {
        println("$nombre tiene depositado la suma de $monto")
    }
}

class Banco {
    val cliente1: Cliente = Cliente("Juan", 0f)
    var cliente2: Cliente = Cliente("Ana", 0f)
    var cliente3: Cliente = Cliente("Luis", 0f)

    fun operar() {
        cliente1.depositar(100f)
        cliente2.depositar(150f)
        cliente3.depositar(200f)
        cliente3.extraer(150f)
    }

    fun depositosTotales() {
        val total = cliente1.monto + cliente2.monto + cliente3.monto
        println("El total de dinero del banco es: $total")
        cliente1.imprimir()
        cliente2.imprimir()
        cliente3.imprimir()
    }
}

fun main(parametro: Array<String>) {
    val banco1 = Banco()
    banco1.operar()
    banco1.depositosTotales()
}
```

Primero hacemos la declaración de la clase Cliente, al constructor principal llegan el nombre del cliente y el monto inicial depositado (las propiedades las estamos definiendo en el propio constructor):

```
class Cliente(var nombre: String, var monto: Float) {
```

El método que aumenta la propiedad monto es:

```
    fun depositar(monto: Float) {
        this.monto += monto
    }
```

Y el método que reduce la propiedad monto del cliente es:

```
    fun extraer(monto: Float) {
        this.monto -= monto
    }
```

Para mostrar los datos del cliente tenemos el método:

```
    fun imprimir() {
        println("$nombre tiene depositado la suma de $monto")
    }
```

La segunda clase de nuestro problema es el Banco. Esta clase define tres propiedades de la clase Cliente (la clase Cliente colabora con la clase Banco):

```
class Banco {
    val cliente1: Cliente = Cliente("Juan", 0f)
    var cliente2: Cliente = Cliente("Ana", 0f)
    var cliente3: Cliente = Cliente("Luis", 0f)
```

El método operar realiza una serie de depósitos y extracciones de los clientes:

```
    fun operar() {
        cliente1.depositar(100f)
        cliente2.depositar(150f)
        cliente3.depositar(200f)
        cliente3.extraer(150f)
    }
```

El método que muestra cuanto dinero tiene depositado el banco se resuelve accediendo a la propiedad monto de cada cliente:

```
    fun depositosTotales() {
        val total = cliente1.monto + cliente2.monto + cliente3.monto
        println("El total de dinero del banco es: $total")
        cliente1.imprimir()
        cliente2.imprimir()
        cliente3.imprimir()
    }
```

En la funcíób main de nuestro programa procedemos a crear un objeto de la clase Banco y llamar a los dos métodos:

```
fun main(parametro: Array<String>) {
    val banco1 = Banco()
    banco1.operar()
    banco1.depositosTotales()
}
```

Problema 2

Plantear un programa que permita jugar a los dados. Las reglas de juego son:
se tiran tres dados si los tres salen con el mismo valor mostrar un mensaje que "gano", sino "perdió".

Lo primero que hacemos es identificar las clases:

Podemos identificar la clase Dado y la clase JuegoDeDados.

Luego las propiedades y los métodos de cada clase:

```
Dado
    propiedades
        valor
    métodos
        tirar
        imprimir

JuegoDeDados
    atributos
        3 Dado (3 objetos de la clase Dado)
    métodos
        jugar
```

Proyecto120 - Principal.kt

```
class Dado (var valor: Int){

    fun tirar() {
        valor = ((Math.random() * 6) + 1).toInt()
        imprimir()
    }

    fun imprimir() {
        println("Valor del dado: $valor")
    }
}

class JuegoDeDados {
    val dado1 = Dado(1)
    val dado2 = Dado(1)
    val dado3 = Dado(1)

    fun jugar() {
        dado1.tirar()
        dado2.tirar()
        dado3.tirar()
        if (dado1.valor == dado2.valor && dado2.valor == dado3.valor)
            println("Ganó")
        else
            print("Perdió")
    }
}

fun main(parametro: Array<String>) {
    val juego1 = JuegoDeDados()
    juego1.jugar()
}
```

La clase Dado define un método tirar que almacena en la propiedad valor un número aleatorio comprendido entre 1 y 6. Además llama al método imprimir para mostrarlo:

```
class Dado (var valor: Int){

    fun tirar() {
        valor = ((Math.random() * 6) + 1).toInt()
        imprimir()
    }
```

La clase JuegoDeDados define tres propiedades de la clase Dado:

```
class JuegoDeDados {
    val dado1 = Dado(1)
    val dado2 = Dado(1)
    val dado3 = Dado(1)
```

En el método jugar de la clase JuegoDeDados procedemos a pedir a cada dado que se tire y verificamos si los tres valores son iguales:

```
    fun jugar() {
        dado1.tirar()
        dado2.tirar()
        dado3.tirar()
        if (dado1.valor == dado2.valor && dado2.valor == dado3.valor)
            println("Ganó")
        else
            print("Perdió")
    }
```

En la función main de nuestro programa creamos un objeto de la clase JuegoDeDados:

```
fun main(parametro: Array<String>) {
    val juego1 = JuegoDeDados()
    juego1.jugar()
}
```

Problema propuesto

- Plantear una clase Club y otra clase Socio.
  La clase Socio debe tener los siguientes propiedades: nombre y la antigüedad en el club (en años).
  Al constructor de la clase socio hacer que llegue el nombre y su antigüedad.
  La clase Club debe tener como propiedades 3 objetos de la clase Socio.
  Definir un método en la clase Club para imprimir el nombre del socio con mayor antigüedad en el club.

Solución

Retornar (index.php?inicio=15)

27 - POO - modificadores de acceso private y public

Uno de los principios fundamentales de la programación orientada a objetos es el encapsulamiento, esto se logra agrupando una serie de métodos y propiedades dentro de una clase.

En Kotlin cuando implementamos una clase por defecto todas las propiedades y métodos son de tipo public. Un método o propiedad public se puede acceder desde donde definimos un objeto de dicha clase.

En el caso que necesitemos definir métodos y propiedades que solo se puedan acceder desde dentro de la clase las debemos definir con el modificador private.

Problema 1

Plantear una clase Operaciones que en un método solicite la carga de 2 enteros y posteriormente llame desde el mismo método a otros dos métodos privados que calculen su suma y producto.

Proyecto122 - Principal.kt

```
class Operaciones {
    private var valor1: Int = 0
    private var valor2: Int = 0

    fun cargar() {
        print("Ingrese primer valor:")
        valor1 = readLine()!!.toInt()
        print("Ingrese segundo valor:")
        valor2 = readLine()!!.toInt()
        sumar()
        restar()
    }

    private fun sumar() {
        val suma = valor1 + valor2
        println("La suma de $valor1 y $valor2 es $suma")
    }

    private fun restar() {
        val resta = valor1 - valor2
        println("La resta de $valor1 y $valor2 es $resta")
    }
}

fun main(parametro: Array<String>) {
    val operaciones1 = Operaciones()
    operaciones1.cargar()
}
```

Un método se lo define como privado antecediendo la palabra clave private:

```
    private fun sumar() {
        val suma = valor1 + valor2
        println("La suma de $valor1 y $valor2 es $suma")
    }
```

Luego si queremos acceder a dicho método desde donde definimos un objetos se genera un error sintáctico:

```kotlin
class Operaciones {
    private var valor1: Int = 0
    private var valor2: Int = 0

    fun cargar() {
        print("Ingrese primer valor:")
        valor1 = readLine()!!.toInt()
        print("Ingrese segundo valor:")
        valor2 = readLine()!!.toInt()
        sumar()
        restar()
    }

    private fun sumar() {
        val suma = valor1 + valor2
        println("La suma de $valor1 y $valor2 es $suma")
    }

    private fun restar() {
        val resta = valor1 - valor2
        println("La resta de $valor1 y $valor2 es $resta")
    }
}

fun main(parametro: Array<String>) {
    val operaciones1 = Operaciones()
    operaciones1.cargar()
    operaciones1.sumar()
}
```

Cannot access 'sumar': it is private in 'Operaciones'

Lo mismo si queremos ocultar el acceso de una propiedad desde fuera de la clase debemos anteceder la palabra clave private:

```kotlin
    private var valor1: Int = 0
    private var valor2: Int = 0
```

La palabra clave public no es necesaria agregarla a un método o propiedad ya que es el valor por defecto que toma el compilador de Kotlin cuando las definimos. No genera error pero es redundante entonces escribir dicho modificador:

```
1      class Operaciones {
2          private var valor1: Int = 0
3          private var valor2: Int = 0
4
5          public fun cargar() {
           ...rimer valor:")
edundant visibility modifier more... (Ctrl+F1) e()!!.toInt()
8              print("Ingrese segundo valor:")
9              valor2 = readLine()!!.toInt()
10             sumar()
11             restar()
12         }
13
```

Problema 2

Plantear una clase llamada Dado. Definir una propiedad privada llamada valor y tres métodos uno privado que dibuje una línea de asteríscos y otro dos públicos, uno que genere un número aleatorio entre 1 y 6 y otro que lo imprima llamando en este último al que dibuja la línea de asteríscos.

Proyecto123 - Principal.kt

```
class Dado{
    private var valor: Int = 1
    fun tirar() {
        valor = ((Math.random() * 6) + 1).toInt()
    }

    fun imprimir() {
        separador()
        println("Valor del dado: $valor")
        separador()
    }

    private fun separador() = println("**************************************")
}

fun main(parametro: Array<String>) {
    val dado1 = Dado()
    dado1.tirar()
    dado1.imprimir()
}
```

Definimos la propiedad valor de tipo private:

```
    private var valor: Int = 1
```

El método separador también lo definimos private:

```
    private fun separador() = println("**************************************")
```

Desde la función main donde definimos un objeto de la clase Dado solo podemos acceder a los métodos tirar e imprimir:

```
fun main(parametro: Array<String>) {
    val dado1 = Dado()
    dado1.tirar()
    dado1.imprimir()
}
```

Problema propuesto

- Desarrollar una clase que defina una propiedad privada de tipo arreglo de 5 enteros.
  En el bloque init llamar a un método privado que cargue valores aleatorios comprendidos entre 0 y 10.
  Definir otros tres métodos públicos que muestren el arreglo, el mayor y el menor elemento.

Solución

Retornar (index.php?inicio=15)

28 - POO - propiedades y sus métodos opcionales set y get

Hemos visto que cuando definimos una propiedad pública podemos acceder a su contenido para modificarla o consultarla desde donde definimos un objeto.

A una propiedad podemos asociarle un método llamado set en el momento que se le asigne un valor y otro método llamado get cuando se accede al contenido de la propiedad.

Estos métodos son opcionales y nos permiten validar el dato a asignar a la propiedad o el valor de retorno.

Cuando no se implementan estos métodos el mismo compilador crea estos dos métodos por defecto.

Problema 1

Declarar una clase llamada persona con dos propiedades que almacenen el nombre y la edad de la persona. En la propiedad nombre almacenar siempre en mayúscula el nombre y cuando se recupere su valor retornarlo entre paréntesis, también controlar que no se pueda ingresar una edad con valor negativo, en dicho caso almacenar un cero.

Proyecto125 - Principal.kt

```
class Persona {
    var nombre: String = ""
    set(valor) {
        field = valor.toUpperCase()
    }
    get() {
        return "(" + field + ")"
    }

    var edad: Int = 0
    set(valor) {
        if (valor >= 0)
            field = valor
        else
            field = 0
    }
}

fun main(parametro: Array<String>) {
    val persona1 = Persona()
    persona1.nombre = "juan"
    persona1.edad = 23
    println(persona1.nombre)   // Se imprime:  (JUAN)
    println(persona1.edad)      // Se imprime:  23
    persona1.edad = -50
    println(persona1.edad)      // Se imprime:  0
}
```

Primero definimos la propiedad nombre de tipo String:

```
    var nombre: String = ""
```

En la línea inmediatamente siguiente definimos el método set (que no lleva la palabra clave fun) y que tiene un parámetro que no se indica el tipo porque es del mismo tipo que la propiedad nombre. Dentro del método set podemos modificar el contenido de la propiedad nombre mediante la palabra clave field (en este problema guardamos el dato que llega pero convertido a mayúsculas):

```
    set(valor) {
        field = valor.toUpperCase()
    }
```

En el caso que la propiedad implemente el método get también lo hacemos antes o después del método set si lo tiene:

```
    get() {
        return "(" + field + ")"
    }
```

La propiedad edad solo implementa el método set donde verificamos si el parámetro que recibe es negativo almacenamos en la propiedad el valor 0 sino almacenamos el valor tal cual llega.

También es muy importante entender que cuando definimos un objeto de la clase persona y le asignamos un valor a una propiedad en realidad se está ejecutando un método:

```
    val persona1 = Persona()
    persona1.nombre = "juan"  //se ejecuta el método set de la propiedad nombre
```

Luego de esta asignación en la propiedad nombre se almacenó el String "JUAN" en mayúsculas ya que eso hace el método set.

En la asignación:

```
    persona1.edad = 23
```

se almacena el número 23 ya que es mayor o igual a cero.

Podemos comprobar los valores de las propiedades:

```
    println(persona1.nombre)   // Se imprime:  (JUAN)
    println(persona1.edad)      // Se imprime:  23
```

Si tratamos de fijar un valor menor a cero en la propiedad edad:

```
    persona1.edad = -50
```

Luego no se carga ya que en el método set verificamos el dato que llega y cargamos en caso que sea negativo el valor cero:

```
    println(persona1.edad)      // Se imprime:  0
```

Problemas propuestos

- Confeccionar una clase que represente un Empleado. Definir como propiedades su nombre y su sueldo.
  No permitir que se cargue un valor negativo en su sueldo.
  Codificar el método imprimir en la clase.

- Plantear una clase llamada Dado. Definir una propiedad llamada valor que permita cargar un valor comprendido entre 1 y 6 si llega un valor que no está comprendido en

este rango se debe cargar un 1.
Definir dos métodos, uno que genere un número aleatorio entre 1 y 6 y otro que lo imprima.
Al constructor llega el valor inicial que debe tener el dado (tratar de enviarle el número 7)

Solución

Retornar (index.php?inicio=15)

29 - POO - data class

Hemos dicho que una clase encapsula un conjunto de funcionalidades (métodos) y datos (propiedades)

En muchas situaciones queremos almacenar un conjunto de datos sin necesidad de implementar funcionalidades, en estos casos el lenguaje Kotlin nos provee de una estructura llamada: data class.

Problema 1

Declarar un data class llamado Articulo que almacene el código del producto, su descripción y precio. Definir luego varios objetos de dicha data class en la main.

Proyecto128 - Principal.kt

```
data class Articulo(var codigo: Int, var descripcion: String, var precio: Float)

fun main(parametro: Array<String>) {
    val articulo1 = Articulo(1, "papas", 34f)
    var articulo2 = Articulo(2, "manzanas", 24f)
    println(articulo1)
    println(articulo2)
    val puntero = articulo1
    puntero.precio = 100f
    println(articulo1)
    var articulo3 = articulo1.copy()
    articulo1.precio = 200f
    println(articulo1)
    println(articulo3)
    if (articulo1 == articulo3)
        println("Son iguales $articulo1 y $articulo3")
    else
        println("Son distintos $articulo1 y $articulo3")
    articulo3.precio = 200f
    if (articulo1 == articulo3)
        println("Son iguales $articulo1 y $articulo3")
    else
        println("Son distintos $articulo1 y $articulo3")
}
```

La sintaxis para la declaración de un data class es:

```
data class Articulo(var codigo: Int, var descripcion: String, var precio: Floa
t)
```

Le antecedemos la palabra clave data y en el constructor definimos las propiedades que contiene dicho data class.

Para definir objetos de un data class es idéntico a la definición de objetos de una clase común:

```
fun main(parametro: Array<String>) {
    val articulo1 = Articulo(1, "papas", 34f)
    var articulo2 = Articulo(2, "manzanas", 24f)
```

Si le pasamos a la función println una variable de tipo data class nos muestra el nombre del data class, los nombres de las propiedades y sus valores:

```
    println(articulo1) // Articulo(codigo=1, descripcion=papas, precio=34.0)
    println(articulo2) // Articulo(codigo=2, descripcion=manzanas, precio=24.0)
```

En realidad todo data class tiene una serie de métodos básicos: toString, copy etc., luego cuando pasamos el data class a la función println lo que sucede es que dicha función llama al método toString, el mismo resultado por pantalla tenemos si escribimos:

```
    println(articulo1.toString()) // Articulo(codigo=1, descripcion=papas, prec
io=34.0)
```

Podemos asignar a una variable un objeto de un determinado data class:

```
    val puntero = articulo1
```

Luego la variable puntero tiene la referencia al mismo objeto referenciado por articulo1, si cambiamos la propiedad precio mediante la variable puntero:

```
    puntero.precio = 100f
```

El contenido de articulo1 ahora es:

```
    println(articulo1) // Articulo(codigo=1, descripcion=papas, precio=100.0)
```

Para obtener una copia de un objeto de tipo data class debemos llamar al método copy:

```
    var articulo3 = articulo1.copy()
    articulo1.precio = 200f
    println(articulo1) // Articulo(codigo=1, descripcion=papas, precio=200.0)
    println(articulo3) // Articulo(codigo=1, descripcion=papas, precio=100.0)
```

La variable articulo3 apunta a un objeto distinto a la variable articulo1. Esto se ve cuando modificamos la propiedad precio del articulo1 no se refleja en la propiedad precio de articulo3.

Cuando utilizamos el operador == en objetos de tipo data class se verifica verdadero si los contenidos de todas sus propiedades tienen almacenado igual valor:

```
    if (articulo1 == articulo3)
        println("Son iguales $articulo1 y $articulo3")
    else
        println("Son distintos $articulo1 y $articulo3")
    articulo3.precio = 200f
    if (articulo1 == articulo3)
        println("Son iguales $articulo1 y $articulo3")
    else
        println("Son distintos $articulo1 y $articulo3")
```

Redefinición de métodos de un data class.

Hemos dicho que al declarar un data class ya heredamos una serie de métodos que nos son útiles para procesar luego los objetos que definamos de dicho data class.

En Kotlin podemos sobreescribir cualquiera de los métodos que nos provee un data class y definir un nuevo algoritmo al mismo.

Problema 2

Declarar un data class llamado Persona que almacene el nombre y la edad. Sobreescribir el método toString para retornar un String con la concatenación del nombre y la edad separadas por una coma.

Proyecto129 - Principal.kt

```
data class Persona(var nombre: String, var edad: Int) {
    override fun toString(): String {
        return "$nombre, $edad"
    }
}

fun main(parametro: Array<String>) {
    var persona1 = Persona("Juan", 22)
    var persona2 = Persona("Ana", 59)
    println(persona1)
    println(persona2)
}
```

Declaramos el data class Persona e implementamos el método toString que como todo data class ya tiene dicho método debemos anteceder al nombre del método la palabra clave override:

```
data class Persona(var nombre: String, var edad: Int) {
    override fun toString(): String {
        return "$nombre, $edad"
    }
}
```

En la main definimos dos objetos del tipo data class Persona:

```
fun main(parametro: Array<String>) {
    var persona1 = Persona("Juan", 22)
    var persona2 = Persona("Ana", 59)
```

Cuando llamamos a la función println y le pasamos persona1 luego se ejecuta el método toString que implementamos nosotros y nos muestra:

```
    println(persona1) // Juan, 22
    println(persona2) // Ana, 59
```

Recordemos que podemos hacer que el método toString sea más concisa implementando una función con una única expresión:

```
    override fun toString() = "$nombre, $edad"
```

Problema propuesto

- Plantear un data class llamado Dado con una única propiedad llamada valor. Sobreescribir el método toPrint para que muestre tantos asteriscos como indica la propiedad valor.

Solución

Retornar (index.php?inicio=15)

30 - POO - enum class

En Kotlin tenemos otro tipo especial de clase que se las declara con las palabras claves enum class.

Se las utiliza para definir un conjunto de constantes.

Problema 1

Declarar una enum class con los nombres de naipes de la baraja inglesa.
Definir una clase carta que tenga una propiedad de la clase enum class.

Proyecto131 - Principal.kt

```
enum class TipoCarta{
    DIAMANTE,
    TREBOL,
    CORAZON,
    PICA
}

class Carta(val tipo: TipoCarta, val valor: Int) {

    fun imprimir() {
        println("Carta: $tipo y su valor es $valor")
    }
}

fun main(parametro: Array<String>) {
    val carta1 = Carta(TipoCarta.TREBOL, 4)
    carta1.imprimir()
}
```

Para declarar un enum class indicamos cada una de las constantes posibles que se podrán guardar:

```
enum class TipoCarta{
    DIAMANTE,
    TREBOL,
    CORAZON,
    PICA
}
```

La clase Carta tiene dos propiedades, la primera llamada tipo que es de el enum class TipoCarta y la segunda el valor de la carta:

```
class Carta(val tipo: TipoCarta, val valor: Int) {
```

El método imprimir muestra el contenido de la propiedad tipo y la propiedad valor:

```
    fun imprimir() {
        println("Carta: $tipo y su valor es $valor")
    }
```

En la función main creamos un objeto de la clase Carta y le pasamos en el primer parámetro alguna de las cuatro constantes definidas dentro del enum class TipoCarta:

```
fun main(parametro: Array<String>) {
    val carta1 = Carta(TipoCarta.TREBOL, 4)
    carta1.imprimir()
}
```

Propiedades en un enum class

Igual que las clases comunes las clases enum class pueden tener propiedades definidas en el constructor. Luego debemos indicar un valor para cada una de esas propiedades en las constantes definidas.

Problema 2

Declarar un enum class que represente las cuatro operaciones básicas, asociar a cada constante un String con el signo de la operación.
Definir una clase Operación que defina tres propiedades, las dos primeras deben ser los números y la tercera el tipo de operación.

Proyecto132 - Principal.kt

```
enum class TipoOperacion (val tipo: String) {
    SUMA("+"),
    RESTA("-"),
    MULTIPLICACION("*"),
    DIVISION("/")
}

class Operacion (val valor1: Int, val valor2: Int, val tipoOperacion: TipoOperac
ion) {

    fun operar() {
        var resultado: Int = 0
        when (tipoOperacion) {
            TipoOperacion.SUMA -> resultado = valor1 + valor2
            TipoOperacion.RESTA -> resultado = valor1 - valor2
            TipoOperacion.MULTIPLICACION -> resultado = valor1 * valor2
            TipoOperacion.DIVISION -> resultado = valor1 / valor2
        }
        println("$valor1 ${tipoOperacion.tipo} $valor2 es igual a $resultado")
    }
}

fun main(parametro: Array<String>) {
    val operacion1 = Operacion(10, 4, TipoOperacion.SUMA)
    operacion1.operar()
}
```

Hemos declarado la clase TipoOperación con cuatro constantes llamadas: SUMA, RESTA, MULTIPLICACION y DIVISION.
Cada constante tiene asociado entre paréntesis el valor de la propiedad tipo que es un String:

```
enum class TipoOperacion (val tipo: String) {
    SUMA("+"),
    RESTA("-"),
    MULTIPLICACION("*"),
    DIVISION("/")
}
```

Declaramos la clase Operacion que recibe dos enteros y un objeto de tipo TipoOperacion:

```
class Operacion (val valor1: Int, val valor2: Int, val tipoOperacion: TipoOpera
cion) {
```

El método operar según el valor almacenado en la propiedad tipoOperacion almacena en la variable resultado el valor respectivo:

```
    fun operar() {
        var resultado: Int = 0
        when (tipoOperacion) {
            TipoOperacion.SUMA -> resultado = valor1 + valor2
            TipoOperacion.RESTA -> resultado = valor1 - valor2
            TipoOperacion.MULTIPLICACION -> resultado = valor1 * valor2
            TipoOperacion.DIVISION -> resultado = valor1 / valor2
        }
```

Para imprimir el operador utilizado ("+", "-" etc.) podemos acceder a la propiedad tipo del objeto tipoOperacion:

```
        println("$valor1 ${tipoOperacion.tipo} $valor2 es igual a $resultado")
```

En la función main definimos un objeto de la clase Operacion y le pasamos los dos enteros a operar y el tipo de operación a efectuar:

```
fun main(parametro: Array<String>) {
    val operacion1 = Operacion(10, 4, TipoOperacion.SUMA)
    operacion1.operar()
}
```

Acotaciones

Otros dato importante que podemos extraer de un enum class es la posición de una constante en la enumeración mediante la propiedad ordinal:

```
enum class TipoOperacion (val tipo: String) {
    SUMA("+"),
    RESTA("-"),
    MULTIPLICACION("*"),
    DIVISION("/")
}

fun main(parametro: Array<String>) {
    val tipo1 = TipoOperacion.MULTIPLICACION
    println(tipo1)         // MULTIPLICACION
    println(tipo1.tipo)    // *
    println(tipo1.ordinal) // 2
}
```

Con la propiedad ordinal nos retorna la posición de la constante dentro del enum class (la primer constante ocupa la posición 0, la segunda la posición 2 y así sucesivamente.

Si necesitamos recorrer mediante un for todos las constantes contenidas en un enum class la sintaxis es la siguiente:

```
enum class TipoOperacion (val tipo: String) {
    SUMA("+"),
    RESTA("-"),
    MULTIPLICACION("*"),
    DIVISION("/")
}

fun main(parametro: Array<String>) {
    for(elemento in TipoOperacion.values()) {
        println(elemento)
    }
}
```

Mediante valueOf podemos pasar un String con una constante y nos retorna una referencia a dicha constante:

```
enum class TipoOperacion (val tipo: String) {
    SUMA("+"),
    RESTA("-"),
    MULTIPLICACION("*"),
    DIVISION("/")
}

fun main(parametro: Array<String>) {
    val tipo1 = TipoOperacion.valueOf("RESTA")
    println(tipo1)         // RESTA
    println(tipo1.tipo)    // -
    println(tipo1.ordinal) // 1
}
```

Como cada constante es un objeto podemos sobreescribr el método toString y hacer que retorne una cadena distinta al nombre de la constante:

```
enum class TipoOperacion (val tipo: String) {
    SUMA("+") {
        override fun toString() = "operación suma"
    },
    RESTA("-"){
        override fun toString() = "operación resta"
    },
    MULTIPLICACION("*"),
    DIVISION("/")
}

fun main(parametro: Array<String>) {
    val tipo1 = TipoOperacion.SUMA
    println(tipo1)         // operación suma
    println(tipo1.tipo)    // +
    println(tipo1.ordinal) // 0
}
```

Problema propuesto

- Definir un enum class almacenando como constante los nombres de varios países sudamericanos y como propiedad para cada país la cantidad de habitantes que tiene. Definir una variable de este tipo e imprimir la constante y la cantidad de habitantes de dicha variable.

Solución

Retornar (index.php?inicio=15)

31 - POO - objeto nombrado

Otra característica del lenguaje Kotlin es poder definir un objeto en forma inmediata sin tener que declarar una clase.

Aparece una nueva palabra clave object con la que podemos crear estos objetos en forma directa.

Problema 1

Definir un objeto llamada Matematica con una propiedad que almacene el valor de PI y un método que retorne un valor aleatorio en un determinado rango.

Proyecto134 - Principal.kt

```
object Matematica {
    val PI = 3.1416
    fun aleatorio(minimo: Int, maximo: Int) = ((Math.random() * (maximo + 1 - mi
nimo)) + minimo).toInt()
}

fun main(parametro: Array<String>) {
    println("El valor de Pi es ${Matematica.PI}")
    print("Un valor aleatorio entre 5 y 10: ")
    println(Matematica.aleatorio(5, 10))
}
```

Para definir un objeto sin tener que declarar una clase utilizamos la palabra clave object y seguidamente el nombre de objeto que vamos a crear, luego entre llaves definimos sus propiedades y métodos(en nuestro ejemplo definimos una propiedad y un método) :

```
object Matematica {
    val PI = 3.1416
    fun aleatorio(minimo: Int, maximo: Int) = ((Math.random() * (maximo + 1 - m
inimo)) + minimo).toInt()
}
```

Para acceder a la propiedad o el método definido en el objeto Matematica lo hacemos antecediendo el nombre del objeto al método o propiedad que tenemos que llamar:

```
fun main(parametro: Array<String>) {
    println("El valor de Pi es ${Matematica.PI}")
    print("Un valor aleatorio entre 5 y 10: ")
    println(Matematica.aleatorio(5, 10))
}
```

Objetos locales a una función o método.

Para poder definir objetos nombrados que no sean globales como el caso anterior sino que estén definidos dentro de una función lo debemos hacer definiendo una variable local y un objeto anónimo (es decir sin nombre)

Problema 2

Crear un objeto local en la función main que permita tirar 5 dados y almacenar dichos valores en un arreglo. Definir una propiedad que almacene 5 enteros y tres métodos: uno que cargue los 5 elementos del arreglo con valores aleatorios comprendidos entre 1 y 6, otro que imprima el arreglo y finalmente otro que retorne el mayor valor del arreglo.

Proyecto135 - Principal.kt

```
fun main(parametro: Array<String>) {
    val dados = object {
        val arreglo = IntArray(5)
        fun generar() {
            for(i in arreglo.indices)
                arreglo[i] = ((Math.random() * 6) + 1).toInt()
        }

        fun imprimir() {
            for(elemento in arreglo)
                print("$elemento - ")
            println();
        }

        fun mayor(): Int {
            var may = arreglo[0]
            for(i in arreglo.indices)
                if (arreglo[i] > may)
                    may = arreglo[i]
            return may
        }
    }

    dados.generar()
    dados.imprimir()
    print("Mayor valor:")
    println(dados.mayor())
}
```

Para definir un objeto local a una función debe ser anónimo, es decir no disponemos un nombre después de la palabra clave object.
Lo que debemos hacer es asignar el valor devuelto por object a una variable:

```
    val dados = object {
```

Dentro de las llaves de object definimos las propiedades e implementamos sus métodos:

```
        fun generar() {
            for(i in arreglo.indices)
                arreglo[i] = ((Math.random() * 6) + 1).toInt()
        }

        fun imprimir() {
            for(elemento in arreglo)
                print("$elemento - ")
            println();
        }

        fun mayor(): Int {
            var may = arreglo[0]
            for(i in arreglo.indices)
                if (arreglo[i] > may)
                    may = arreglo[i]
            return may
        }
    }
```

Luego podemos acceder a las propiedades y sus métodos antecediendo el nombre de la variable que se carga al llamar object:

```
    dados.generar()
    dados.imprimir()
    print("Mayor valor:")
    println(dados.mayor())
```

Problema propuesto

- Definir un objeto nombrado:

  ```
  object Mayor {
  ```

  con tres métodos llamados "maximo" con dos parámetros cada uno. Los métodos difieren en que uno recibe tipos de datos Int, otro de tipos Float y finalmente el último recibe tipos de datos Double. Los tres métodos deben retornar el mayor de los dos datos que reciben.

Solución

Retornar (index.php?inicio=30)

32 - POO - herencia

Vimos en conceptos anteriores que dos clases pueden estar relacionadas por la colaboración (en una de ellas definimos una propiedad del tipo de la otra clase). Ahora veremos otro tipo de relación entre clases que es la Herencia.

La herencia significa que se pueden crear nuevas clases partiendo de clases existentes, que tendrá todas las propiedades y los métodos de su 'superclase' o 'clase padre' y además se le podrán añadir otras propiedades y métodos propios.

clase padre

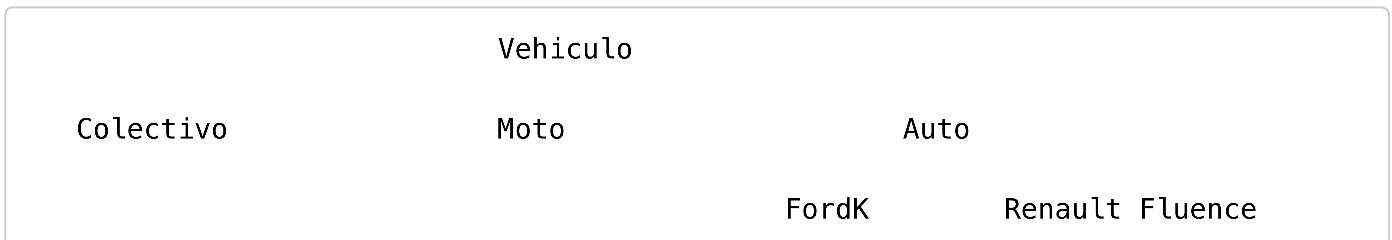
Clase de la que desciende o deriva una clase. Las clases hijas (descendientes) heredan (incorporan) automáticamente las propiedades y métodos de la la clase padre.

Subclase

Clase descendiente de otra. Hereda automáticamente los atributos y métodos de su superclase. Es una especialización de otra clase.
Admiten la definición de nuevos atributos y métodos para aumentar la especialización de la clase.

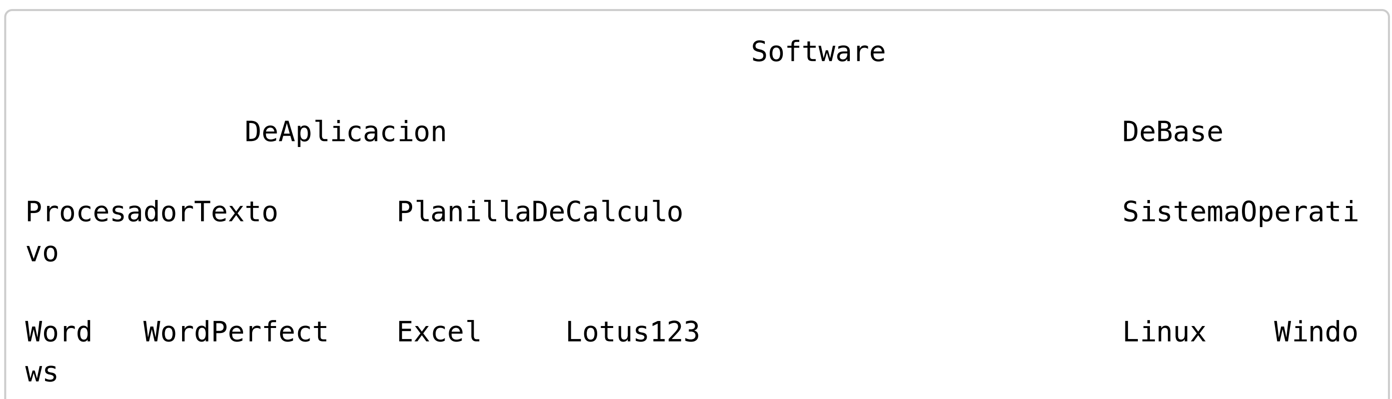
Veamos algunos ejemplos teóricos de herencia:

1) Imaginemos la clase Vehículo. Qué clases podrían derivar de ella?



Siempre hacia abajo en la jerarquía hay una especialización (las subclases añaden nuevas propiedades y métodos.

2) Imaginemos la clase Software. ¿Qué clases podrían derivar de ella?



El primer tipo de relación que habíamos visto entre dos clases, es la de colaboración. Recordemos que es cuando una clase contiene un objeto de otra clase como atributo. Cuando la relación entre dos clases es del tipo "...tiene un..." o "...es parte de...", no debemos implementar herencia. Estamos frente a una relación de colaboración de clases no de herencia.

Si tenemos una ClaseA y otra ClaseB y notamos que entre ellas existe una relacion de tipo "... tiene un...", no debe implementarse herencia sino declarar en la clase ClaseA un atributo de la clase ClaseB.

Por ejemplo: tenemos una clase Auto, una clase Rueda y una clase Volante. Vemos que la relación entre ellas es: Auto "...tiene 4..." Rueda, Volante "...es parte de..." Auto; pero la clase Auto no debe derivar de Rueda ni Volante de Auto porque la relación no es de tipo-subtipo sino de colaboración. Debemos declarar en la clase Auto 4 atributos de tipo Rueda y 1 de tipo Volante.

Luego si vemos que dos clase responden a la pregunta ClaseA "..es un.." ClaseB es posible que haya una relación de herencia.

Por ejemplo:

```
Auto "es un" Vehiculo
Circulo "es una" Figura
Mouse "es un" DispositivoEntrada
Suma "es una" Operacion
```

Problema 1:

Plantear una clase Persona que contenga dos propiedades: nombre y edad. Definir como responsabilidades el constructor que reciba el nombre y la edad.
En la función main del programa definir un objeto de la clase Persona y llamar a sus métodos.

Declarar una segunda clase llamada Empleado que herede de la clase Persona y agregue una propiedad sueldo y muestre si debe pagar impuestos (sueldo superior a 3000)
También en la función main del programa crear un objeto de la clase Empleado.

Proyecto137 - Principal.kt

```
open class Persona(val nombre: String, val edad: Int) {
    open fun imprimir() {
        println("Nombre: $nombre")
        println("Edad: $edad")
    }
}

class Empleado(nombre: String, edad: Int, val sueldo: Double): Persona(nombre, e
dad) {
    override fun imprimir() {
        super.imprimir()
        println("Sueldo: $sueldo")
    }

    fun pagaImpuestos() {
        if (sueldo > 3000)
            println("El empleado $nombre paga impuestos")
        else
            println("El empleado $nombre no paga impuestos")
    }
}

fun main(parametro: Array<String>) {
    val persona1 = Persona("Jose", 22)
    println("Datos de la persona")
    persona1.imprimir()

    val empleado1 = Empleado("Ana", 30, 5000.0)
    println("Datos del empleado")
    empleado1.imprimir()
    empleado1.pagaImpuestos()
}
```

En Kotlin para que una clase sea heredable debemos anteceder la palabra clave open previo a class:

```
open class Persona(val nombre: String, val edad: Int) {
```

Cuando un programador defina una clase open debe pensar seriamente la definición de sus propiedades y métodos.

Si queremos que un método se pueda reescribir en una subclase debemos anteceder la palabra clave open:

```
    open fun imprimir() {
        println("Nombre: $nombre")
        println("Edad: $edad")
    }
```

Cuando definimos un objeto de la clase Persona en la función main por más que sea open y tenga métodos open no cambia en nada a como definimos objetos y accedemos a sus propiedades y métodos:

```
fun main(parametro: Array<String>) {
    val persona1 = Persona("Jose", 22)
    println("Datos de la persona")
    persona1.imprimir()
```

Lo nuevo aparece cuando declaramos la clase Empleado que recibe tres parámetros, es importante notar que en el tercero estamos definiendo una propiedad llamada sueldo (porque antecedemos la palabra clave val)

La herencia la indicamos después de los dos puntos indicando el nombre de la clase de la cual heredamos y pasando inmediatamente los datos del constructor de dicha clase:

```
class Empleado(nombre: String, edad: Int, val sueldo: Double): Persona(nombre,
edad) {
```

Podemos decir que la clase Empleado tiene tres propiedades (nombre, edad y sueldo), una propia y dos heredadas.

Como no le antecedemos la palabra clave open a la declaración de la clase luego no se podrán declarar clases que hereden de la clase Empleado. Si queremos que de la clase Empleado puedan heredar otras clases la debemos definir con la sintaxis:

```
open class Empleado(nombre: String, edad: Int, val sueldo: Double): Persona(nom
bre, edad) {
```

La clase Empleado aparte del constructor define dos métodos, uno que imprime un mensaje si debe parar impuestos el empleado:

```
    fun pagaImpuestos() {
        if (sueldo > 3000)
            println("El empleado $nombre paga impuestos")
        else
            println("El empleado $nombre no paga impuestos")
    }
```

El método imprimir indicamos mediante la palabra clave override que estamos sobreescribiendo el método imprimir de la clase persona (dentro del método imprimir podemos llamar al método imprimir de la clase padre antecediendo la palabra clave super):

```
    override fun imprimir() {
        super.imprimir()
        println("Sueldo: $sueldo")
    }
```

Con la llamada al método imprimir de la clase que hereda podemos mostrar todos los datos del empleado que son su nombre, edad y sueldo.

Para definir un objeto de la clase Empleado en la función main lo hacemos con la sintaxis que ya conocemos:

```
    val empleado1 = Empleado("Ana", 30, 5000.0)
    println("Datos del empleado")
    empleado1.imprimir()
    empleado1.pagaImpuestos()
```

Problema 2:

Declarar una clase llamada Calculadora que reciba en el constructor dos valores de tipo Double. Hacer la clase abierta para que sea heredable
Definir las responsabilidades de sumar, restar, multiplicar, dividir e imprimir.

Declarar luego una clase llamada CalculadoraCientifica que herede de Calculadora y añada las responsabilidades de calcular el cuadrado del primer número y la raíz cuadrada.

Proyecto138 - Principal.kt

```
open class Calculadora(val valor1: Double, val valor2: Double ){
    var resultado: Double = 0.0
    fun sumar() {
        resultado = valor1 + valor2
    }

    fun restar() {
        resultado = valor1 - valor2
    }

    fun multiplicar() {
        resultado = valor1 * valor2
    }

    fun dividir() {
        resultado = valor1 / valor2
    }

    fun imprimir() {
        println("Resultado: $resultado")
    }
}

class CalculadoraCientifica(valor1: Double, valor2: Double): Calculadora(valor1,
valor2) {
    fun cuadrado() {
        resultado = valor1 * valor1
    }

    fun raiz() {
        resultado = Math.sqrt(valor1)
    }
}

fun main(parametro: Array<String>) {
    println("Prueba de la clase Calculadora (suma de dos números)")
    val calculadora1 = Calculadora(10.0, 2.0)
    calculadora1.sumar()
    calculadora1.imprimir()
    println("Prueba de la clase Calculadora Cientrífica (suma de dos números y e
l cuadrado y la raiz del primero)")
    val calculadoraCientifica1 = CalculadoraCientifica(10.0, 2.0)
    calculadoraCientifica1.sumar()
    calculadoraCientifica1.imprimir()
    calculadoraCientifica1.cuadrado()
    calculadoraCientifica1.imprimir()
    calculadoraCientifica1.raiz()
```

```
    calculadoraCientifica1.imprimir()
}
```

Declaramos la clase Calculadora open para permitir que otras clases hereden de esta, llegan al constructor dos valores de tipo Double:

```
open class Calculadora(val valor1: Double, val valor2: Double ){
```

Definimos una tercer propiedad llamada resultado también de tipo Double:

```
    var resultado: Double = 0.0
```

Es importante tener en cuenta que la propiedad resultado es public, luego podemos tener acceso en sus subclases. Si la definimos de tipo private luego la subclase CalculadoraCientifica no puede acceder a su contenido:

```
    private var resultado: Double = 0.0
```

En la clase CalculadoraCientifica se genera un error sintáctico cuando queremos acceder a dicha propiedad:

```
    fun cuadrado() {
        resultado = valor1 * valor1
    }
```

Vimos que los modificadores de acceso pueden ser public (este es el valor por defecto) y private, existe un tercer modificador de acceso llamado protected que permite que una subclase tenga acceso a la propiedad pero no se tenga acceso desde donde definimos un objeto de dicha clase. Luego lo más conveniente para este problema es definir la propiedad resultado de tipo protected:

```
    protected var resultado: Double = 0.0
```

La clase Calculadora define los cuatro métodos con las operaciones básicas y la impresión del resultado.

La clase CalculadoraCientifica hereda de la clase Calculadora:

```
class CalculadoraCientifica(valor1: Double, valor2: Double): Calculadora(valor
1, valor2) {
```

Una calculadora científica aparte de las cuatro operaciones básicas agrega la posibilidad de calcular el cuadrado y la raíz de un número:

```
    fun cuadrado() {
        resultado = valor1 * valor1
    }

    fun raiz() {
        resultado = Math.sqrt(valor1)
    }
```

En la función main creamos un objeto de la clase Calculadora y llamamos a varios de sus métodos:

```
fun main(parametro: Array<String>) {
    println("Prueba de la clase Calculadora (suma de dos números)")
    val calculadora1 = Calculadora(10.0, 2.0)
    calculadora1.sumar()
    calculadora1.imprimir()
```

Lo mismo hacemos definiendo un objeto de la clase CalculadoraCientifica y llamando a algunos de sus métodos:

```
    println("Prueba de la clase Calculadora Cientrífica (suma de dos números y el cuadrado y la raiz del primero)")
    val calculadoraCientifica1 = CalculadoraCientifica(10.0, 2.0)
    calculadoraCientifica1.sumar()
    calculadoraCientifica1.imprimir()
    calculadoraCientifica1.cuadrado()
    calculadoraCientifica1.imprimir()
    calculadoraCientifica1.raiz()
    calculadoraCientifica1.imprimir()
```

Problema propuesto

- Declarar una clase Dado que genere un valor aleatorio entre 1 y 6, mostrar su valor.
  Crear una segunda clase llamada DadoRecuadro que genere un valor entre 1 y 6, mostrar el valor recuadrado en asteríscos.
  Utilizar la herencia entre estas dos clases.

Solución

Retornar (index.php?inicio=30)

33 - POO - herencia - clases abstractas

En algunas situaciones tenemos métodos y propiedades comunes a un conjunto de clases, podemos agrupar dichos métodos y propiedades en una clase abstracta.

Hay una sintaxis especial en Kotlin para indicar que una clase es abstracta.
No se pueden definir objetos de una clase abstracta y seguramente será heredada por otras clases de las que si podremos definir objetos.

Problema 1:

Declarar una clase abstracta que represente una Operación. Definir en la misma tres propiedades valor1, valor2 y resultado, y dos métodos: calcular e imprimir.
Plantear dos clases llamadas Suma y Resta que hereden de la clase Operación.

Proyecto140 - Principal.kt

```
abstract class Operacion(val valor1: Int, val valor2: Int) {
    protected var resultado: Int = 0

    abstract fun operar()

    fun imprimir() {
        println("Resultado: $resultado")
    }
}

class Suma(valor1: Int, valor2: Int): Operacion(valor1, valor2) {
    override fun operar() {
        resultado = valor1 + valor2
    }
}

class Resta(valor1: Int, valor2: Int): Operacion(valor1, valor2) {
    override fun operar() {
        resultado = valor1 - valor2
    }
}

fun main(parametro: Array<String>) {
    val suma1 = Suma(10, 4)
    suma1.operar()
    suma1.imprimir()
    val resta1 = Resta(20, 5)
    resta1.operar()
    resta1.imprimir()
}
```

Una clase abstracta se la declara antecediendo la palabra clave abstract a la palabra clave class:

```
abstract class Operacion(val valor1: Int, val valor2: Int) {
```

Podemos definir propiedades que serán heredables:

```
    protected var resultado: Int = 0
```

Podemos declarar métodos abstractos que obligan a las clases que heredan de esta a su implementación (esto ayuda a unificar las funcionalidades de todas sus subclases, tiene sentido que toda clase que herede de la clase Operación implemente un método operar):

```
    abstract fun operar()
```

Una clase abstracta puede tener también métodos concretos, es decir implementados:

```
    fun imprimir() {
        println("Resultado: $resultado")
    }
```

La sintaxis para declarar que una clase hereda de una clase abstracta es lo mismo que vimos en el concepto anterior:

```
class Suma(valor1: Int, valor2: Int): Operacion(valor1, valor2) {
```

Lo que si es obligatorio implementar el método abstracto operar, en caso que no se lo haga aparece un error sintáctico:

```
    override fun operar() {
        resultado = valor1 + valor2
    }
```

En la función main podemos definir objetos de la clave Suma y Resta, pero no podemos definir objetos de la clase Operacion ya que la misma es abstracta:

```
fun main(parametro: Array<String>) {
    val suma1 = Suma(10, 4)
    suma1.operar()
    suma1.imprimir()
    val resta1 = Resta(20, 5)
    resta1.operar()
    resta1.imprimir()
}
```

Problema propuesto

- Declarar una clase abstracta Cuenta y dos subclases CajaAhorra y PlazoFijo. Definir las propiedades y métodos comunes entre una caja de ahorro y un plazo fijo y agruparlos en la clase Cuenta.
  Una caja de ahorro y un plazo fijo tienen un nombre de titular y un monto. Un plazo fijo añade un plazo de imposición en días y una tasa de interés. Hacer que la caja de ahorro no genera intereses.
  En la función main del programa definir un objeto de la clase CajaAhorro y otro de la clase PlazoFijo.

Solución

Retornar (index.php?inicio=30)

34 - POO - declaración e implementación de interfaces

Una interface declara una serie de métodos y propiedades que deben ser implementados luego por una o más clases, también una interface en Kotlin puede tener implementado métodos.

Las interfaces vienen a suplir la imposibilidad de herencia múltiple en Kotlin.

Se utiliza la misma sintaxis que la herencia para indicar que una clase implementa una interface.

Por ejemplo podemos tener dos clases que representen un avión y un helicóptero. Luego plantear una interface con un método llamado volar. Las dos clases pueden implementar dicha interface y codificar el método volar (los algoritmos seguramente sean distintos pero el comportamiento de volar es común tanto a un avión como un helicóptero)

La sintaxis en Kotlin para declarar una interface es:

```
interface [nombre de la interface] {
    [declaración de propiedades]
    [declaración de métodos]
    [implementación de métodos]
}
```

Problema 1

Definir una interface llamada Punto que declare un método llamado imprimir. Luego declarar dos clases que la implementen.

Proyecto142 - Principal.kt

```
interface Punto {
    fun imprimir()
}

class PuntoPlano(val x: Int, val y: Int): Punto {
    override fun imprimir() {
        println("Punto en el plano: ($x,$y)")
    }
}

class PuntoEspacio(val x: Int, val y: Int, val z: Int): Punto {
    override fun imprimir() {
        println("Punto en el espacio: ($x,$y,$z)")
    }
}

fun main(parametro: Array<String>) {
    val puntoPlano1 = PuntoPlano(10, 4)
    puntoPlano1.imprimir()
    val puntoEspacio1 = PuntoEspacio(20, 50, 60)
    puntoEspacio1.imprimir()
}
```

Para declarar una interface en Kotlin utilizamos la palabra clave interface y seguidamente su nombre. Luego entre llaves indicamos todas las cabeceras de métodos, propiedades y métodos ya implementados. En nuestro ejemplo declaramos la interface Punto e indicamos que quien la implemente debe definir un método llamado imprimir sin parámetros y que no retorna nada:

```
interface Punto {
    fun imprimir()
}
```

Por otro lado declaramos dos clases llamados PuntoPlano con dos propiedades y PuntoEspacio con tres propiedades, además indicamos que dichas clases implementarán la interface Punto:

```
class PuntoPlano(val x: Int, val y: Int): Punto {
    override fun imprimir() {
        println("Punto en el plano: ($x,$y)")
    }
}

class PuntoEspacio(val x: Int, val y: Int, val z: Int): Punto {
    override fun imprimir() {
        println("Punto en el espacio: ($x,$y,$z)")
    }
}
```

La sintaxis para indicar que una clase implementa una interface es igual a la herencia. Si una clase hereda de otra también puede implementar una o más interfaces separando por coma cada una de las interfaces.

El método imprimir en cada clase se implementa en forma distinta, en uno se imprimen 3 propiedades y en la otra se imprimen 2 propiedades.

Luego definimos en la función main un objeto de la clase PuntoPlano y otro de tipo PuntoEspacio:

```
fun main(parametro: Array<String>) {
    val puntoPlano1 = PuntoPlano(10, 4)
    puntoPlano1.imprimir()
    val puntoEspacio1 = PuntoEspacio(20, 50, 60)
    puntoEspacio1.imprimir()
}
```

Problema 2

Se tiene la siguiente interface:

```
interface Figura {
    fun calcularSuperficie(): Int
    fun calcularPerimetro(): Int
    fun tituloResultado() {
        println("Datos de la figura")
    }
}
```

Definir dos clases que representen un Cuadrado y un Rectángulo. Implementar la interface Figura en ambas clases.

Proyecto143 - Principal.kt

```
interface Figura {
    fun calcularSuperficie(): Int
    fun calcularperimetro(): Int
    fun tituloResultado() {
        println("Datos de la figura")
    }
}

class Cuadrado(val lado: Int): Figura {
    override fun calcularSuperficie(): Int {
        return lado * lado
    }

    override fun calcularPerimetro(): Int{
        return lado * 4
    }
}

class Rectangulo(val ladoMayor:Int, val ladoMenor: Int): Figura {
    override fun calcularSuperficie(): Int {
        return ladoMayor * ladoMenor
    }

    override fun calcularPerimetro(): Int {
        return (ladoMayor * 2) + (ladoMenor * 2)
    }
}

fun main(parametro: Array<String>) {
    val cuadrado1 = Cuadrado(10)
    cuadrado1.tituloResultado()
    println("Perimetro del cuadrado : ${cuadrado1.calcularPerimetro()}")
    println("Superficie del cuadrado : ${cuadrado1.calcularSuperficie()}")
    val rectangulo1 = Rectangulo(10, 5)
    rectangulo1.tituloResultado()
    println("Perimetro del rectangulo : ${rectangulo1.calcularPerimetro()}")
    println("Superficie del cuadrado : ${rectangulo1.calcularSuperficie()}")
}
```

En este problema la interface Figura tiene dos métodos abstractos que deben ser implementados por las clases y un método concreto, es decir ya implementado:

```
interface Figura {
    fun calcularSuperficie(): Int
    fun calcularPerimetro(): Int
    fun tituloResultado() {
        println("Datos de la figura")
    }
}
```

La clase Cuadrado indica que implementa la interface Figura, esto hace necesario que se implementen los métodos calcularSuperficie y calcularPerimetro:

```
class Cuadrado(val lado: Int): Figura {
    override fun calcularSuperficie(): Int {
        return lado * lado
    }

    override fun calcularPerimetro(): Int{
        return lado * 4
    }
}
```

De forma similar la clase Rectangulo implementa la interface Figura:

```
class Rectangulo(val ladoMayor:Int, val ladoMenor: Int): Figura {
    override fun calcularSuperficie(): Int {
        return ladoMayor * ladoMenor
    }

    override fun calcularPerimetro(): Int {
        return (ladoMayor * 2) + (ladoMenor * 2)
    }
}
```

En la función main definimos un objeto de la clase Cuadrado y otro de la clase Rectangulo, luego llamamos a los métodos tituloResultado, calcularPerimetro y calcularSuperficie para cada objeto:

```
fun main(parametro: Array<String>) {
    val cuadrado1 = Cuadrado(10)
    cuadrado1.tituloResultado()
    println("Perimetro del cuadrado : ${cuadrado1.calcularPerimetro()}")
    println("Superficie del cuadrado : ${cuadrado1.calcularSuperficie()}")
    val rectangulo1 = Rectangulo(10, 5)
    rectangulo1.tituloResultado()
    println("Perimetro del rectangulo : ${rectangulo1.calcularPerimetro()}")
    println("Superficie del cuadrado : ${rectangulo1.calcularSuperficie()}")
}
```

Acotaciones

Un método o función puede recibir como parámetro una interface. Luego le podemos pasar objetos de distintas clases que implementan dicha interface:

```
interface Figura {
    fun calcularSuperficie(): Int
    fun calcularPerimetro(): Int
    fun tituloResultado() {
        println("Datos de la figura")
    }
}

class Cuadrado(val lado: Int): Figura {
    override fun calcularSuperficie(): Int {
        return lado * lado
    }

    override fun calcularPerimetro(): Int{
        return lado * 4
    }
}

class Rectangulo(val ladoMayor:Int, val ladoMenor: Int): Figura {
    override fun calcularSuperficie(): Int {
        return ladoMayor * ladoMenor
    }

    override fun calcularPerimetro(): Int {
        return (ladoMayor * 2) + (ladoMenor * 2)
    }
}

fun imprimir(fig: Figura) {
    println("Perimetro: ${fig.calcularPerimetro()}")
    println("Superficie: ${fig.calcularsuperficie()}")
}

fun main(parametro: Array<String>) {
    val cuadrado1 = Cuadrado(10)
    cuadrado1.tituloResultado()
    println("Perimetro del cuadrado : ${cuadrado1.calcularPerimetro()}")
    println("Superficie del cuadrado : ${cuadrado1.calcularsuperficie()}")
    val rectangulo1 = Rectangulo(10, 5)
    rectangulo1.tituloResultado()
    println("Perimetro del rectangulo : ${rectangulo1.calcularPerimetro()}")
    println("Superficie del cuadrado : ${rectangulo1.calcularsuperficie()}")
    imprimir(cuadrado1)
    imprimir(rectangulo1)
}
```

La función imprimir recibe como parámetro fig que es de tipo Figura:

```
fun imprimir(fig: Figura) {
    println("Perimetro: ${fig.calcularPerimetro()}")
    println("Superficie: ${fig.calcularSuperficie()}")
}
```

En la main podemos llamar a la función imprimir pasando tanto el objeto cuadrado1 como rectangulo1:

```
    imprimir(cuadrado1)
    imprimir(rectangulo1)
```

Esto es posible ya que ambos objetos sus clases implementan la interface Figura.

Retornar (index.php?inicio=30)

35 - POO - arreglos con objetos

Dijimos que un arreglo es una estructura de datos que permite almacenar un CONJUNTO de datos del MISMO tipo. Con un único nombre se define un arreglo y por medio de un subíndice hacemos referencia a cada elemento del mismo (componente)

Vimos como crear arreglos con componentes de tipo Int, Char, Float, Double etc., ahora veremos como definir un arreglo con componentes de una determinada clase que declaramos nosotros.

Problema 1

Declarar una clase Persona con las propiedades nombre y edad, definir como métodos su impresión y otra que retorna true si es mayor de edad o false en caso contrario
En la función main definir un arreglo con cuatro elementos de tipo Persona. Calcular cuantas personas son mayores de edad.

Proyecto144 - Principal.kt

```
class Persona(val nombre: String, val edad: Int) {
    fun imprimir() {
        println("Nombre: $nombre  Edad: $edad")
    }

    fun esMayor() = if (edad >= 18) true else false
}

fun main(parametro: Array<String>) {
    val personas: Array<Persona> = arrayOf(Persona("ana", 22), Persona("juan", 1
3), Persona("carlos", 6), Persona("maria", 72))
    println("Listado de personas")
    for(per in personas)
        per.imprimir()
    var cant = 0
    for(per in personas)
        if (per.esMayor())
            cant++
    println("Cantidad de personas mayores de edad: $cant")
}
```

La declaración de la clase Persona define 2 propiedades en el mismo constructor y sus dos métodos:

```
class Persona(val nombre: String, val edad: Int) {
    fun imprimir() {
        println("Nombre: $nombre  Edad: $edad")
    }

    fun esMayor() = if (edad >= 18) true else false
}
```

En la función main definimos una variable llamada personas que es un Array con componentes de tipo Persona. Para definir sus componentes utilizamos la función arrayOf que nos provee la librería estándar de Kotlin:

```
    val personas: Array<Persona> = arrayOf(Persona("ana", 22), Persona("juan",
13), Persona("carlos", 6), Persona("maria", 72))
```

A la función arrayOf se le pasa cada uno de los objetos de tipo Persona.

Un Array una vez creado no puede cambiar su tamaño.

La forma más fácil de recorrer el Array es mediante un for:

```
    for(per in personas)
        per.imprimir()
```

En cada ciclo del for en la variable per se almacena una de las componentes del arreglo.

De forma similar para contar la cantidad de personas mayores de edad procedemos a definir un contador y mediante un for recorremos el arreglo y llamamos al método esMayor para cada objeto:

```
    var cant = 0
    for(per in personas)
        if (per.esMayor())
            cant++
    println("Cantidad de personas mayores de edad: $cant")
```

Acotaciones

A un Array lo podemos acceder por medio de un subíndice o por medio de llamadas a métodos, podemos cambiar el valor almacenado en una componente etc.:

```
    //imprimir los datos de la persona almacenada en la componente 0
    personas[0].imprimir()
    //imprimir la cantidad de compoentes del arreglo
    println(personas.size)
    //imprimir la edad de la persona almacenada en la última componente
    println(personas[3].nombre)
    //Copiar la persona almacenada en la primer componente en la segunda
    personas[1] = personas[0]
    personas[0].imprimir()
    personas[1].imprimir()
    //Acceder a la primer componente por medio de un método en lugar de un subí
ndice
    personas.get(0).imprimir()
    //Copiar la primer componente en la tercera mediante un método en lugar de
un subíndice
    personas.set(2, personas[0])
    println("-------------------------------")
    for(per in personas)
        per.imprimir()
```

Problemas propuestos

- Se tiene la declaración del siguiente data class:

  ```
  data class Articulo(val codigo: Int, val descripcion: String, var precio:
  Float)
  ```

  Definir un Array con 4 elmentos de tipo Articulo.
  Implementar dos funciones, una que le enviemos el Array y nos muestre todos sus componentes, y otra que también reciba el Array de artículos y proceda a aumentar en 10% a todos los productos.

- Declarar una clase Dado que tenga como propiedad su valor y dos métodos que permitan tirar el dado e imprimir su valor.
  En la main definir un Array con 5 objetos de tipo Dado.
  Tirar los 5 dados e imprimir los valores de cada uno.

Solución

Retornar (index.php?inicio=30)

36 - Funciones de orden superior

Kotlin es un lenguaje orientado a objetos pero introduce características existentes en los lenguajes funcionales que nos permiten crear un código más claro y expresivo.

Una de las características del paradigma de la programación funcional son las funciones de orden superior.

Las funciones de orden superior son funciones que pueden recibir como parámetros otras funciones y/o devolverlas como resultados.

Veremos una serie de ejemplos muy sencillos para ver como Kotlin implementa el concepto de funciones de orden superior y a medida que avancemos en el curso podremos ver las ventajas de este paradigma.

Problema 1

Definir una función de orden superior llamada operar. Llegan como parámetro dos enteros y una función. En el bloque de la función llamar a la función que llega como parámetro y enviar los dos primeros parámetros.
La función retorna un entero.

Proyecto147 - Principal.kt

```
fun operar(v1: Int, v2: Int, fn: (Int, Int) -> Int) : Int {
    return fn(v1, v2)
}

fun sumar(x1: Int, x2: Int) = x1 + x2

fun restar(x1: Int, x2: Int) = x1 - x2

fun multiplicar(x1: Int, x2: Int) = x1 * x2

fun dividir(x1: Int, x2: Int) = x1 / x2

fun main(parametro: Array<String>) {
    val resu1 = operar(10, 5, ::sumar)
    println("La suma de 10 y 5 es $resu1")
    val resu2 = operar(5, 2, ::sumar)
    println("La suma de 5 y 2 es $resu2")
    println("La resta de 100 y 40 es ${operar(100, 40, ::restar)}")
    println("El producto entre 5 y 20 es ${operar(5, 20, ::multiplicar)}")
    println("La división entre 10 y 5 es ${operar(10, 5, ::dividir)}")
}
```

Tenemos definidas 6 funciones en este problema. La única función de orden superior es la llamada "operar":

```
fun operar(v1: Int, v2: Int, fn: (Int, Int) -> Int) : Int {
    return fn(v1, v2)
}
```

El tercer parámetro de esta función se llama "fn" y es de tipo función. Cuando un parámetro es de tipo función debemos indicar los parámetros que tiene dicha función (en este caso tiene dos parámetros enteros) y luego del operador -> el tipo de dato que retorna esta función:

```
fn: (Int, Int) -> Int
```

Cuando tengamos una función como parámetro que no retorne dato se indica el tipo Unit, por ejemplo:

```
fn: (Int, Int) -> Unit
```

El algoritmo de la función operar consiste en llamar a la función fn y pasar los dos enteros que espera dicha función:

```
    return fn(v1, v2)
```

Como la función operar retorna un entero debemos indicar con la palabra clave return que devuelva el dato que retorna la función "fn".

Las cuatro funciones que calculan la suma, resta, multiplicación y división no tienen nada nuevo a lo visto en conceptos anteriores:

```
fun sumar(x1: Int, x2: Int) = x1 + x2

fun restar(x1: Int, x2: Int) = x1 - x2

fun multiplicar(x1: Int, x2: Int) = x1 * x2

fun dividir(x1: Int, x2: Int) = x1 / x2
```

En la función main llamamos a la función operar y le pasamos tres datos, dos enteros y uno con la referencia de una función:

```
    val resu1 = operar(10, 5, ::sumar)
```

Como vemos para pasar la referencia de una función antecedemos el operador ::

La función operar retorna un entero y lo almacenamos en la variable resu1 que mostramos luego por pantalla:

```
    println("La suma de 10 y 5 es $resu1")
```

Es importante imaginar el funcionamiento de la función operar que recibe tres datos y utiliza uno de ellos para llamar a otra función que retorna un valor y que luego este valor lo retorna operar y llega finalmente a la variable "resu1".

Llamamos a operar y le pasamos nuevamente la referencia a la función sumar:

```
    val resu2 = operar(5, 2, ::sumar)
    println("La suma de 5 y 2 es $resu2")
```

De forma similar llamamos a operar y le pasamos las referencias a las otras funciones:

```
    println("La resta de 100 y 40 es ${operar(100, 40, ::restar)}")
    println("El producto entre 5 y 20 es ${operar(5, 20, ::multiplicar)}")
    println("La división entre 10 y 5 es ${operar(10, 5, ::dividir)}")
```

Tener en cuenta que para sumar dos enteros es mejor llamar directamente a la función sumar y pasar los dos enteros, pero el objetivo de este problema es conocer la sintaxis de las funciones de orden superior presentando el problema más sencillo.

Las funciones de orden superior se pueden utilizar perfectamente en los métodos de una clase.

Problema 2

Declarar una clase que almacene el nombre y la edad de una persona. Definir un método que retorne true o false según si la persona es mayor de edad o no. Esta función debe recibir como parámetro una función que al llamarla pasando la edad de la persona retornara si es mayor o no de edad.
Tener en cuenta que una persona es mayor de edad en Estados Unidos si tiene 21 o más años y en Argentina si tiene 18 o más años.

Proyecto148 - Principal.kt

```
class Persona(val nombre: String, val edad: Int) {
    fun esMayor(fn:(Int) -> Boolean): Boolean {
        return fn(edad)
    }
}

fun mayorEstadosUnidos(edad: Int): Boolean {
    if (edad >= 21)
        return true
    else
        return false
}

fun mayorArgentina(edad: Int): Boolean {
    if (edad >= 18)
        return true
    else
        return false
}

fun main(parametro: Array<String>) {
    val persona1 = Persona("juan", 18)
    if (persona1.esMayor(::mayorArgentina))
        println("${persona1.nombre} es mayor si vive en Argentina")
    else
        println("${persona1.nombre} no es mayor si vive en Argentina")
    if (persona1.esMayor(::mayorEstadosUnidos))
        println("${persona1.nombre} es mayor si vive en Estados Unidos")
    else
        println("${persona1.nombre} no es mayor si vive en Estados Unidos")
}
```

Declaramos la clase Persona con dos propiedades llamadas nombre y edad:

```
class Persona(val nombre: String, val edad: Int) {
```

La clase persona por si misma no guarda la nacionalidad de la persona, en cambio cuando se pregunta si es mayor de edad se le pasa como referencia una función que al pasar la edad nos retorna true o false:

```
    fun esMayor(fn:(Int) -> Boolean): Boolean {
        return fn(edad)
    }
```

Tenemos dos funciones que al pasar una edad nos retornan si es mayor de edad o no:

```
fun mayorEstadosUnidos(edad: Int): Boolean {
    if (edad >= 21)
        return true
    else
        return false
}

fun mayorArgentina(edad: Int): Boolean {
    if (edad >= 18)
        return true
    else
        return false
}
```

En la función main creamos un objeto de la clase persona:

```
    val persona1 = Persona("juan", 18)
```

Llamamos al método esMayor del objeto persona1 y le pasamos la referencia de la función "mayorArgentina":

```
    if (persona1.esMayor(::mayorArgentina))
        println("${persona1.nombre} es mayor si vive en Argentina")
    else
        println("${persona1.nombre} no es mayor si vive en Argentina")
```

Ahora llamamos al método esMayor pero pasando la referencia de la función "mayorEstadosUnidos":

```
    if (persona1.esMayor(::mayorEstadosUnidos))
        println("${persona1.nombre} es mayor si vive en Estados Unidos")
    else
        println("${persona1.nombre} no es mayor si vive en Estados Unidos")
```

Como podemos comprobar el concepto de funciones de orden superior es aplicable a los métodos de una clase.

No hicimos un código más conciso con el objeto que quede más claro la sintaxis de funciones de orden superior, pero el mismo problema puede ser:

```
class Persona(val nombre: String, val edad: Int) {
    fun esMayor(fn:(Int) -> Boolean) = fn(edad)
}

fun mayorEstadosUnidos(edad: Int) = if (edad >= 21) true else false

fun mayorArgentina(edad: Int) = if (edad >= 18) true else false

fun main(parametro: Array<String>) {
    val persona1 = Persona("juan", 18)
    if (persona1.esMayor(::mayorArgentina))
        println("${persona1.nombre} es mayor si vive en Argentina")
    else
        println("${persona1.nombre} no es mayor si vive en Argentina")
    if (persona1.esMayor(::mayorEstadosUnidos))
        println("${persona1.nombre} es mayor si vive en Estados Unidos")
    else
        println("${persona1.nombre} no es mayor si vive en Estados Unidos")
}
```


Retornar (index.php?inicio=30)

37 - Expresiones lambda

Una expresión lambda es cuando enviamos a una función de orden superior directamente una función anónima.

Es más común enviar una expresión lambda en lugar de enviar la referencia de una función como vimos en el concepto anterior.

Problema 1

Definir una función de orden superior llamada operar. Llegan como parámetro dos enteros y una función. En el bloque de la función llamar a la función que llega como parámetro y enviar los dos primeros parámetros.

Desde la función main llamar a operar y enviar distintas expresiones lambdas que permitan sumar, restar y elevar el primer valor al segundo .

Proyecto149 - Principal.kt

```
fun operar(v1: Int, v2: Int, fn: (Int, Int) -> Int) : Int{
    return fn(v1, v2)
}

fun main(parametro: Array<String>) {
    val suma = operar(2, 3, {x, y -> x + y})
    println(suma)
    val resta = operar(12, 2, {x, y -> x - y})
    println(resta)
    var elevarCuarta = operar(2, 4, {x, y ->
        var valor = 1
        for(i in 1..y)
            valor = valor * x
        valor
    })
    println(elevarCuarta)
}
```

La función de orden superior operar es la que vimos en el concepto anterior:

```
fun operar(v1: Int, v2: Int, fn: (Int, Int) -> Int) : Int{
    return fn(v1, v2)
}
```

La primer expresión lambda podemos identificarla en la primer llamada a la función operar:

```
    val suma = operar(2, 3, {x, y -> x + y})
    println(suma)
```

Una expresión lambda está compuesta por una serie de parámetros (en este caso x e y), el signo -> y el cuerpo de una función:

```
{x, y -> x + y}
```

Como podemos comprobar toda expresión lambda va encerrada entre llaves.

Podemos indicar el tipo de dato de los parámetros de la función pero normalmente no se los dispone:

```
val suma = operar(2, 3, {x: Int, y: Int -> x + y})
```

El nombre de los parámetros se llaman x e y, pero podrían tener cualquier nombre:

```
    val suma = operar(2, 3, {valor1, valor2: Int -> valor1 + valor2})
```

Siempre hay que indicar el tipo de dato que devuelve la función, en este caso retorna un Int que lo indicamos después de los dos puntos. Si no retorna dato la función se dispone Unit.

El algoritmo de la función lo expresamos después del signo ->

```
x + y
```

Se infiere que la suma de x e y es el valor a retornar.

La segunda llamada a operar le pasamos una expresión lambda similar a la primer llamada:

```
    val resta = operar(12, 2, {x, y -> x - y})
    println(resta)
```

La tercer llamada a operar le pasamos una expresión lambda que en su algoritmo procedemos a elevar el primer valor según el dato que llega en el segundo valor:

```
    var elevarCuarta = operar(2, 4, {x, y ->
        var valor = 1
        for(i in 1..y)
            valor = valor * x
        valor
    })
```

Dentro de la expresión lambda podemos implementar un algoritmo más complejo como es este caso donde elevamos el parámetro x al exponente indicado con y.

Es importante notar que la función de orden superior recibe como parámetro una función con dos parámetros de tipo Int y retorna un Int:

```
fn: (Int, Int) -> Int
```

La expresión lambda debe pasar una función con la misma estructura, es decir dos parámetros enteros y retornar un entero:

```
{x, y ->
        var valor = 1
        for(i in 1..y)
            valor = valor * x
        valor
    }
```

Podemos identificar los dos parámetros x e y, y el valor que devuelve es el dato indicado después del for.

Un último cambio que implementaremos a nuestro programa es que cuando una expresión lambda es el último parámetro de una función podemos indicar la expresión lambda después de los paréntesis para que sea más legible el código de nuestro programa:

```
fun main(parametro: Array<String>) {
    val suma = operar(2, 3) {x, y -> x + y}
    println(suma)
    val resta = operar(12, 2)  {x, y -> x - y}
    println(resta)
    var elevarCuarta = operar(2, 4) {x, y ->
        var valor = 1
        for(i in 1..y)
            valor = valor * x
        valor
    }
    println(elevarCuarta)
}
```

Lo más común es utilizar esta sintaxis para pasar una expresión lambda cuando es el último parámetro de una función.

Problema 2

Confeccionar una función de orden superior que reciba un arreglo de enteros y una función con un parámetro de tipo Int y que retorne un Boolean.

La función debe analizar cada elemento del arreglo llamando a la función que recibe como parámetro, si retorna un true se pasa a mostrar el elemento.

En la función main definir un arreglo de enteros de 10 elementos y almacenar valores aleatorios comprendidos entre 0 y 99.

Imprimir del arreglo:

- Los valores múltiplos de 2
- Los valores múltiplos de 3 o de 5
- Los valores mayores o iguales a 50
- Los valores comprendidos entre 1 y 10, 20 y 30, 90 y 95

Proyecto150 - Principal.kt

```
fun imprimirSi(arreglo: IntArray, fn:(Int) -> Boolean) {
    for(elemento in arreglo)
        if (fn(elemento))
            print("$elemento ")
    println();
}

fun main(parametro: Array<String>) {
    val arreglo1 = IntArray(10)
    for(i in arreglo1.indices)
        arreglo1[i] = ((Math.random() * 100)).toInt()
    println("Imprimir los valores múltiplos de 2")
    imprimirSi(arreglo1) {x -> x % 2 == 0}
    println("Imprimir los valores múltiplos de 3 o de 5")
    imprimirSi(arreglo1) {x -> x % 3 == 0 || x % 5 ==0}
    println("Imprimir los valores mayores o iguales a 50")
    imprimirSi(arreglo1) {x -> x >= 50}
    println("Imprimir los valores comprendidos entre 1 y 10, 20 y 30, 90 y 95")
    imprimirSi(arreglo1) {x -> when(x) {
        in 1..10 -> true
        in 20..30 -> true
        in 90..95 -> true
        else -> false
    }}
    println("Imprimir todos los valores")
    imprimirSi(arreglo1) {x -> true}
}
```

La función de orden superior ImprimirSi recibe un arreglo de enteros y una función:

```
fun imprimirSi(arreglo: IntArray, fn:(Int) -> Boolean) {
```

Dentro de la función recorremos el arreglo de enteros y llamamos a la función que recibe como parámetro para cada elemento del arreglo, si retorna un true pasamos a mostrar el valor:

```
    for(elemento in arreglo)
        if (fn(elemento))
            print("$elemento ")
    println();
}
```

La ventaja de la función imprimirSi es que podemos utilizarla para resolver una gran cantidad de problemas, solo debemos pasar una expresión lambda que analice cada entero que recibe.

En la función main creamos el arreglo de 10 enteros y cargamos 10 valores aleatorios comprendidos entre 0 y 99:

```
fun main(parametro: Array<String>) {
    val arreglo1 = IntArray(10)
    for(i in arreglo1.indices)
        arreglo1[i] = ((Math.random() * 100)).toInt()
```

Para imprimir los valores múltiplos de 2 nuestra expresión lambda recibe como parámetro un Int llamado x y el algoritmo que verifica si es par o no consiste en verificar si el resto de dividirlo por 2 es cero:

```
    println("Imprimir los valores múltiplos de 2")
    imprimirSi(arreglo1) {x -> x % 2 == 0}
```

Es importante entender que nuestra expresión lambda no tiene una estructura repetitiva, sino que desde la función imprimirSi llamará a esta función anónima tantas veces como elementos tenga el arreglo.

El algoritmo de la función varía si queremos identificar si el número es múltiplo de 3 o de 5:

```
    println("Imprimir los valores múltiplos de 3 o de 5")
    imprimirSi(arreglo1) {x -> x % 3 == 0 || x % 5 ==0}
```

Para imprimir todos los valores mayores o iguales a 50 debemos verificar si el parámetro x es >= 50:

```
    println("Imprimir los valores mayores o iguales a 50")
    imprimirSi(arreglo1) {x -> x >= 50}
```

Para analizar si está la componente en distintos rangos podemos utilizar la instrucción when:

```
    println("Imprimir los valores comprendidos entre 1 y 10, 20 y 30, 90 y 95")
    var cant = 0
    imprimirSi(arreglo1) {x -> when(x) {
        in 1..10 -> true
        in 20..30 -> true
        in 90..95 -> true
        else -> false
    }}
```

En forma muy sencilla si queremos imprimir todo el arreglo retornamos para cada elemento que analiza nuestra expresión lambda el valor true:

```
    println("Imprimir todos los valores")
    imprimirSi(arreglo1) {x -> true}
```

Este es un problema donde ya podemos notar las potencialidades de las funciones de orden superior y las expresiones lambdas que le pasamos a las mismas.

Acotaciones

Dijimos que uno de los principios de Kotlin es permitir escribir código conciso, para esto cuando tenemos una expresión lambda cuya función recibe un solo parámetro podemos obviarlo, inclusive el signo ->
Luego por convención ese único parámetro podemos hacer referencia al mismo con la palabra "it"

Entonces nuestro programa definitivo conviene escribirlo con la siguiente sintaxis:

Proyecto150 - Principal.kt

```
fun imprimirSi(arreglo: IntArray, fn:(Int) -> Boolean) {
    for(elemento in arreglo)
        if (fn(elemento))
            print("$elemento ")
    println();
}

fun main(parametro: Array<String>) {
    val arreglo1 = IntArray(10)
    for(i in arreglo1.indices)
        arreglo1[i] = ((Math.random() * 100)).toInt()
    println("Imprimir los valores múltiplos de 2")
    imprimirSi(arreglo1) {it % 2 == 0}
    println("Imprimir los valores múltiplos de 3 o de 5")
    imprimirSi(arreglo1) {it % 3 == 0 || it % 5 ==0}
    println("Imprimir los valores mayores o iguales a 50")
    imprimirSi(arreglo1) {it >= 50}
    println("Imprimir los valores comprendidos entre 1 y 10, 20 y 30, 90 y 95")
    imprimirSi(arreglo1) {when(it) {
        in 1..10 -> true
        in 20..30 -> true
        in 90..95 -> true
        else -> false
    }}
    println("Imprimir todos los valores")
    imprimirSi(arreglo1) {true}
}
```

Tener en cuenta que cuando llamamos desde la función imprimirSi:

```
        if (fn(elemento))
```

La expresión lambda recibe un parámetro llamado por defecto "it" y se almacena el valor pasado "elemento":

```
    imprimirSi(arreglo1) {it % 2 == 0}
```

Problema 3

Confeccionar una función de orden superior que reciba un String y una función con un parámetro de tipo Char y que retorne un Boolean.

La función debe analizar cada elemento del String llamando a la función que recibe como parámetro, si retorna un true se agrega dicho caracter al String que se retornará.

En la función main definir un String con una cadena cualquiera.

Llamar a la función de orden superior y pasar expresiones lambdas para filtrar y generar otro String con las siguientes restricciones:

- Un String solo con las vocales
- Un String solo con los caracteres en minúsculas
- Un String con todos los caracteres no alfabéticos

Proyecto151 - Principal.kt

```
fun filtrar(cadena: String, fn: (Char) -> Boolean): String
{
    val cad = StringBuilder()
    for(ele in cadena)
        if (fn(ele))
            cad.append(ele)
    return cad.toString()
}

fun main(parametro: Array<String>) {
    val cadena="¿Esto es la prueba 1 o la prueba 2?"
    println("String original")
    println(cadena)
    val resultado1 = filtrar(cadena) {
        if (it == 'a' || it == 'e' || it == 'i' || it == 'o' || it == 'u' ||
            it == 'A' || it == 'E' || it == 'I' || it == 'O' || it == 'U' )
            true
        else
            false
    }
    println("Solo las vocales")
    println(resultado1)
    var resultado2 = filtrar(cadena) {
        if (it in 'a'..'z')
            true
        else
            false
    }
    println("Solo los caracteres en minúsculas")
    println(resultado2)
    var resultado3 = filtrar(cadena) {
        if (it !in 'a'..'z' && it !in 'A'..'Z')
            true
        else
            false
    }
    println("Solo los caracteres no alfabéticos")
    println(resultado3)
}
```

Nuestra función de orden superior se llama filtrar y recibe un String y una función:

```
fun filtrar(cadena: String, fn: (Char) -> Boolean): String
```

La función que recibe tiene un parámetro de tipo Char y retorna un Boolean.

El algoritmo de la función filtrar consiste en recorrer el String que llega como parámetro y llamar a la función fn que es la que informará si el caracter analizado debe formar parte del String final a retornar.

Mediante un objeto de la clase StringBuilder almacenamos los caracteres a devolver, previo a retornar extraemos como String el contenido del StringBuilder:

```
    val cad = StringBuilder()
    for(ele in cadena)
        if (fn(ele))
            cad.append(ele)
    return cad.toString()
```

En la función main definimos un String con una cadena cualquiera:

```
fun main(parametro: Array<String>) {
    val cadena="¿Esto es la prueba 1 o la prueba 2?"
    println("String original")
    println(cadena)
```

La primer llamada a la función de orden superior la hacemos enviando una expresión lambda que considere parte del String a generar solo las vocales minúsculas y mayúsculas:

```
    val resultado1 = filtrar(cadena) {
        if (it == 'a' || it == 'e' || it == 'i' || it == 'o' || it == 'u' ||
            it == 'A' || it == 'E' || it == 'I' || it == 'O' || it == 'U' )
            true
        else
            false
    }
    println("Solo las vocales")
    println(resultado1)
```

Recordemos que utilizamos "it" ya que la función anónima tiene un solo parámetro y nos permite expresar un código más conciso, en la forma larga debería ser:

```
    val resultado1 = filtrar(cadena) { car ->
        if (car == 'a' || car == 'e' || car == 'i' || car == 'o' || car == 'u'
||
            car == 'A' || car == 'E' || car == 'I' || car == 'O' || car == 'U'
)
            true
        else
            false
    }
    println("Solo las vocales")
    println(resultado1)
```

Para generar un String solo con las letras mayúsculas debemos verificar si el parámetro de la función anónima se encuentra en el rango 'a'..'z':

```
    var resultado2 = filtrar(cadena) {
        if (it in 'a'..'z')
            true
        else
            false
    }
    println("Solo los caracteres en minúsculas")
    println(resultado2)
```

Por último para recuperar todos los símbolos que no sean letras expresamos la siguiente condición:

```
    var resultado3 = filtrar(cadena) {
        if (it !in 'a'..'z' && it !in 'A'..'Z')
            true
        else
            false
    }
    println("Solo los caracteres no alfabéticos")
    println(resultado3)
```

Con este problema podemos seguir apreciando las grandes ventajas que nos proveen las expresiones lambdas para la resolución de algoritmos.

Retornar (index.php?inicio=30)

38 - Expresiones lambda con arreglos IntArray, FloatArray, DoubleArray etc.

Vimos en conceptos anteriores que para los tipos básicos Byte, Short, Int, Long, Float, Double, Char y Boolean tenemos una serie de clases para definir arreglos de dichos tipos: ByteArray, ShortArray, IntArray, LongArray, FloatArray, DoubleArray, CharArray y BooleanArray.

Vimos como crear por ejemplo un arreglo de 3 enteros y su posterior carga con la sintaxis:

```
var arreglo = IntArray(3)
arreglo[0] = 10
arreglo[1] = 10
arreglo[2] = 10
```

La clase IntArray tiene una serie de métodos que requieren una expresión lambda, como no conocíamos las expresiones lambda no las utilizamos en los conceptos que vimos arreglos.

La clase IntArray como las otras que nombramos (ByteArray, ShortArray, etc.) tienen un segundo constructor con dos parámetros, el primero indica la cantidad de elementos del arreglo y el segundo debe ser una expresión lambda que le indicamos el valor a almacenar en cada componente del arreglo:

```
var arreglo = IntArray(10) {5}
```

Se crea un arreglo de 10 elementos y se almacena el número 5 en cada componente.

Si queremos almacenar en la primer componente un 0, en la segunda un 1 y así sucesivamente podemos disponer la siguiente sintaxis:

```
var arreglo = IntArray(10) {it}
```

Esto funciona debido a que la función de orden superior cuando llama a nuestra expresión lambda para cada componente del arreglo nos envía la posición. Recordemos que de forma larga podemos expresarlo con la siguiente sintaxis:

```
var arreglo = IntArray(10) {indice -> indice}
```

Si queremos guardar los valores 0,2,4,6 etc. podemos utilizar la sintaxis:

```
var arreglo = IntArray(10) {it * 2}
```

Si queremos almacenar valores aleatorios comprendidos entre 1 y 6 podemos crear el objeto de la clase IntArray con la siguiente sintaxis:

```
    var arreglo = IntArray(10) {((Math.random() * 6) + 1).toInt()}
```

Problema 1

Crear un objeto de la clase IntArray de 20 elementos con valores aleatorios comprendidos entre 0 y 10.

Imprimir del arreglo:

- Todos los elementos.
- Cantidad de elementos menores o iguales a 5.
- Mostrar un mensaje si todos los elementos son menores o iguales a 9.
- Mostrar un mensaje si al menos uno de los elementos del arreglo almacena un 10.

Proyecto152 - Principal.kt

```
fun main(parametro: Array<String>) {
    var arreglo = IntArray(20) {(Math.random() * 11).toInt()}
    println("Listado completo del arreglo")
    for(elemento in arreglo)
        print("$elemento ")
    println()
    val cant1 = arreglo.count { it <= 5}
    println("Cantidad de elementos menores o iguales a 5: $cant1")
    if (arreglo.all {it <= 9})
        println("Todos los elementos son menores o iguales a 9")
    else
        println("No todos los elementos son menores o iguales a 9")
    if (arreglo.any {it == 10})
        println("Al menos uno de los elementos es un 10")
    else
        println("Todos los elementos son distintos a 10")
}
```

Para resolver estos algoritmos utilizamos una serie de métodos que nos provee la clase IntArray.

Definimos un objeto de la clase IntArray de 20 elementos y pasamos una expresión lambda para generar valores aleatorios comprendidos entre 0 y 10:

```
fun main(parametro: Array<String>) {
    var arreglo = IntArray(20) {(Math.random() * 11).toInt()}
```

Listado completo del arreglo:

```
    println("Listado completo del arreglo")
    for(elemento in arreglo)
        print("$elemento ")
```

Para contar la cantidad de elementos iguales al valor 5 la clase IntArray dispone un método llamado count que tiene un solo parámetro de tipo función:

```
    val cant1 = arreglo.count { it <= 5}
```

Cuando una función de orden superior tiene un solo parámetro como ocurre con count no es necesario disponer los paréntesis:

```
    val cant1 = arreglo.count() { it <= 5}
```

La forma más larga y poco empleada de codificar la expresión lambda con un único parámetro es:

```
val cant1 = arreglo.count({ valor -> valor <= 5})
```

Si queremos analizar si todos los elementos del arreglo cumplen una condición disponemos de la función all (retorna un Boolean):

```
    if (arreglo.all {it <= 9})
        println("Todos los elementos son menores o iguales a 9")
    else
        println("No todos los elementos son menores o iguales a 9")
```

Para verificar si alguno de los elementos cumplen una condición podemos utilizar el método any pasando la expresión lambda:

```
    if (arreglo.any {it == 10})
        println("Al menos uno de los elementos es un 10")
    else
        println("Todos los elementos son distintos a 10")
```

Siempre debemos consultar la página oficial de Kotlin donde se documentan todas las clases de la librería estándar, en nuestro caso deberíamos consultar la clase IntArray (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-int-array/).

Problema propuesto

- Crear un arreglo de tipo FloatArray de 10 elementos, cargar sus elementos por teclado.
  Imprimir la cantidad de valores ingresados menores a 50.
  Imprimir un mensaje si todos los valores son menores a 50.

Solución

Retornar (index.php?inicio=30)

39 - Expresiones lambda: Acceso a las variables externas a la expresión lambda

Hemos visto en conceptos anteriores que una expresión lambda es cuando enviamos a una función de orden superior directamente una función anónima.
Dentro de la función lambda podemos acceder a los parámetros de la misma si los tiene, definir variables locales y veremos ahora que podemos acceder a variables externas a la expresión lambda definida.

Problema 1

Confeccionar una función de orden superior que reciba un arreglo de enteros y una función con un parámetro de tipo Int y que no retorne nada.

La función debe enviar cada elemento del arreglo a la expresión lambda definida.

En la función main definir un arreglo de enteros de 10 elementos y almacenar valores aleatorios comprendidos entre 0 y 99.

Imprimir del arreglo:

- La cantidad de múltiplos de 3
- La suma de todas las componentes superiores a 50

Proyecto154 - Principal.kt

```
fun recorrerTodo(arreglo: IntArray, fn:(Int) -> Unit) {
    for(elemento in arreglo)
        (fn(elemento))
}

fun main(parametro: Array<String>) {
    val arreglo1 = IntArray(10)
    for (i in arreglo1.indices)
        arreglo1[i] = ((Math.random() * 100)).toInt()
    println("Impresion de todo el arreglo")
    for (elemento in arreglo1)
        print("$elemento ")
    println()
    var cantidad = 0
    recorrerTodo(arreglo1) {
        if (it % 3 == 0)
            cantidad++
    }
    println("La cantidad de elementos múltiplos de 3 son $cantidad")
    var suma = 0
    recorrerTodo(arreglo1) {
        if (it > 50)
            suma += it
    }
    println("La suma de todos los elementos mayores a 50 es $suma")
}
```

Lo más importante es entender y tener en cuenta que una variable que definamos fuera de la expresión lambda podemos accederla dentro de la expresión:

```
    var cantidad = 0
    recorrerTodo(arreglo1) {
        if (it % 3 == 0)
            cantidad++
    }
    println("La cantidad de elementos múltiplos de 3 son $cantidad")
```

La variable cantidad es de tipo Int y se inicializa en 0 previo a llamar a la funcióm recorrerTodo y pasar la expresión lambda donde incrementamos el contador "cantidad" cada vez que analizamos un elemento del arreglo que nos envía la función de orden superior.

Cuando finalizan todas las ejecuciones de la expresión lambda procedemos a mostrar el contador.

De forma similar para sumar todas las componentes del arreglo que tienen un valor superior a 50 llamamos a la función recorrerTodo y en la expresión lambda acumulamos la componente siempre y cuando tenga almacenado un valor superior a 50:

```
    var suma = 0
    recorrerTodo(arreglo1) {
        if (it > 50)
            suma += it
    }
    println("La suma de todos los elementos mayores a 50 es $suma")
```

Otra cosa que debe quedar claro que cuando una función no retorna dato y es un parámetro de otra función es obligatorio indicar el objeto Unit como dato de devolución:

```
fun recorrerTodo(arreglo: IntArray, fn:(Int) -> Unit) {
```

La función recorrerTodo también no retorna dato pero Kotlin interpreta por defecto que devuelve un tipo Unit. Luego generalmente no se utiliza la sintaxis:

```
fun recorrerTodo(arreglo: IntArray, fn:(Int) -> Unit): Unit {
```

No es común indicar el objeto Unit si la función no retorna dato, por defecto Kotlin sabe que debe retornar un objeto Unit.

La clase IntArray como las otras clases similares ByteArray, ShortArray, FloatArray etc. disponen de un método llamado forEach que se le debe enviar una expresión lambda, la cual se ejecuta para cada elemento del arreglo de forma similar a la función que creamos recorrerTodo.

Problema 2

Resolver el mismo problema anterior pero emplear el método forEach que dispone la clase IntArray para analizar todas las componentes del arreglo.

En la función main definir un arreglo de enteros de 10 elementos y almacenar valores aleatorios comprendidos entre 0 y 99.

Imprimir del arreglo:

- La cantidad de múltiplos de 3
- La suma de todas las componentes superiores a 50

Proyecto155 - Principal.kt

```
fun main(parametro: Array<String>) {
    val arreglo1 = IntArray(10)
    for (i in arreglo1.indices)
        arreglo1[i] = ((Math.random() * 100)).toInt()
    println("Impresion de todo el arreglo")
    for (elemento in arreglo1)
        print("$elemento ")
    println()
    var cantidad = 0
    arreglo1.forEach {
        if (it % 3 == 0)
            cantidad++
    }
    println("La cantidad de elementos múltiplos de 3 son $cantidad")
    var suma = 0
    arreglo1.forEach {
        if (it > 50)
            suma += it
    }
    println("La suma de todos los elementos mayores a 50 es $suma")
}
```

Es más lógico utilizar la función de orden superior forEach que dispone la clase IntArray en lugar de crear nuestra propia función (lo hicimos en el problema anterior con el objetivo de practicar la creación de funciones de orden superior)

Para contar la cantidad de múltiplos de 3 definimos un contador previo a llamar a forEach:

```
    var cantidad = 0
    arreglo1.forEach {
        if (it % 3 == 0)
            cantidad++
    }
    println("La cantidad de elementos múltiplos de 3 son $cantidad")
```

Recordar que dentro de la expresión lambda tenemos acceso a la variable externa cantidad.

Problema 3

Declarar una clase Persona con las propiedades nombre y edad, definir como métodos su impresión y otra que retorna true si es mayor de edad o false en caso contrario (18 años o más para ser mayor)
En la función main definir un arreglo con cuatro elementos de tipo Persona. Calcular cuantas personas son mayores de edad llamando al método forEach de la clase Array.

Proyecto156 - Principal.kt

```
class Persona(val nombre: String, val edad: Int) {
    fun imprimir() {
        println("Nombre: $nombre  Edad: $edad")
    }

    fun esMayor() = if (edad >= 18) true else false
}

fun main(parametro: Array<String>) {
    val personas: Array<Persona> = arrayOf(Persona("ana", 22),
            Persona("juan", 13),
            Persona("carlos", 6),
            Persona("maria", 72))
    println("Listado de personas")
    for(per in personas)
        per.imprimir()
    var cant = 0
    personas.forEach {
        if (it.esMayor())
            cant++
    }
    println("Cantidad de personas mayores de edad: $cant")
}
```

También la clase Array dispone de un método forEach que le pasamos una expresión lambda y recibe como parámetro un elemento del Array cada vez que se ejecuta. Mediante el parámetro it que en este caso es de la clase Persona analizamos si es mayor de edad:

```
    var cant = 0
    personas.forEach {
        if (it.esMayor())
            cant++
    }
    println("Cantidad de personas mayores de edad: $cant")
```

Siempre recordemos que utilizamos la palabra it para que sea más conciso nuestro programa, en la forma larga podemos escribir:

```
    var cant = 0
    personas.forEach {persona: Persona ->
        if (persona.esMayor())
            cant++
    }
    println("Cantidad de personas mayores de edad: $cant")
```

Problema propuesto

- Declarar una clase Dado que tenga como propiedad su valor y dos métodos que permitan tirar el dado e imprimir su valor.
  En la main definir un Array con 5 objetos de tipo Dado.
  Tirar los 5 dados e imprimir cuantos 1, 2, 3, 4, 5 y 6 salieron.

Solución

Retornar (index.php?inicio=30)

40 - Funciones de extensión

Otra de las características muy útil de Kotlin para los programadores es el concepto de las funciones de extensión.

Mediante las funciones de extensión Kotlin nos permite agregar otros métodos a una clase existente sin tener que heredar de la misma.

Con una serie de ejemplos quedará claro esta característica.

Problema 1

Agregar dos funciones a la clase String que permitan recuperar la primer mitad y la segunda mitad.

Proyecto158 - Principal.kt

```
fun String.mitadPrimera(): String {
    return this.substring(0..this.length/2-1)
}

fun String.segundaMitad(): String{
    return this.substring(this.length/2..this.length-1)
}

fun main(args: Array<String>) {
    val cadena1 = "Viento"
    println(cadena1.mitadPrimera())
    println(cadena1.segundaMitad())
}
```

Para definir una función de extensión a una clase solo debemos anteceder al nombre de la función a que clase estamos extendiendo:

```
fun String.mitadPrimera(): String {
    return this.substring(0..this.length/2-1)
}
```

Dentro de la función mediante la palabra clave this podemos acceder al objeto que llamó a dicha función (en nuestro ejemplo como llamamos a mitadPrimera mediante el objeto cadena1 tenemos acceso a la cadena "Viento")

Cuando llamamos a una función de extensión lo hacemos en forma idéntica a las otras funciones que tiene la clase:

```
    println(cadena1.mitadPrimera())
```

La segunda función retorna un String con la segunda mitad del String:

```
fun String.segundaMitad(): String{
    return this.substring(this.length/2..this.length-1)
}
```

Podemos definir funciones de extensión que sobrescriban funciones ya existentes en la clase, es decir podríamos en la clase String crear una función llamada capitalize() y definir una nueva funcionalidad.

Problema 2

Agregar una función a la clase IntArray que permitan imprimir todas sus componentes en una línea.

Proyecto159 - Principal.kt

```
fun IntArray.imprimir() {
    print("[")
    for(elemento in this) {
        print("$elemento ")
    }
    println("]");
}

fun main(args: Array<String>) {
    val arreglo1= IntArray(10, {it})
    arreglo1.imprimir()
}
```

Definimos una función de extensión llamada imprimir en la clase IntArray, en su algoritmo mediante un for imprimimos todas sus componentes:

```
fun IntArray.imprimir() {
    print("[")
    for(elemento in this) {
        print("$elemento ")
    }
    println("]");
}
```

En la main definimos un arreglo de tipo IntArray de 10 elemento y guardamos mediante una expresión lambda los valores de 0 al 9.

Como ahora la clase IntArray tiene una función que imprime todas sus componentes podemos llamar directamente al método imprimir de la clase IntArray:

```
fun main(args: Array<String>) {
    val arreglo1= IntArray(10, {it})
    arreglo1.imprimir()
}
```

Problemas propuestos

- Agregar a la clase String un método imprimir que muestre todos los caracteres que tiene almacenado en una línea.

- Codicar la función de extensión llamada "hasta" que debe extender la clase Int y tiene por objetivo mostrar desde el valor entero que almacena el objeto hasta el valor que llega como parámetro:

  ```
  fun Int.hasta(fin: Int) {
  ```

Solución

Retornar (index.php?inicio=30)

41 - Sobrecarga de operadores

El lenguaje Kotlin permite que ciertos operadores puedan sobrecargarse y actúen de diferentes maneras según al objeto que se aplique.

La sobrecarga de operadores debe utilizarse siempre y cuando tenga sentido para la clase que se está implementando. Los conceptos matemáticos de vectores y matrices son casos donde la sobrecarga de operadores nos puede hacer nuestro código más legible y elegante.

Para sobrecargar los operadores +, -, * y / debemos implementar una serie de métodos especiales dentro de la clase:

```
Operación    Nombre del método a definir
a + b        plus
a - b        minus
a * b        times
a / b        div
a % b        rem
a..b         rangeTo
```

Problema 1

Declarar una clase llamada Vector que administre un array de 5 elementos de tipo entero y cargar valores aleatorios entre 1 y 10. Sobrecargar los operadores +, -, * y /
En la main definir una serie de objetos de la clase Vector y operar con ellos

Proyecto162 - Principal.kt

```
class Vector {
    val arreglo = IntArray(5)

    fun cargar() {
        for(i in arreglo.indices)
            arreglo[i] = (Math.random() * 11 + 1).toInt()
    }

    fun imprimir() {
        for(elemento in arreglo)
            print("$elemento ")
        println()
    }
    operator fun plus(vector2: Vector): Vector {
        var suma = Vector()
        for(i in arreglo.indices)
            suma.arreglo[i] = arreglo[i] + vector2.arreglo[i]
        return suma
    }

    operator fun minus(vector2: Vector): Vector {
        var resta = Vector()
        for(i in arreglo.indices)
            resta.arreglo[i] = arreglo[i] - vector2.arreglo[i]
        return resta
    }

    operator fun times(vector2: Vector): Vector {
        var producto = Vector()
        for(i in arreglo.indices)
            producto.arreglo[i] = arreglo[i] * vector2.arreglo[i]
        return producto
    }

    operator fun div(vector2: Vector): Vector {
        var division = Vector()
        for(i in arreglo.indices)
            division.arreglo[i] = arreglo[i] / vector2.arreglo[i]
        return division
    }
}

fun main(args: Array<String>) {
    val vec1 = Vector()
    vec1.cargar()
    val vec2 = Vector()
    vec2.cargar()
    vec1.imprimir()
```

```
    vec2.imprimir()
    val vecSuma = vec1 + vec2
    println("Suma componente a componente de los dos vectores")
    vecSuma.imprimir()
    val vecResta = vec1 - vec2
    println("La resta componente a componente de los dos vectores")
    vecResta.imprimir()
    val vecProducto = vec1 * vec2
    println("El producto componente a componente de los dos vectores")
    vecProducto.imprimir()
    val vecDivision = vec1 / vec2
    println("La división componente a componente de los dos vectores")
    vecDivision.imprimir()
}
```

Para sobrecargar el operador + debemos implementar el método plus que reciba un único parámetro, en este caso de tipo Vector y retorne otro objeto de la clase Vector.

Debemos anteceder a la palabra clave fun la palabra clave operator.
Dentro del método creamos un objeto de la clase Vector y procedemos a inicializar su propiedad arreglo con la suma componente a componente de los dos arreglos de cada objeto:

```
    operator fun plus(vector2: Vector): Vector {
        var suma = Vector()
        for(i in arreglo.indices)
            suma.arreglo[i] = arreglo[i] + vector2.arreglo[i]
        return suma
    }
```

Para llamar a este método en la main utilizamos el operador + :

```
    val vecSuma = vec1 + vec2
```

La otra sintaxis que podemos utilizar es la ya conocida de invocar el método:

```
    val vecSuma = vec1.plus(vec2)
```

Con esto podemos comprobar que el programa queda más legible utilizando el operador +.

Pensemos que si tenemos que sumar 4 vectores la sintaxis sería:

```
    val vecSuma = vec1 + vec2 + vec3 + vec4
```

En algoritmos complejos puede simplificar mucho nuestro código el uso correcto de la sobrecarga de operadores.

Podemos sobrecargar un operador con objetos de distinta clase.

Problema 2

Declarar una clase llamada Vector que administre un array de 5 elementos de tipo entero y cargar valores aleatorios entre 1 y 10. Sobrecargar el operador * que permita multiplicar un Vector con un número entero (se debe multiplicar cada componente del arreglo por el entero)

Proyecto163 - Principal.kt

```
class Vector {
    val arreglo = IntArray(5)

    fun cargar() {
        for(i in arreglo.indices)
            arreglo[i] = (Math.random() * 11 + 1).toInt()
    }

    fun imprimir() {
        for(elemento in arreglo)
            print("$elemento ")
        println()
    }

    operator fun times(valor: Int): Vector {
        var suma = Vector()
        for(i in arreglo.indices)
            suma.arreglo[i] = arreglo[i] * valor
        return suma
    }
}

fun main(args: Array<String>) {
    val vec1 = Vector()
    vec1.cargar()
    vec1.imprimir()
    println("El producto de un vector con el número 10 es")
    val vecProductoEnt = vec1 * 10
    vecProductoEnt.imprimir()
}
```

En este problema para sobrecargar el operador * de un Vector por un tipo entero al método debe llegar un tipo de dato Int:

```
    operator fun times(valor: Int): Vector {
        var suma = Vector()
        for(i in arreglo.indices)
            suma.arreglo[i] = arreglo[i] * valor
        return suma
    }
```

En la función main cuando procedemos a multiplicar un objeto de la clase Vector por un tipo de dato Int debemos hacerlo en este orden:

```
    println("El producto de un vector con el número 10 es")
    val vecProductoEnt = vec1 * 10
    vecProductoEnt.imprimir()
```

Si invertimos el producto, es decir un Int por un Vector debemos definir el método times en la clase Int. El programa completo luego quedaría:

```
class Vector {
    val arreglo = IntArray(5)

    fun cargar() {
        for(i in arreglo.indices)
            arreglo[i] = (Math.random() * 11 + 1).toInt()
    }

    fun imprimir() {
        for(elemento in arreglo)
            print("$elemento ")
        println()
    }

    operator fun times(valor: Int): Vector {
        var suma = Vector()
        for(i in arreglo.indices)
            suma.arreglo[i] = arreglo[i] * valor
        return suma
    }
}

operator fun Int.times(vec: Vector): Vector {
    var suma = Vector()
    for(i in vec.arreglo.indices)
        suma.arreglo[i] = vec.arreglo[i] * this
    return suma
}

fun main(args: Array<String>) {
    val vec1 = Vector()
    vec1.cargar()
    vec1.imprimir()
    println("El producto de un vector con el número 10 es")
    val vecProductoEnt = 10 * vec1
    vecProductoEnt.imprimir()
    println("El producto de un entero 5 por un vector es")
    val vecProductoEnt2 = vec1 * 5
    vecProductoEnt2.imprimir()

}
```

Utilizamos el concepto de funciones de extensión que vimos anteriormente:

```
operator fun Int.times(vec: Vector): Vector {
    var suma = Vector()
    for(i in vec.arreglo.indices)
        suma.arreglo[i] = vec.arreglo[i] * this
    return suma
}
```

Problema 3

Declarar una clase llamada Vector que administre un array de 5 elementos de tipo entero y cargar valores aleatorios entre 1 y 10. Sobrecargar los operadores ++ y -- (se debe incrementar o disminuir en uno cada elemento del arreglo)

Proyecto164 - Principal.kt

```
class Vector {
    val arreglo = IntArray(5)

    fun cargar() {
        for(i in arreglo.indices)
            arreglo[i] = (Math.random() * 11 + 1).toInt()
    }

    fun imprimir() {
        for(elemento in arreglo)
            print("$elemento ")
        println()
    }

    operator fun inc(): Vector {
        var suma1 = Vector()
        for(i in arreglo.indices)
            suma1.arreglo[i] = arreglo[i] + 1
        return suma1
    }

    operator fun dec(): Vector {
        var resta1 = Vector()
        for(i in arreglo.indices)
            resta1.arreglo[i] = arreglo[i] - 1
        return resta1
    }
}


fun main(args: Array<String>) {
    var vec1 = Vector()
    vec1.cargar()
    println("Vector original")
    vec1.imprimir()
    vec1++
    println("Luego de llamar al operador ++")
    vec1.imprimir()
    vec1--
    println("Luego de llamar al operador --")
    vec1.imprimir()
}
```

Cuando queremos sobrecargar los operadores ++ y -- debemos implementar los métodos inc y dec. Estos métodos no reciben ningún parámetro y retornan objetos de la clase Vector incrementando en uno o disminuyendo en uno cada componente del arreglo:

```
    operator fun inc(): Vector {
        var suma1 = Vector()
        for(i in arreglo.indices)
            suma1.arreglo[i] = arreglo[i] + 1
        return suma1
    }

    operator fun dec(): Vector {
        var resta1 = Vector()
        for(i in arreglo.indices)
            resta1.arreglo[i] = arreglo[i] - 1
        return resta1
    }
```

Cuando llamamos a los operadores ++ y -- el objeto que devuelve se asigna a la variable por la que llamamos, esto hace necesario definir el objeto de la clase Vector con la palabra clave var:

```
    var vec1 = Vector()
    vec1.cargar()
    println("Vector original")
    vec1.imprimir()
```

Una vez llamado al operador ++:

```
    vec1++
```

Ahora vec1 tiene la referencia al objeto que se creó en el método inc:

```
    println("Luego de llamar al operador ++")
    vec1.imprimir()
```

No hay que hacer cambios si necesitamos utilizar notación prefija con los operadores ++ y --:

```
    ++vec1
    --vec1
```

Sobrecarga de operadores > >= y < <=

Para sobrecargar estos operadores debemos implementar el método compareTo.

Problema 4

Implementar una clase llamada Persona que tendrá como propiedades su nombre y edad. Sobrecargar los operadores > >= y < <= .

Proyecto165 - Principal.kt

```
class Persona (val nombre: String, val edad: Int) {

    fun imprimir() {
        println("Nombre: $nombre y tiene una edad de $edad")
    }

    operator fun compareTo(per2: Persona): Int {
        return when {
            edad < per2.edad -> -1
            edad > per2.edad -> 1
            else -> 0
        }
    }
}

fun main(parametro: Array<String>) {
    val persona1 = Persona("Juan", 30)
    persona1.imprimir()
    val persona2 = Persona("Ana", 30)
    persona2.imprimir()
    println("Persona mayor")
    when {
        persona1 > persona2 -> persona1.imprimir()
        persona1 < persona2 -> persona2.imprimir()
        else -> println("Tienen la misma edad")
    }
}
```

El método compareTo debe retornar un Int, indicando que si devuelve un 0 las dos personas tienen la misma edad. Si retornar un 1 la persona que está a la izquierda del operador es mayor de edad y viceversa.:

```
    operator fun compareTo(per2: Persona): Int {
        return when {
            edad < per2.edad -> -1
            edad > per2.edad -> 1
            else -> 0
        }
    }
```

Luego podemos preguntar con estos operadores para objetos de la clase Persona cual tiene una edad mayor:

```
    when {
        persona1 > persona2 -> persona1.imprimir()
        persona1 < persona2 -> persona2.imprimir()
        else -> println("Tienen la misma edad")
    }
```

Sobrecarga de operadores de subíndices

En Kotlin podemos sobrecargar el manejo de subíndices implementando los métodos get y set.

```
la expresión                se traduce
a[i] = b                    a.set(i, b)
a[i, j] = b                 a.set(i, j, b)
a[i_1, ..., i_n] = b        a.set(i_1, ..., i_n, b)

a[i]                        a.get(i)
a[i, j]                     a.get(i, j)
a[i_1, ..., i_n]            a.get(i_1, ..., i_n)
```

Problema 5

Implementar una clase TaTeTi que defina una propiedad para el tablero que sea un IntArray de 9 elementos con valor cero.
Hay dos jugadores que disponen fichas, el primer jugador carga el 1 y el segundo carga un 2.
Mediante sobrecarga de operadores de subíndices permitir asignar la fichas a cada posición del tablero mediante dos subíndices que indiquen la fila y columna del tablero.

Proyecto166 - Principal.kt

```
class TaTeTi {
    val tablero = IntArray(9)

    fun imprimir() {
        for(fila in 0..2) {
            for (columna in 0..2)
                print("${this[fila, columna]} ")
            println()
        }
        println()
    }

    operator fun set(fila: Int, columna: Int, valor: Int){
        tablero[fila*3 + columna] = valor
        imprimir()
    }

    operator fun get(fila: Int, columna: Int): Int{
        return tablero[fila*3 + columna]
    }
}

fun main(args: Array<String>) {
    val juego = TaTeTi()
    juego[0, 0] = 1
    juego[0, 2] = 2
    juego[2, 0] = 1
    juego[1, 2] = 2
    juego[1, 0] = 1
}
```

La propiedad tablero almacena las nueve casillas del juego de Taterí en un arreglo. Para que sea más intuitivo la carga de fichas en el juego de tateti procedemos a sobrecargar el operador de subíndices con fila y columna.

La idea es que podamos indicar una fila, columna y la ficha que se carga con la sintaxis:

```
    juego[0, 0] = 1
    juego[0, 2] = 2
    juego[2, 0] = 1
    juego[1, 2] = 2
    juego[1, 0] = 1
```

Cada asignación en realidad llama al método set, que además de cargar la ficha pasa a imprimir el tablero:

```
    operator fun set(fila: Int, columna: Int, valor: Int){
        tablero[fila*3 + columna] = valor
        imprimir()
    }
```

La impresión del tablero procede a acceder mediante this a la sobrecarga del operador de subíndices accediendo al método get:

```
    fun imprimir() {
        for(fila in 0..2) {
            for (columna in 0..2)
                print("${this[fila, columna]} ")
            println()
        }
        println()
    }
```

La sobrecarga para recuperar el valor almacenado se hace implementando el método get:

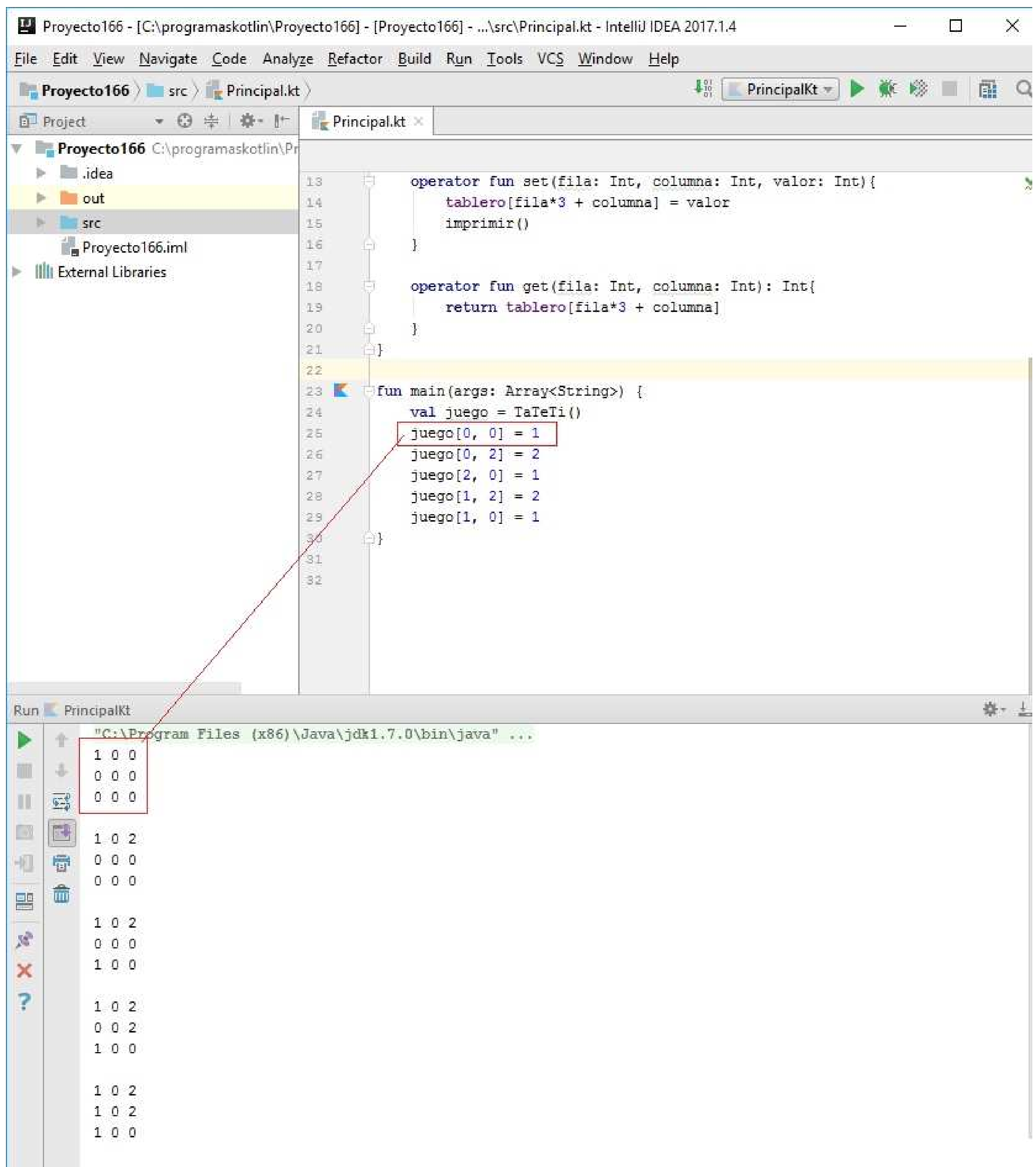
```
    operator fun get(fila: Int, columna: Int): Int{
        return tablero[fila*3 + columna]
    }
```

Recordar que la sobrecarga de operadores tiene por objetivo hacer más legible nuestro programa, acá lo logramos porque es muy fácil entender como cargamos las fichas en el tablero con la asignación:

```
    val juego = TaTeTi()
    juego[0, 0] = 1
```

Tengamos siempre en cuenta que cuando se sobrecarga un operador en realidad se está llamando un método de la clase.

La ejecución de este programa nos muestra en pantalla los estados sucesivos del tablero de Tatetí a medida que le asignamos piezas:



Sobrecarga de paréntesis

En Kotlin también podemos sobrecargar los paréntesis implementando el método invoke.

```
la expresión            se traduce
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)
```

Problema 6

Plantear una clase Dados que administre 10 valores de dados en un arreglo de tipo IntArray. Sobrecargar el operador de paréntesis para la clase y acceder mediante una posición al valor de un dado específico.

Proyecto167 - Principal.kt

```
class Dados (){

    val arreglo = IntArray(10)

    fun tirar() {
        for(i in arreglo.indices)
            arreglo[i] = ((Math.random() * 6) + 1).toInt()
    }

    operator fun invoke(nro: Int) = arreglo[nro]
}

fun main(args: Array<String>) {
    var dados = Dados()
    dados.tirar()
    println(dados(0))
    println(dados(1))
    for(i in 2..9)
        println(dados(i))
}
```

Declaramos un arreglo de 10 enteros y guardamos valores aleatorios entre 1 y 6:

```
class Dados (){

    val arreglo = IntArray(10)

    fun tirar() {
        for(i in arreglo.indices)
            arreglo[i] = ((Math.random() * 6) + 1).toInt()
    }
```

En la función main creamos el objeto de la clase Dados y llamamos al método tirar:

```
    var dados = Dados()
    dados.tirar()
```

Luego si disponemos el nombre del objeto y entre paréntesis pasamos un entero se llamará al método invoke y retornará en este caso un entero que representa el valor del dado para dicha posición:

```
    println(dados(0))
    println(dados(1))
    for(i in 2..9)
        println(dados(i))
```

En la clase Dados especificamos la sobrecarga del operador de paréntesis con la implementación del método invoke:

```
    operator fun invoke(nro: Int) = arreglo[nro]
```

Para este problema podíamos también haber sobrecargado el operador de corchetes como vimos anteriormente.

La elección de que operador sobrecargar dependerá del problema a desarrollar y ver con cual queda más claro su empleo.

Sobrecarga de operadores += -= *= /= %=

Los métodos que se invocan para estos operadores son:

```
la expresión       se traduce
a += b             a.plusAssign(b)
a -= b             a.minusAssign(b)
a *= b             a.timesAssign(b)
a /= b             a.divAssign(b)
a %= b             a.modAssign(b)
```

Problema 7

Declarar una clase llamada Vector que administre un array de 5 elementos de tipo entero.
Sobrecargar el operador +=
En la main definir una serie de objetos de la clase y emplear el operador +=

Proyecto168 - Principal.kt

```
class Vector {
    val arreglo = IntArray(5, {it})

    fun imprimir() {
        for (elemento in arreglo)
            print("$elemento ")
        println()
    }

    operator fun plusAssign(vec2: Vector) {

        for(i in arreglo.indices)
            arreglo[i] += vec2.arreglo[i]
    }
}

fun main(args: Array<String>) {
    val vec1 = Vector()
    vec1.imprimir()
    val vec2 = Vector()
    vec2.imprimir()
    vec1 += vec2
    vec1.imprimir()
}
```

Para sobrecargar el operador += en la clase Vector definimos el método plusAssign:

```
    operator fun plusAssign(vec2: Vector) {

        for(i in arreglo.indices)
            arreglo[i] += vec2.arreglo[i]
    }
```

Al método llega un objeto de la clase Vector que nos sirve para acceder al arreglo contenido en el mismo.

En la función main definimos dos objetos de la clase Vector y procedemos a acumular en vec1 el contenido de vec2 mediante el operador +=:

```
fun main(args: Array<String>) {
    val vec1 = Vector()
    vec1.imprimir()
    val vec2 = Vector()
    vec2.imprimir()
    vec1 += vec2
    vec1.imprimir()
}
```

Sobrecarga de operadores in y !in

Los métodos que se invocan para estos operadores son:

```
la expresión        se traduce
a in b              b.contains(a)
a !in b            !b.contains(a)
```

Problema 8

Plantear un data class Alumno que almacene su número de documento y su nombre. Luego en una clase Curso definir un Array con 3 objetos de la clase Alumno. Sobrecargar el operador in para verificar si un número de documento se encuentra inscripto en el curso.

Proyecto169 - Principal.kt

```
data class Alumno(val documento: Int, val nombre: String)

class Curso {
    val alumnos = arrayOf(Alumno(123456, "Marcos"),
                          Alumno(666666, "Ana"),
                          Alumno(777777, "Luis"))

    operator fun contains(documento: Int): Boolean {
        return alumnos.any {documento == it.documento}
    }
}

fun main(args: Array<String>) {
    val curso1 = Curso()
    if (123456 in curso1)
        println("El alumno Marcos está inscripto en el cuso")
    else
        println("El alumno Marcos no está inscripto en el cuso")
}
```

Declaramos un data class que representa un Alumno:

```
data class Alumno(val documento: Int, val nombre: String)
```

Declaramos la clase Cursos definiendo un Array con cuatro alumnos:

```
class Curso {
    val alumnos = arrayOf(Alumno(123456, "Marcos"), Alumno(666666, "Ana"), Alum
no(777777, "Luis"))
```

Sobrecargamos el operador in:

```
    operator fun contains(documento: Int): Boolean {
        return alumnos.any {documento == it.documento}
    }
```

Llega al método un entero que representa el número de documento a buscar dentro del arreglo de alumnos. En el caso que se encuentre retornamos un true, en caso negativo retornamos un falso.

Podemos escribir en forma más conciso el método contains:

```
    operator fun contains(documento: Int) = alumnos.any {documento == it.docume
nto}
```

En la main creamos un objeto de la clase curso y luego mediante el operador in podemos verificar si un determinado número de documento se encuentra inscripto en el curso de alumnos:

```
fun main(args: Array<String>) {
    val curso1 = Curso()
    if (123456 in curso1)
        println("El alumno Marcos está inscripto en el cuso")
    else
        println("El alumno Marcos no está inscripto en el cuso")
}
```

Retornar (index.php?inicio=30)

42 - Funciones: número variable de parámetros

En el lenguaje Kotlin un método de una clase o función puede recibir una cantidad variable de parámetros utilizando la palabra clave "vararg" previa al nombre del parámetro.

Desde donde llamamos a la función pasamos una lista de valores y lo recibe un único parámetro agrupando dichos datos en un arreglo.

```
fun imprimir(vararg nombres: String) {
    for(elemento in nombres)
        print("$elemento ")
    println()
}

fun main(args: Array<String>) {
    imprimir("Juan", "Ana", "Luis")
}
```

Como podemos observar utilizamos vararg previo al nombre del parámetro y el tipo de dato que almacenará el arreglo:

```
fun imprimir(vararg nombres: String) {
    for(elemento in nombres)
        print("$elemento ")
    println()
}
```

Dentro de la función tratamos al parámetro nombres como un arreglo.

Cuando llamamos a la función imprimir no enviamos un arreglo sino una lista de String, el compilador se encarga de transformar esa lista a un arreglo.

Problema 1

Elaborar una función que reciba una cantidad variable de enteros y nos retorne su suma.

Proyecto170 - Principal.kt

```
fun sumar(vararg numeros: Int): Int {
    var suma = 0
    for(elemento in numeros)
        suma += elemento
    return suma
}

fun main(args: Array<String>) {
    val total = sumar(10, 20, 30, 40, 50)
    println(total)
}
```

La función sumar recibe un parámetro con un número variable de elementos, dentro del algoritmo recorremos el arreglo, sumamos sus elementos y retornamos la suma:

```
fun sumar(vararg numeros: Int): Int {
    var suma = 0
    for(elemento in numeros)
        suma += elemento
    return suma
}
```

Desde la main llamamos a sumar pasando en este caso 5 enteros:

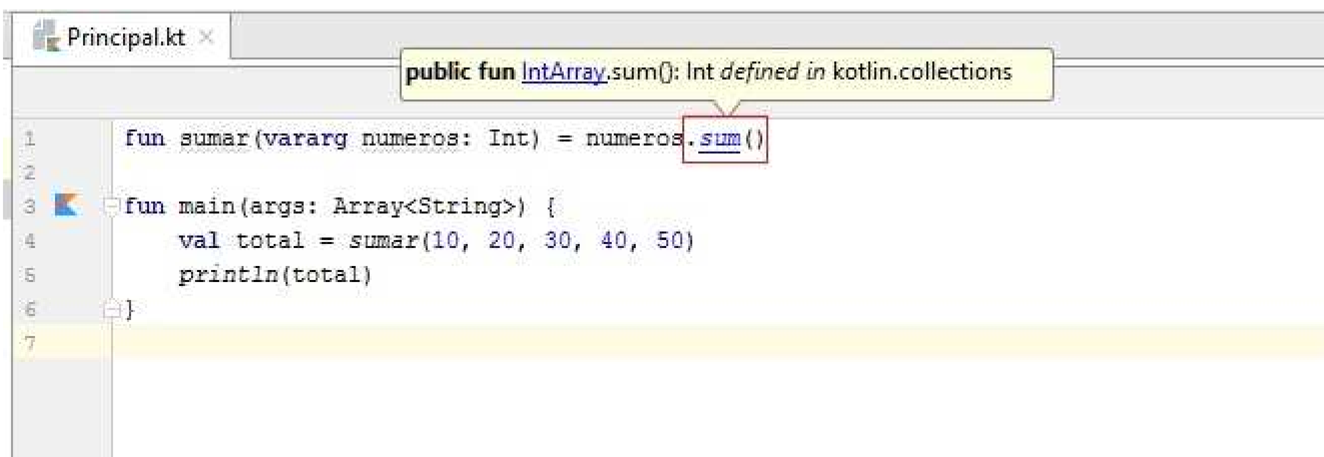
```
    val total = sumar(10, 20, 30, 40, 50)
```

Recordemos que Kotlin tiene como principio permitir implementar algoritmos muy concisos, la función sumar podemos codificarla en forma más concisa con la sintaxis:

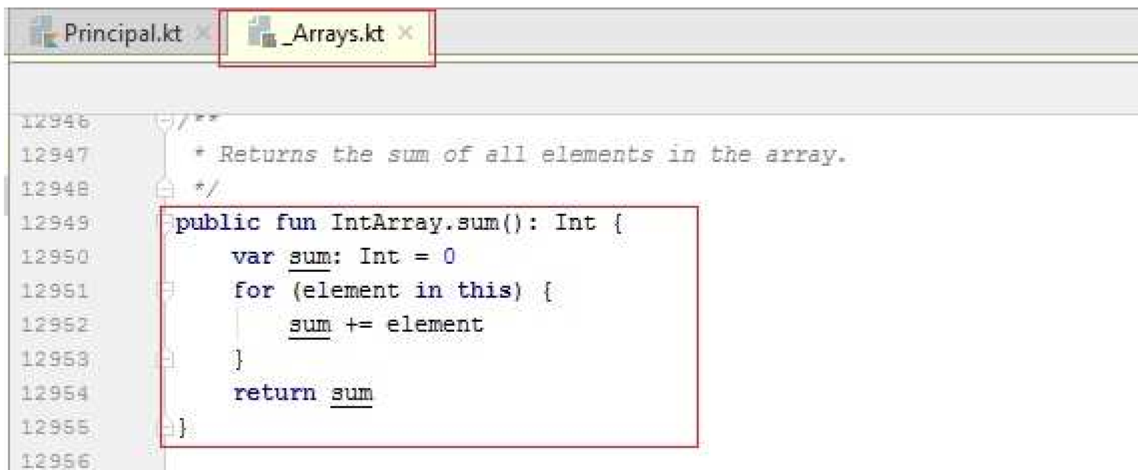
```
fun sumar(vararg numeros: Int) = numeros.sum()
```

Utilizamos una función con expresión única que calcula la suma del arreglo llamando al método sum del arreglo.

En el entorno de programación IntelliJ IDEA se presionamos la tecla "Control", disponemos la flecha del mouse sobre un método:



Luego de presionar el botón del mouse se abrirá otra pestaña donde está implementado el método sum:



Esto nos permite desplazarnos en nuestro programa en forma muy rápida y poder analizar algoritmos que están definidos en la librería estándar de Kotlin.

Solo un parámetro de una función puede ser de tipo vargar y normalmente es el último.

Problema 2

Elaborar una función que reciba como primer parámetro que tipo de operación queremos hacer con los siguientes datos enteros que le enviamos.

Proyecto171 - Principal.kt

```
enum class Operacion{
    SUMA,
    PROMEDIO
}

fun operar(tipoOperacion: Operacion, vararg arreglo: Int): Int {
    when (tipoOperacion) {
        Operacion.SUMA -> return arreglo.sum()
        Operacion.PROMEDIO -> return arreglo.average().toInt()
    }
}

fun main(args: Array<String>) {
    val resultado1 = operar(Operacion.SUMA, 10, 20, 30)
    println("La suma es $resultado1")
    val resultado2 = operar(Operacion.PROMEDIO, 10, 20, 30)
    println("El promedio es $resultado2")
}
```

Declaramos un enum class con dos valores posibles:

```
enum class Operacion{
    SUMA,
    PROMEDIO
}
```

A la función operar le enviamos como primer parámetro un tipo de dato Operacion y como segundo parámetro serán la lista de enteros a procesar:

```
fun operar(tipoOperacion: Operacion, vararg arreglo: Int): Int {
    when (tipoOperacion) {
        Operacion.SUMA -> return arreglo.sum()
        Operacion.PROMEDIO -> return arreglo.average().toInt()
    }
}
```

Cuando llamamos desde la main a la función operar debemos pasar como primer dato el tipo de operación que queremos hacer con la lista de enteros a enviarle:

```
fun main(args: Array<String>) {
    val resultado1 = operar(Operacion.SUMA, 10, 20, 30)
    println("La suma es $resultado1")
    val resultado2 = operar(Operacion.PROMEDIO, 10, 20, 30)
    println("El promedio es $resultado2")
}
```

La conveniencia de que el parámetro vararg sea el último es que si no respetamos esto estaremos obligado a utilizar la llamada a la función con argumentos nombrados. El programa anterior disponiendo primero el vararg será:

```
enum class Operacion{
    SUMA,
    PROMEDIO
}

fun operar(vararg arreglo: Int, tipoOperacion: Operacion): Int {
    when (tipoOperacion) {
        Operacion.SUMA -> return arreglo.sum()
        Operacion.PROMEDIO -> return arreglo.average().toInt()
    }
}

fun main(args: Array<String>) {
    val resultado1 = operar( 10, 20, 30, tipoOperacion = Operacion.SUMA)
    println("La suma es $resultado1")
    val resultado2 = operar(10, 20, 30, tipoOperacion = Operacion.PROMEDIO)
    println("El promedio es $resultado2")
}
```

Debemos en forma obligatoria nombrar el segundo parámetro en la llamada:

```
    val resultado1 = operar( 10, 20, 30, tipoOperacion = Operacion.SUMA)
```

Operador spread

Si una función recibe un parámetro de tipo vararg y desde donde la llamamos queremos enviarle un arreglo debemos indicarle al compilador tal situación, Con el primer ejemplo que vimos el código quedaría:

```
fun imprimir(vararg nombres: String) {
    for(elemento in nombres)
        print("$elemento ")
    println()
}

fun main(args: Array<String>) {
    val personas = arrayOf("Juan", "Ana", "Luis")
    imprimir(*personas)
}
```

Es decir le antecedemos el caracter * previo al arreglo que le enviamos a la función.

Problema propuesto

- Confeccionar una función que reciba una serie de edades y nos retorne la cantidad que son mayores o iguales a 18 (como mínimo se envía un entero a la función)

Solución

Retornar (index.php?inicio=30)

43 - Valores nulos en variables

Hasta ahora no analizamos como Kotlin trata los valores nulos (null) en la definición de variables.

En Kotlin se trata en forma separada las variables que permiten almacenar un valor null y aquellas que por su naturaleza no lo pueden almacenar.

Hemos trabajado hasta ahora con variables que no pueden almacenar un valor null, fácilmente lo podemos comprobar con las siguientes líneas de código:

```
1 fun main(args: Array<String>) {
2     var nombre: String
3     nombre = "Juan"
4     nombre = null
5 Null can not be a value of a non-null type String
6
```

La variable nombre debe almacenar una cadena de caracteres y por definición no puede almacenar un valor null.

Podemos perfectamente asignarle otro objeto de tipo String, pero no el valor null:

```
    var nombre: String
    nombre = "Juan"
    nombre = "Ana"
```

Lo mismo ocurre si definimos variables de tipo Int, Float, IntArray etc:

```
    var telefono: Int
    telefono = null // error de compilación
    var arreglo: IntArray = IntArray(5)
    arreglo = null // error de compilación
```

Para permitir que una variable contenga un valor null, debemos agregar el caracter "?" en el momento que definimos el tipo de la variable (con esto indicamos al compilador de Kotlin que son variables que permiten almacenar el valor null):

```
fun main(args: Array<String>) {
    var telefono: Int?
    telefono = null
    var arreglo: IntArray? = IntArray(5)
    arreglo = null
}
```

Cuando trabajamos con variables que pueden almacenar valores nulos nuestro código debe verificar el valor que almacena la variable:

```
fun main(args: Array<String>) {
    var nombre: String
    print("Ingrese su nombre:")
    nombre = readLine()!!
    println("El nombre ingresado es $nombre")}
```

Hasta ahora para hacer más simples nuestros programas cuando cargábamos datos por teclado llamábamos a la función readLine() y el operador !! al final,esto debido a que la función readLine retorna un tipo de dato String?:

```
public fun readLine(): String? = stdin.readLine()
```

Si la función readLine retorna un String? no lo podemos almacenar en una variable String. Para poder copiar un dato que puede almacenar un valor null en otro que no lo puede hacer utilizamos el !! y debe evitarse en lo posible.

Podemos ahora modificar el programa anterior y definir una variable String?:

```
fun main(args: Array<String>) {
    var nombre: String?
    print("Ingrese su nombre:")
    nombre = readLine()
    println("El nombre ingresado es $nombre")
}
```

Como vemos ahora no disponemos el operador !! al final de la llamada a readLine.

Control de nulos

Cuando trabajamos con variables que pueden almacenar valor nulo podemos mediante if verificar si su valor es distinto a null.

```
fun main(args: Array<String>) {
    var cadena1: String? = null
    var cadena2: String? = "Hola"
    if (cadena1 != null)
        println("cadena1 almacena $cadena1")
    else
        println("cadena1 apunta a null")
    if (cadena2 != null)
        println("cadena2 almacena $cadena2")
    else
        println("cadena2 apunta a null")
}
```

Mediante if verificamos si la variable almacena un null o un valor distinto a null y actuamos según el valor almacenado:

```
    if (cadena1 != null)
        println("cadena1 almacena $cadena1")
    else
        println("cadena1 apunta a null")
```

Podemos intentar llamar a sus métodos y propiedades sin que se genere un error cuando disponemos el operador "?" seguido a la variable:

```
    var cadena1: String? = null
    println(cadena1?.length)
```

Se imprime en pantalla un null pero no se genera un error ya que no se accede a la propiedad length de la clase String, esto debido a que cadena1 almacena un null.

Esta sintaxis es muy conveniente si tenemos llamadas como:

```
if (empleado1?.datosPersonales?.telefono? != null)
```

Cualquiera de los objetos que apunte a null luego el if se verifica falso.

Un ejemplo de acceso sería:

```
data class DatosPersonales(val nombre: String, val telefono: Int?)
data class Empleado (val nroEmpleado: Int, val datosPersonales: DatosPersonale
s?)

fun main(args: Array<String>) {
    var empleado1: Empleado?
    empleado1= Empleado(100, DatosPersonales("Juan", null))
    if (empleado1?.datosPersonales?.telefono == null )
        println("El empleado no tiene telefono")
    else
        println("El telefono del empleado es ${empleado1?.datosPersonales?.tele
fono}")
}
```

Con esta sintaxis decimos que la variable empleado1 puede almacenar null (por ejemplo si la empresa no tiene empleados):

```
    var empleado1: Empleado?
```

La propiedad datosPersonales de la clase Empleado puede almacenar null (por ejemplo si no tenemos los datos personales del empleado):

```
data class Empleado (val nroEmpleado: Int, val datosPersonales: DatosPersonale
s?)
```

La clase DatosPersonales tiene una propiedad telefono que puede almacenar null (por ejemplo si el empleado no tiene teléfono):

```
data class DatosPersonales(val nombre: String, val telefono: Int?)
```

Para imprimir el teléfono de un empleado debemos controlar si existe el empleado, si tiene almacenado sus datos personales y si tiene teléfono, una aproximación con varios if sería:

```
    if (empleado1 != null)
        if (empleado1.datosPersonales != null)
            if (empleado1.datosPersonales?.telefono != null)
                println("El telefono del empleado es ${empleado1.datosPersonale
s?.telefono}")
```

Analizando si tienen almacenado valores distintos a null hasta llegar a la propiedad del telefono para mostrarla.

Pero en Kotlin la sintaxis más concisa para acceder al teléfono es:

```
    if (empleado1?.datosPersonales?.telefono == null )
        println("El empleado no tiene telefono")
    else
        println("El telefono del empleado es ${empleado1?.datosPersonales?.tele
fono}")
```

Ya sea que empleado1 almacene null o la propiedad datosPersonales almacene null luego retorna null la expresión.

Retornar (index.php?inicio=30)

44 - Colecciones

Kotlin provee un amplio abanico de clases para administrar colecciones de datos.

Ya vimos algunas de las clases que permiten administrar colecciones:

- Para tipos de datos básicos (se encuentran optimizadas)

```
ByteArray
ShortArray
LongArray
FloatArray
DoubleArray
BooleanArray
CharArray
```

- Para cualquier tipo de dato:

```
Array
```

- ```
  List
  MutableList
  ```

- ```
  Map
  MutableMap
  ```

- ```
  Set
  MutableSet
  ```

Ya trabajamos en conceptos anteriores con los arreglos que almacenan tipos de datos básicos y con la clase Array que nos permite almacenar cualquier tipo de dato. En los próximos conceptos veremos las clases List, Map y Set.

Para administrar listas (List), mapas (Map) y conjuntos (Set) hay dos grandes grupos: mutables (es decir que luego de ser creada podemos agregar, modificar y borrar elementos) e inmutables (una vez creada la colección no podemos modificar la estructura)

Retornar (index.php?inicio=30)
  ```

45 - Colecciones - List y MutableList

Una lista es una colección ordenada de datos. Se puede recuperar un elemento por la posición que se encuentra en la colección.

Podemos crear en Kotlin tanto listas inmutables como mutables.

Creación de una lista inmutable.

Problema 1

Crear una lista inmutable con los días de la semana. Probar las propiedades y métodos principales para administrar la lista.

Proyecto173 - Principal.kt

```
fun main(args: Array<String>) {
    var lista1: List<String> = listOf("lunes", "martes", "miercoles", "jueves",
"viernes", "sábado", "domingo")
    println("Imprimir la lista completa")
    println(lista1)
    println("Imprimir el primer elemento de la lista")
    println(lista1[0])
    println("Imprimir el primer elemento de la lista")
    println(lista1.first())
    println("Imprimir el último elemento de la lista")
    println(lista1.last())
    println("Imprimir el último elemento de la lista")
    println(lista1[lista1.size-1])
    println("Imprimir la cantidad de elementos de la lista")
    println(lista1.size)
    println("Recorrer la lista completa con un for")
    for(elemento in lista1)
        print("$elemento ")
    println()
    println("Imprimir el elemento y su posición")
    for(posicion in lista1.indices)
        print("[$posicion]${lista1[posicion]} ")
}
```

Para crea una lista inmutable podemos llamar a la función listOf y pasar como parámetro los datos a almacenar, debemos indicar el tipo de datos que almacena luego de List:

```
    var lista1: List<String> = listOf("lunes", "martes", "miercoles", "jueves",
"viernes", "sábado", "domingo")
```

Una vez creada la lista no podemos modificar sus datos:

```
    lista1[0] = "domingo"
```

Tampoco podemos agregar elementos:

```
lista1.add("enero")
```

Lo que podemos hacer con una lista inmutable es acceder a sus elementos, por ejemplo recorrerla con un for:

```
    for(elemento in lista1)
        print("$elemento ")
```

Acceder a cualquier elemento por un subíndice:

```
    println("Imprimir el primer elemento de la lista")
    println(lista1[0])
```

Si queremos conocer todas las propiedades y métodos de List podemos visitar la documentación de la biblioteca estándar (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/index.html) de Kotlin.

Problema 2

Veamos otro ejemplo de crear una lista inmutable pero utilizando la función List a la cual le pasamos el tamaño de la lista y una función lambda indicando que valores almacenar:

Proyecto174 - Principal.kt

```
fun cargar(): Int {
    print("Ingrese un entero:")
    val valor = readLine()!!.toInt()
    return valor
}

fun main(args: Array<String>) {
    var lista1: List<Int> = List(5, {cargar()})
    println(lista1)
}
```

En este ejemplo creamos una lista inmutable llamando a la función List a la cual le pasamos en el primer parámetro el tamaño de la lista a crear y el segundo parámetro es la función lambda que se ejecutará para cada elemento.

En la función lambda llamamos a cargar pero podríamos haber codificado en dicho lugar la carga:

```
fun main(args: Array<String>) {
    var lista1: List<Int> = List(5) {
        print("Ingrese un entero:")
        val valor = readLine()!!.toInt()
        valor
    }
    println(lista1)
}
```

El valor que retorna la función lambda es el dato que se va almacenando en cada componente de la colección.

Creación de una lista mutable.

Problema 3

Crear una lista mutable con las edades de varias personas. Probar las propiedades y métodos principales para administrar la lista mutable.

Proyecto175 - Principal.kt

```
fun main(args: Array<String>) {
    val edades: MutableList<Int> = mutableListOf(23, 67, 12, 35, 12)
    println("Lista de edades")
    println(edades)
    edades[0] = 60
    println("Lista completa después de modificar la primer edad")
    println(edades)
    println("Primera edad almacenada en la lista")
    println(edades[0])
    println("Promedio de edades")
    println(edades.average())
    println("Agregamos una edad al final de la lista:")
    edades.add(50)
    println("Lista de edades")
    println(edades)
    println("Agregamos una edad al principio de la lista:")
    edades.add(0, 17)
    println("Lista de edades")
    println(edades)
    println("Eliminamos la primer edad de la lista:")
    edades.removeAt(0)
    println("Lista de edades")
    println(edades)
    print("Cantidad de personas mayores de edad:")
    val cant = edades.count { it >= 18 }
    println(cant)
    edades.removeAll { it == 12 }
    println("Lista de edades después de borrar las que tienen 12 años")
    println(edades)
    edades.clear()
    println("Lista de edades luego de borrar la lista en forma completa")
    println(edades)
    if (edades.isEmpty())
        println("No hay edades almacenadas en la lista")
}
```

Este ejemplo muestra como crear una lista mutable llamando a la función mutableListOf e indicando los valores iniciales:

```
    val edades: MutableList<Int> = mutableListOf(23, 67, 12, 35, 12)
```

Para agregar un nuevo elemento a la lista al final llamamos al método add:

```
    edades.add(50)
```

Pero podemos agregarlo en cualquier posición dentro de la lista mediante el método add con dos parámetros, en el primero indicamos la posición y en el segundo el valor a almacenar:

```
    edades.add(0, 17)
```

Mediante el método count y pasando una lambda podemos contar todos los elementos que cumplen una condición:

```
    print("Cantidad de personas mayores de edad:")
    val cant = edades.count { it >= 18 }
    println(cant)
```

También podemos eliminar todos los elementos de la lista que cumplen una determinada condición indicada una lambda:

```
    edades.removeAll { it == 12 }
    println("Lista de edades después de borrar las que tienen 12 años")
    println(edades)
```

Si queremos conocer todas las propiedades y métodos de MutableList podemos visitar la documentación de la biblioteca estándar (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/index.html) de Kotlin.

Problema 4

Crear una lista mutable de 20 elementos con valores aleatorios comprendidos entre 1 y 6.
Contar cuantos elementos tienen almacenado el valor 1.
Eliminar los elementos que almacenan un 6.

Proyecto176 - Principal.kt

```
fun main(args: Array<String>) {
    val lista: MutableList<Int> = MutableList(20) { ((Math.random() * 6) + 1).to
Int() }
    println("Lista completa")
    println(lista)
    val cant = lista.count { it == 1 }
    println("Cantidad de elementos con el valor 1: $cant")
    lista.removeAll { it == 6 }
    println("Lista luego de borrar los elementos con el valor 6")
    println(lista)
}
```

Para crear una lista de 20 elementos utilizamos la función MutableList y le pasamos en el primer parámetro la cantidad de elementos y en el segundo parámetro una lambda generando valores aleatorios comprendidos entre 1 y 6.

Para contar los elementos que almacenan un 1 utilizamos la función count y pasamos una lambda con la condición que se debe cumplir para ser contado:

```
val cant = lista.count { it == 1 }
```

Para eliminar llamamos al método removeAll:

```
lista.removeAll { it == 6 }
```

Problema 5

Implementar una clase llamada Persona que tendrá como propiedades su nombre y edad. Definir además dos métodos, uno que imprima las propiedades y otro muestre si es mayor de edad.

Definir una lista mutable de objetos de la clase Persona.
Imprimir los datos de todas las personas.
Imprimir cuantas personas son mayor de edad.

Proyecto177 - Principal.kt

```
class Persona (var nombre: String, var edad: Int) {

    fun imprimir() {
        println("Nombre: $nombre y tiene una edad de $edad")
    }

    fun esMayorEdad() {
        if (edad >= 18)
            println("Es mayor de edad $nombre")
        else
            println("No es mayor de edad $nombre")
    }
}

fun main(args: Array<String>) {
    val personas: MutableList<Persona>
    personas = mutableListOf(Persona("Juan", 22), Persona("Ana", 19), Persona("M
arcos", 12))
    println("Listado de todas personas")
    personas.forEach { it.imprimir() }
    val cant = personas.count { it.edad >= 18}
    println("La cantidad de personas mayores de edad es $cant")
}
```

Declaramos la clase Persona con dos propiedades y dos métodos:

```
class Persona (var nombre: String, var edad: Int) {

    fun imprimir() {
        println("Nombre: $nombre y tiene una edad de $edad")
    }

    fun esMayorEdad() {
        if (edad >= 18)
            println("Es mayor de edad $nombre")
        else
            println("No es mayor de edad $nombre")
    }
}
```

Definimos una lista mutable que almacena componentes de tipo Persona:

```
fun main(args: Array<String>) {
    val personas: MutableList<Persona>
```

Creamos la lista mediante el método mutableListOf y le pasamos la referencia de tres objetos de la clase Persona:

```
    personas = mutableListOf(Persona("Juan", 22), Persona("Ana", 19), Persona
("Marcos", 12))
```

Imprimimos todos los datos de las personas procesando cada objeto almacenado en la lista llamando al método forEach y en la función lambda llamando al método imprimir de cada persona:

```
    println("Listado de todas personas")
    personas.forEach { it.imprimir() }
```

Finalmente contamos la cantidad de personas mayores de edad:

```
    val cant = personas.count { it.edad >= 18}
    println("La cantidad de personas mayores de edad es $cant")
```

Problemas propuestos

- Crear una lista inmutable de 100 elementos enteros con valores aleatorios comprendidos entre 0 y 20.
  contar cuantos hay comprendidos entre 1 y 4, 5 y 8 y cuantos entre 10 y 13.
  Imprimir si el valor 20 está presente en la lista.

- Confeccionar una clase que represente un Empleado. Definir como propiedades su nombre y su sueldo.
  Definir una lista mutable con tres empleados.
  Imprimir los datos de los empleados.
  Calcular cuanto gasta la empresa en sueldos.
  Agregar un nuevo empleado a la lista y calcular nuevamente el gasto en sueldos.
- Cargar por teclado y almacenar en una lista inmutable las alturas de 5 personas (valores Float)
  Obtener el promedio de las mismas. Contar cuántas personas son más altas que el promedio y cuántas más bajas.

Solución

Retornar (index.php?inicio=30)

46 - Colecciones - Map y MutableMap

La estructura de datos Map (Mapa) utiliza una clave para acceder a un valor. El subíndice puede ser cualquier tipo de clase, lo mismo que su valor

Podemos relacionarlo con conceptos que conocemos:

- Un diccionario tradicional que conocemos podemos utilizar un Map de Kotlin para representarlo. La clave sería la palabra y el valor sería la definición de dicha palabra.
- Una agenda personal también la podemos representar como un Map. La fecha sería la clave y las actividades de dicha fecha sería el valor.
- Un conjunto de usuarios de un sitio web podemos almacenarlo en un Map. El nombre de usuario sería la clave y como valor podríamos almacenar su mail, clave, fechas de login etc.

Hay muchos problemas de la realidad que se pueden representar mediante un Map.

Problema 1

En el bloque principal del programa definir un Map inmutable que almacene los nombres de países como clave y como valor la cantidad de habitantes de dicho pais.
Probar distintos métodos y propiedades que nos provee la clase Map.

Proyecto181 - Principal.kt

```
fun main(args: Array<String>) {
    val paises: Map<String, Int> = mapOf( Pair("argentina", 40000000),
                                          Pair("españa", 46000000),
                                          Pair("uruguay", 3400000))
    println("Listado completo del Map")
    println(paises)
    for ((clave, valor) in paises)
        println("Para la clave $clave tenemos almacenado $valor")
    println("La cantidad de elementos del mapa es ${paises.size}")
    val cantHabitantes1: Int? = paises["argentina"]
    if (cantHabitantes1 != null)
        println("La cantidad de habitantes de argentina es $cantHabitantes1")
    val cantHabitantes2: Int? = paises["brasil"]
    if (cantHabitantes2 != null)
        println("La cantidad de habitantes de brasil es $cantHabitantes2")
    else
        println("brasil no se encuentra cargado en el Map")
    var suma = 0
    paises.forEach { suma += it.value }
    println("Cantidad total de habitantes de todos los paises es $suma")
}
```

Para crear un Map en Kotlin debemos definir una variable e indicar de que tipo de datos son la clave del mapa y el valor que almacena.
Mediante la función mapOf retornamos un Map indicando cada entrada en nuestro Map mediante un objeto Pair:

```
    val paises: Map<String, Int> = mapOf( Pair("argentina", 40000000),
                                          Pair("españa", 46000000),
                                          Pair("uruguay", 3400000))
```

Hemos creado un Map que almacena tres entradas.

La función println nos permite mostrar el Map en forma completo:

```
    println("Listado completo del Map")
    println(paises)
```

Si queremos recorrer e ir imprimiendo elemento por elemento las componentes del mapa podemos hacerlo mediante un for, en cada iteración recuperamos una clave y su valor:

```
    for ((clave, valor) in paises)
        println("Para la clave $clave tenemos almacenado $valor")
```

Como las listas la clase Map dispone de una propiedad size que nos retorna la cantidad de elementos del mapa:

```
    println("La cantidad de elementos del mapa es ${paises.size}")
```

Si necesitamos recuperar el valor para una determinada clave podemos hacerlo por medio de la sintaxis:

```
    val cantHabitantes1: Int? = paises["argentina"]
```

Como puede suceder que no exista la clave buscada en el mapa definimos la variable cantHabitantes1 de Int? ya que puede almacenar un null si no existe el pais buscado.

Luego con un if podemos controlar si se recuperó la cantidad de habitantes para el país buscado:

```
    if (cantHabitantes1 != null)
        println("La cantidad de habitantes de argentina es $cantHabitantes1")
    val cantHabitantes2: Int? = paises["brasil"]
    if (cantHabitantes2 != null)
        println("La cantidad de habitantes de brasil es $cantHabitantes2")
    else
        println("brasil no se encuentra cargado en el Map")
```

Finalmente para acumular la cantidad de habitantes de todos los paises podemos recorrer el Map y sumar el valor de cada componente:

```
    var suma = 0
    paises.forEach { suma += it.value }
    println("Cantidad total de habitantes de todos los paises es $suma")
```

Otra sintaxis común para crear el Map en Kotlin es:

```
    val paises: Map<String, Int> = mapOf( "argentina" to 40000000,
                                          "españa" to 46000000,
                                          "uruguay" to 3400000)
```

La función "to" ya veremos que mediante una definición "infix" podemos pasar un parámetro, luego el nombre de la función y finalmente el segundo parámetro. La función to ya existe y tiene esta sintaxis:

```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

Ya veremos más adelante funciones con notación infix y con parámetros genéricos. Podemos comprobar que la función retorna un objeto de la clase Pair.

Si queremos conocer todas las propiedades y métodos de Map podemos visitar la documentación de la biblioteca estándar (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/index.html) de Kotlin.

Problema 2

Crear un mapa que permita almacenar 5 artículos, utilizar como clave el nombre de productos y como valor el precio del mismo.
Desarrollar además las funciones de:
1) Imprimir en forma completa el diccionario
2) Mostrar la cantidad de artículos con precio superior a 20.

Proyecto182 - Principal.kt

```
fun imprimir(productos: Map<String, Float>) {
    for((clave, valor) in productos)
        println("$clave tiene un precio $valor")
    println();
}

fun cantidadPrecioMayor20(productos: Map<String, Float>) {
    val cant = productos.count{ it.value > 20}
    println("Cantidad de productos con un precio superior a 20: $cant")
}

fun main(args: Array<String>) {
    val productos: Map<String, Float> = mapOf("papas" to 12.5f,
                                              "manzanas" to 26f,
                                              "peras" to 31f,
                                              "mandarinas" to 15f,
                                              "pomelos" to 28f)
    imprimir(productos)
    cantidadPrecioMayor20(productos)
}
```

En este caso creamos un mapa cuya clave es de tipo String y su valor es un Float:

```
fun main(args: Array<String>) {
    val productos: Map<String, Float> = mapOf("papas" to 12.5f,
                                              "manzanas" to 26f,
                                              "peras" to 31f,
                                              "mandarinas" to 15f,
                                              "pomelos" to 28f)
```

Para mostrar el mapa en forma completo lo hacemos recorriendo por medio de un for:

```
fun imprimir(productos: Map<String, Float>) {
    for((clave, valor) in productos)
        println("$clave tiene un precio $valor")
    println();
}
```

Para contar la cantidad de productos que tienen un precio superior a 20 llamamos al método count y le pasamos una función lambda analizando el parámetro it en la propiedad value si cumple la condición de superar al valor 20:

```
fun cantidadPrecioMayor20(productos: Map<String, Float>) {
    val cant = productos.count{ it.value > 20}
    println("Cantidad de productos con un precio superior a 20: $cant")
}
```

Problema 3

Desarrollar una aplicación que nos permita crear un diccionario ingles/castellano. La clave es la palabra en ingles y el valor es la palabra en castellano.
Crear las siguientes funciones:
1) Cargar el diccionario (se ingresan por teclado la palabra en ingles y su traducción en castellano)
2) Listado completo del diccionario.
3) Ingresar por teclado una palabra en ingles y si existe en el diccionario mostrar su traducción.

Proyecto183 - Principal.kt

```
fun cargar(diccionario: MutableMap<String, String>) {
    do {
        print("Ingrese palabra en castellano:")
        val palCastellano = readLine()!!
        print("Ingrese palabra en ingles:")
        val palIngles = readLine()!!
        diccionario[palIngles] = palCastellano
        print("Continua cargando otra palabra en el diccionario? (si/no):")
        val fin = readLine()!!
    } while (fin=="si")
}

fun listado(diccionario: MutableMap<String,String>) {
    for((ingles, castellano) in diccionario)
        println("$ingles: $castellano")
    println()
}

fun consultaIngles(diccionario: MutableMap<String, String>) {
    print("Ingrese una palabra en ingles para verificar su traducción:")
    val ingles = readLine()
    if (ingles in diccionario)
        println("La traducción en castellano es ${diccionario[ingles]}")
    else
        println("No existe esa palabra en el diccionario")
}

fun main(args: Array<String>) {
    val diccionario: MutableMap<String, String> = mutableMapOf()
    cargar(diccionario)
    listado(diccionario)
    consultaIngles(diccionario)
}
```

En la función main definimos un MutableMap vacío:

```
fun main(args: Array<String>) {
    val diccionario: MutableMap<String, String> = mutableMapOf()
```

En la función cargar procedemos a ingresar una palabra en ingles y su traducción en castellano, para cargarla al Map procedemos a acceder por medio del subíndice:

```
        print("Ingrese palabra en castellano:")
        val palCastellano = readLine()!!
        print("Ingrese palabra en ingles:")
        val palIngles = readLine()!!
        diccionario[palIngles] = palCastellano
```

La función cargar finaliza cuando el operador carga un String distinto a "si":

```
        print("Continua cargando otra palabra en el diccionario? (si/no):")
        val fin = readLine()!!
    } while (fin=="si")
```

Para controlar si un Map contiene una determinada clave lo podemos hacer mediante el operador in:

```
    print("Ingrese una palabra en ingles para verificar su traducción:")
    val ingles = readLine()
    if (ingles in diccionario)
        println("La traducción en castellano es ${diccionario[ingles]}")
    else
        println("No existe esa palabra en el diccionario")
```

Si queremos conocer todas las propiedades y métodos de MutableMap podemos visitar la documentación de la biblioteca estándar (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/index.html) de Kotlin.

Problema 4

Confeccionar un programa que permita cargar un código de producto como clave en un Map. Guardar para dicha clave un objeto de la clase Producto que defina como propiedades su nombre del producto, su precio y cantidad en stock.

Implementar las siguientes actividades:
1) Carga de datos en el Map.
2) Listado completo de productos.
3) Consulta de un producto por su clave, mostrar el nombre, precio y stock.
4) Cantidad de productos que tengan un stock con valor cero.

Proyecto184 - Principal.kt

```
data class Producto(val nombre: String, val precio: Float, val cantidad: Int)

fun cargar(productos: MutableMap<Int, Producto>) {
    productos[1] = Producto("Papas", 13.15f, 200)
    productos[15] = Producto("Manzanas", 20f, 0)
    productos[20] = Producto("Peras", 3.50f, 0)
}

fun listadoCompleto(productos: MutableMap<Int, Producto>) {
    println("Listado completo de productos")
    for((codigo, producto) in productos)
        println("Código: $codigo  Descripción ${producto.nombre} Precio: ${produ
cto.precio} Stock Actual: ${producto.cantidad}")
    println()
}

fun consultaProducto(productos: MutableMap<Int, Producto>) {
    print("Ingrese el código de un producto:")
    val codigo = readLine()!!.toInt()
    if (codigo in productos)
        println("Nombre: ${productos[codigo]?.nombre} Precio: ${productos[codig
o]?.precio} Stock: ${productos[codigo]?.cantidad}")
    else
        println("No existe un producto con dicho código")
}

fun sinStock(productos: MutableMap<Int, Producto>) {
    val cant = productos.count { it.value.cantidad == 0 }
    println("Cantidad de artículos sin stock: $cant")
}

fun main(args: Array<String>) {
    val productos: MutableMap<Int, Producto> = mutableMapOf()
    cargar(productos);
    listadoCompleto(productos)
    consultaProducto(productos)
    sinStock(productos)
}
```

Definimos un Map mutable en la función main y llamamos a una serie de funciones donde en uno lo cargamos y en el resto procesamos sus elementos:

```
fun main(args: Array<String>) {
    val productos: MutableMap<Int, Producto> = mutableMapOf()
    cargar(productos);
    listadoCompleto(productos)
    consultaProducto(productos)
    sinStock(productos)
}
```

En la función de cargar llega el Map y agregamos tres productos:

```
fun cargar(productos: MutableMap<Int, Producto>) {
    productos[1] = Producto("Papas", 13.15f, 200)
    productos[15] = Producto("Manzanas", 20f, 0)
    productos[20] = Producto("Peras", 3.50f, 0)
}
```

La función que analiza la cantidad de productos sin stock lo hacemos llamando a count y pasando una función lambda que contará todos los productos cuya cantidad sea cero:

```
fun sinStock(productos: MutableMap<Int, Producto>) {
    val cant = productos.count { it.value.cantidad == 0 }
    println("Cantidad de artículos sin stock: $cant")
}
```

Problema 5

Se desea almacenar los datos de n alumnos (n se ingresa por teclado). Definir un MutableMap cuya clave sea el número de documento del alumno.
Como valor almacenar una lista con componentes de la clase Materia donde almacenamos nombre de materia y su nota.

Crear las siguientes funciones:
1) Carga de los alumnos (de cada alumno solicitar su dni y los nombres de las materias y sus notas)
2) Listado de todos los alumnos con sus notas
3) Consulta de un alumno por su dni, mostrar las materias que cursa y sus notas.

Proyecto185 - Principal.kt

```
data class Materia(val nombre: String, val nota: Int)

fun cargar(alumnos: MutableMap<Int, MutableList<Materia>>) {
    print("Cuantos alumnos cargará ?:")
    val cant = readLine()!!.toInt()
    for(i in 1..cant) {
        print("Ingrese dni:")
        val dni = readLine()!!.toInt()
        val listaMaterias = mutableListOf<Materia>()
        do {
            print("Ingrese materia del alumno:")
            val nombre = readLine()!!
            print("Ingrese nota:")
            val nota = readLine()!!.toInt()
            listaMaterias.add(Materia(nombre, nota))
            print("Ingrese otra nota (si/no):")
            val opcion = readLine()!!
        } while (opcion == "si")
        alumnos[dni] = listaMaterias
    }
}

fun imprimir(alumnos: MutableMap<Int, MutableList<Materia>>) {
    for((dni, listaMaterias) in alumnos) {
        println("Dni del alumno: $dni")
        for(materia in listaMaterias) {
            println("Materia: ${materia.nombre}")
            println("Nota: ${materia.nota}")
        }
        println()
    }
}

fun consultaPorDni(alumnos: MutableMap<Int, MutableList<Materia>>) {
    print("Ingrese el dni del alumno a consultar:")
    val dni = readLine()!!.toInt()
    if (dni in alumnos) {
        println("Cursa las siguentes materias")
        val lista = alumnos[dni]
        if (lista!=null)
            for((nombre, nota) in lista) {
                println("Nombre de materia: $nombre")
                println("Nota: $nota")
            }
    }
}

fun main(args: Array<String>) {
```

```
    val alumnos: MutableMap<Int, MutableList<Materia>> = mutableMapOf()
    cargar(alumnos)
    imprimir(alumnos)
    consultaPorDni(alumnos)
}
```

A medida que tenemos que representar conceptos más complejos necesitamos definir en este caso un Map cuya clave es un entero pero su valor es una lista mutable con elementos de la clase Materia:

```
fun main(args: Array<String>) {
    val alumnos: MutableMap<Int, MutableList<Materia>> = mutableMapOf()
```

En la función de cargar previa a un for solicitamos la cantidad de alumnos a almacenar en el mapa:

```
fun cargar(alumnos: MutableMap<Int, MutableList<Materia>>) {
    print("Cuantos alumnos cargará ?:")
    val cant = readLine()!!.toInt()
```

Luego mediante un for procedemos a cargar el dni del alumno y crear una lista mutable donde se almacenarán las materias y sus respectivas notas del alumno:

```
    for(i in 1..cant) {
        print("Ingrese dni:")
        val dni = readLine()!!.toInt()
        val listaMaterias = mutableListOf<Materia>()
```

Mediante una estructura repetitiva cargamos cada materia y nota hasta que finalizamos con ese alumno:

```
        do {
            print("Ingrese materia del alumno:")
            val nombre = readLine()!!
            print("Ingrese nota:")
            val nota = readLine()!!.toInt()
            listaMaterias.add(Materia(nombre, nota))
            print("Ingrese otra nota (si/no):")
            val opcion = readLine()!!
        } while (opcion == "si")
```

Cuando salimos del ciclo do/while procedemos a insertar la lista de materias en el mapa:

```
        alumnos[dni] = listaMaterias
    }
}
```

Para imprimir en forma completa el mapa lo recorremos con un for que rescata en cada ciclo el dni del alumno y la lista de materias que cursa:

```
fun imprimir(alumnos: MutableMap<Int, MutableList<Materia>>) {
    for((dni, listaMaterias) in alumnos) {
        println("Dni del alumno: $dni")
```

Mediante otro for interno recorremos la lista de materias de ese alumno y mostramos los nombres de materias y sus notas:

```
        for(materia in listaMaterias) {
            println("Materia: ${materia.nombre}")
            println("Nota: ${materia.nota}")
        }
        println()
    }
}
```

Por último la consulta de las materias que cursa un alumno se ingresa el dni y luego de controlar si está almacenado en el mapa procedemos a recorrer con un ciclo for todas las materias que cursa:

```
fun consultaPorDni(alumnos: MutableMap<Int, MutableList<Materia>>) {
    print("Ingrese el dni del alumno a consultar:")
    val dni = readLine()!!.toInt()
    if (dni in alumnos) {
        println("Cursa las siguentes materias")
        val lista = alumnos[dni]
        if (lista!=null)
            for((nombre, nota) in lista) {
                println("Nombre de materia: $nombre")
                println("Nota: $nota")
            }
    }
}
```

Problema propuesto

- Confeccionar una agenda. Utilizar un MutableMap cuya clave sea de la clase Fecha:

  ```
  data class Fecha(val dia: Int, val mes: Int, val año: Int)
  ```

  Como valor en el mapa almacenar un String.
  Implementar las siguientes funciones:
  1) Carga de datos en la agenda.

2) Listado completo de la agenda.
3) Consulta de una fecha.

Solución

Retornar (index.php?inicio=45)

47 - Colecciones - Set y MutableSet

La clase Set y MutableSet (conjunto) permiten almacenar un conjunto de elementos que deben ser todos distintos. No permite almacenar datos repetidos.

Un conjunto es una colección de elementos sin un orden específico a diferencia de las listas.

Problema 1

Crear un conjunto mutable (MutableSet) con una serie de valores Int. Probar las propiedades y métodos principales para administrar el conjunto.

Proyecto187 - Principal.kt

```
fun main(args: Array<String>) {
    val conjunto1: MutableSet<Int> = mutableSetOf(2, 7, 20, 150, 3)
    println("Listado completo del conjunto")
    for(elemento in conjunto1)
        print("$elemento ")
    println()
    println("Cantidad de elementos del conjunto: ${conjunto1.size}")
    conjunto1.add(500)
    println("Listado completo del conjunto luego de agregar el 500")
    for(elemento in conjunto1)
        print("$elemento ")
    println()
    conjunto1.add(500)
    println("Listado completo del conjunto luego de volver a agregar el 500")
    for(elemento in conjunto1)
        print("$elemento ")
    println()
    if (500 in conjunto1)
        println("El 500 está almacenado en el conjunto")
    else

        println("El 500 no está almacenado en el conjunto")
    println("Eliminamos el elemento 500 del conjunto")
    conjunto1.remove(500)
    if (500 in conjunto1)
        println("El 500 está almacenado en el conjunto")
    else
        println("El 500 no está almacenado en el conjunto")
    val cant = conjunto1.count { it >= 10 }
    println("Cantidad de elementos con valores superiores o igual a 10: $cant")
}
```

Creamos un MutableSet con elementos de tipo Int y almacenamos 5 enteros:

```
val conjunto1: MutableSet<Int> = mutableSetOf(2, 7, 20, 150, 3)
```

Podemos recorrer con un for todos los elementos de un conjunto igual que las otras colecciones que proporciona Kotlin:

```
for(elemento in conjunto1)
    print("$elemento ")
```

Para añadir un nuevo elemento a un conjunto llamamos al método add (esto se puede hacer solo con la clase MutableSet y no con Set):

```
conjunto1.add(500)
```

Si el elemento que enviamos al método add ya existe luego no se inserta (esto debido a que las colecciones de tipo conjunto no pueden tener valores repetidos)

Para verificar si un elemento se encuentra contenido en el conjunto podemos hacerlo mediante el operador in:

```
if (500 in conjunto1)
    println("El 500 está almacenado en el conjunto")
else
    println("El 500 no está almacenado en el conjunto")
```

Para eliminar un elemento del conjunto podemos llamar al método remove y pasar el valor a eliminar:

```
conjunto1.remove(500)
```

Si queremos conocer todas las propiedades y métodos de MutableSet podemos visitar la documentación de la biblioteca estándar (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-set/index.html) de Kotlin.

Lo mismo para Set podemos visitar la documentación de la biblioteca estándar (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-set/index.html) de Kotlin.

Problema 1

Crear un conjunto inmutable que almacene las fechas de este año que son feriados. Ingresar luego por teclado una fecha y verificar si se encuentra en el conjunto de feriados

Proyecto188 - Principal.kt

```
data class Fecha(val dia: Int, val mes: Int, val año: Int)

fun main(args: Array<String>) {
    var feriados: Set<Fecha> = setOf(Fecha(1, 1, 2017),
                                     Fecha(25, 12, 2017))
    println("Ingrese una fecha")
    print("Ingrese el día:")
    val dia = readLine()!!.toInt()
    print("Ingrese el mes:")
    val mes = readLine()!!.toInt()
    print("Ingrese el año:")
    val año = readLine()!!.toInt()
    if (Fecha(dia, mes, año) in feriados)
        println("La fecha ingresada es feriado")
    else
        println("La fecha ingresada no es feriado")

}
```

Declaramos un data class que representa una fecha:

```
data class Fecha(val dia: Int, val mes: Int, val año: Int)
```

Definimos un conjunto inmutable de tipo Fecha y guardamos dos fechas mediante la llamada a la función setOf:

```
    var feriados: Set<Fecha> = setOf(Fecha(1, 1, 2017),
                                     Fecha(25, 12, 2017))
```

Cargamos una fecha cualquiera por teclado:

```
    println("Ingrese una fecha")
    print("Ingrese el día:")
    val dia = readLine()!!.toInt()
    print("Ingrese el mes:")
    val mes = readLine()!!.toInt()
    print("Ingrese el año:")
    val año = readLine()!!.toInt()
```

Mediante el operador in verificamos si la fecha ingresada se encuentra en el conjunto de feriados:

```
    if (Fecha(dia, mes, año) in feriados)
        println("La fecha ingresada es feriado")
    else
        println("La fecha ingresada no es feriado")
```

Problema propuesto

- Definir un MutableSet y almacenar 6 valores aleatorios comprendidos entre 1 y 50. El objeto de tipo MutableSet representa un cartón de lotería.
  Comenzar a generar valores aleatorios (comprendidos entre 1 y 5) todos distintos y detenerse cuando salgan todos los que contiene el cartón de lotería. Mostrar cuantos números tuvieron que sortearse hasta que se completó el cartón.

Solución

Retornar (index.php?inicio=45)

48 - Package e Import

Los paquetes nos permiten agrupar clases, funciones, objetos, constantes etc. en un espacio de nombres para facilitar su uso y mantenimiento.

Los paquetes agrupan funcionalidades sobre un tema específico, por ejemplo funcionalidades para el acceso a base de datos, interfaces visuales, encriptación de datos, acceso a archivos, comunicaciones en la web etc.

Los paquetes son una forma muy efectiva de organizar nuestro código para ser utilizado en nuestro proyecto o reutilizado por otros.

La librería estándar de Kotlin se organiza en paquetes, los principales son:

```
kotlin
kotlin.annotation
kotlin.collections
kotlin.comparisons
kotlin.io
kotlin.ranges
kotlin.sequences
kotlin.text
```

Todos estos paquetes se importan en forma automática cuando compilamos nuestra aplicación.

Por ejemplo el paquete kotlin.collections tiene las declaraciones de clases e interfaces: List, MutableList etc.

Veremos ahora como crear nuestro propio paquete y luego consumir su contenido.

Problema 1

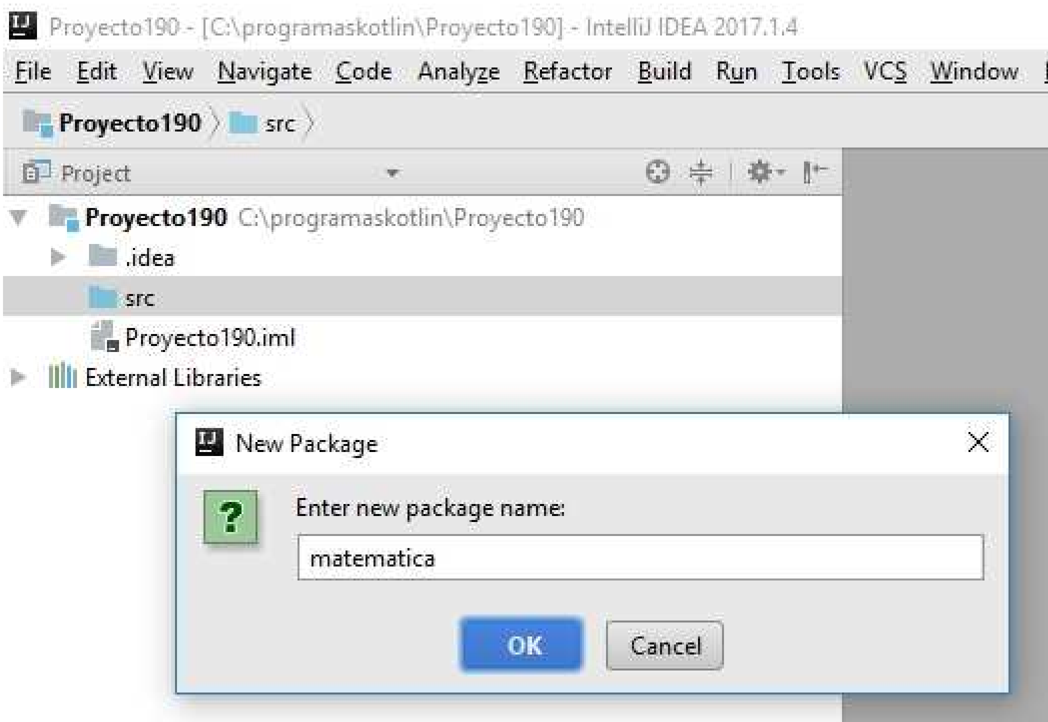
Crear un paquete llamado matematica y definir dos funciones en su interior que permitan sumar y restar dos valores de tipo Int.

Importar luego el paquete desde nuestro programa principal y llamar a dichas funciones.

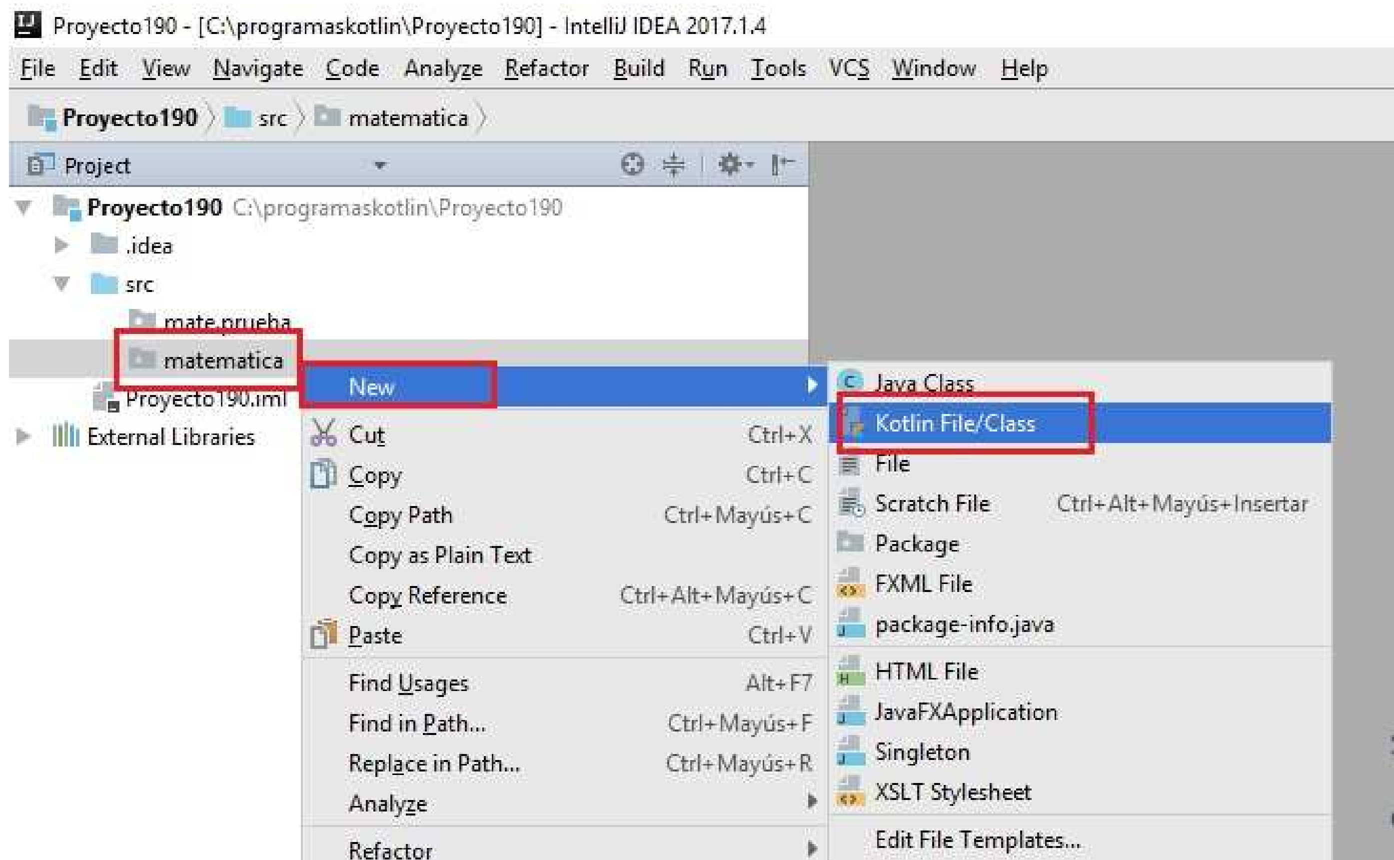
El primer paso luego de crear el Proyecto190 nos posicionamos en la carpeta src, presionamos el botón derecho del mouse y seleccionamos New -> Package:

Proyecto190 - [C:\programaskotlin\Proyecto190] - IntelliJ IDEA 2017.1.4
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Proyecto190 〉 src 〉
Project
Proyecto190 C:\programaskotlin\Proyecto190
.idea
src
Proy
External
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Mayús+C
Copy as Plain Text
Copy Reference Ctrl+Alt+Mayús+C
Paste Ctrl+V
Find Usages Alt+F7
Find in Path... Ctrl+Mayús+F
Replace in Path... Ctrl+Mayús+R
Analyze
Refactor
Add to Favorites
Show Image Thumbnails Ctrl+Mayús+T
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Suprimir
Build Module 'Proyecto190'
Rebuild '<default>' Ctrl+Mayús+F9
Java Class
Kotlin File/Class
File
Scratch File Ctrl+Alt+Mayús+Insertar
Package
FXML File
package-info.java
HTML File
JavaFXApplication
Singleton
XSLT Stylesheet
Edit File Templates...
GUI Form
Dialog
Form Snapshot
Resource Bundle

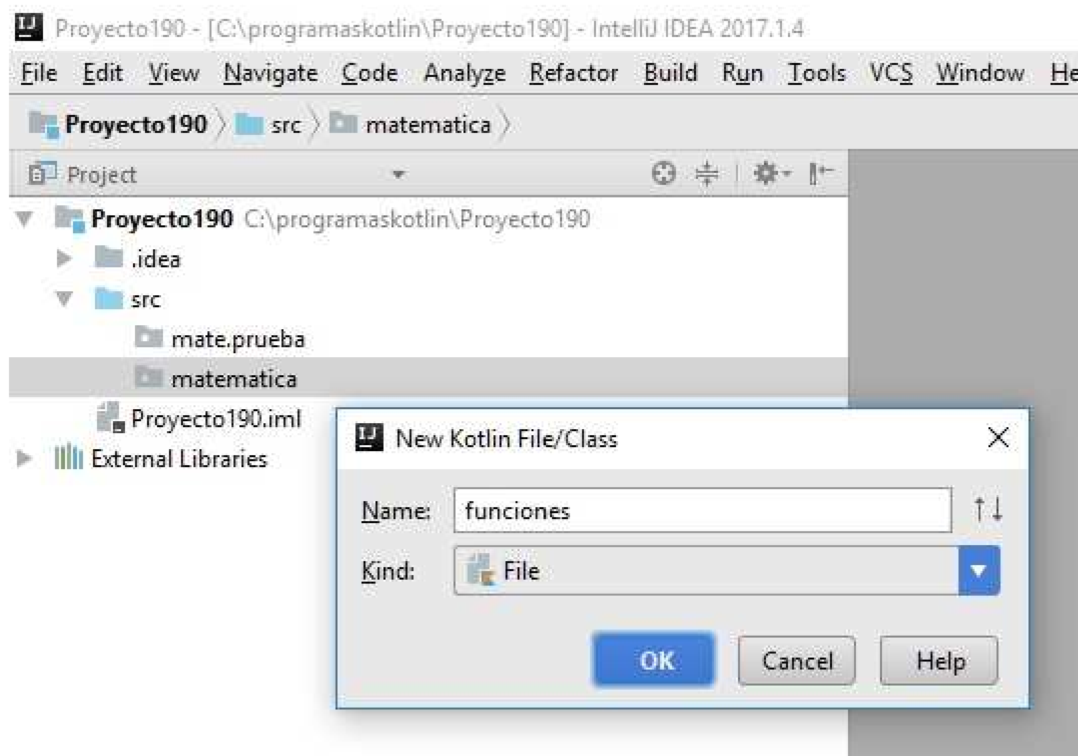
En el diálogo que aparece ingresamos el nombre del paquete a crear "matematica":



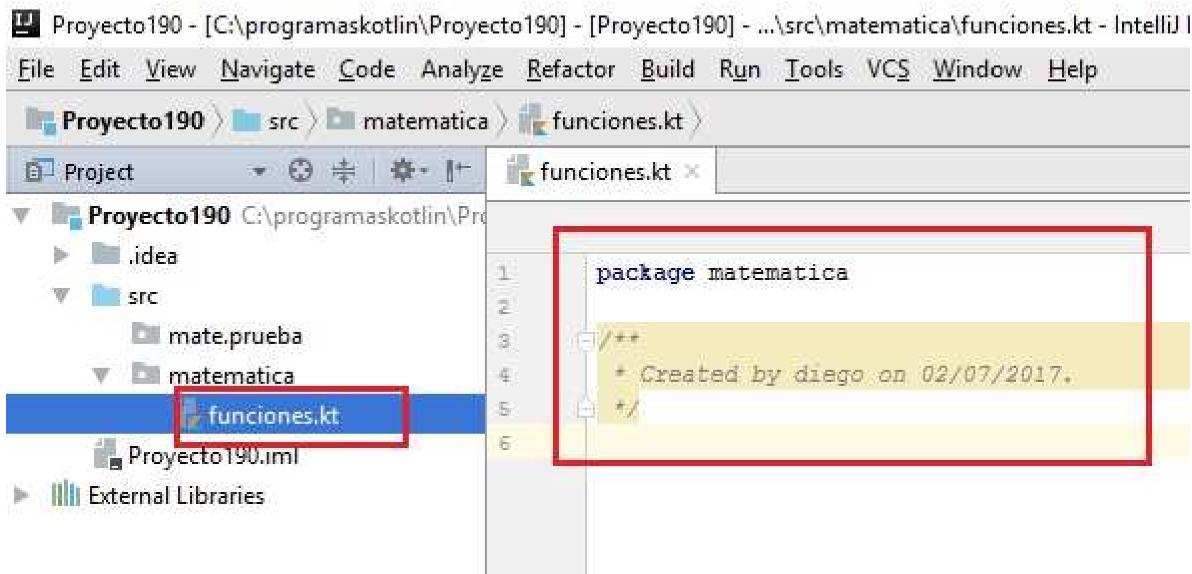
Ahora dentro de esta carpeta procedemos a crear un archivo llamado "funciones" presionando el botón derecho del mouse sobre la carpeta "matematica" que acabamos de crear:



El archivo que crearemos en este paquete se llamará "funciones" (un paquete puede tener muchos archivos):



Ahora ya tenemos el archivo donde codificaremos las funciones de sumar y restar:



Procedemos:

Proyecto190 - funciones.kt

```
package matematica

fun sumar(valor1: Int, valor2: Int) = valor1 + valor2

fun restar(valor1: Int, valor2: Int) = valor1 - valor2
```

El nombre del paquete va en la primera línea del código fuente después de la palabra clave package.

Ahora codificaremos nuestro programa principal donde consumiremos la funcionalidad de este paquete. Crearemos el archivo Principal.tk en la carpeta src como siempre:

Proyecto190 - Principal.kt

```
import matematica.*

fun main(args: Array<String>) {
    val su = sumar(5, 7)
    println("La suma es $su")
    val re = restar(10, 3)
    println("La resta es $re")
}
```

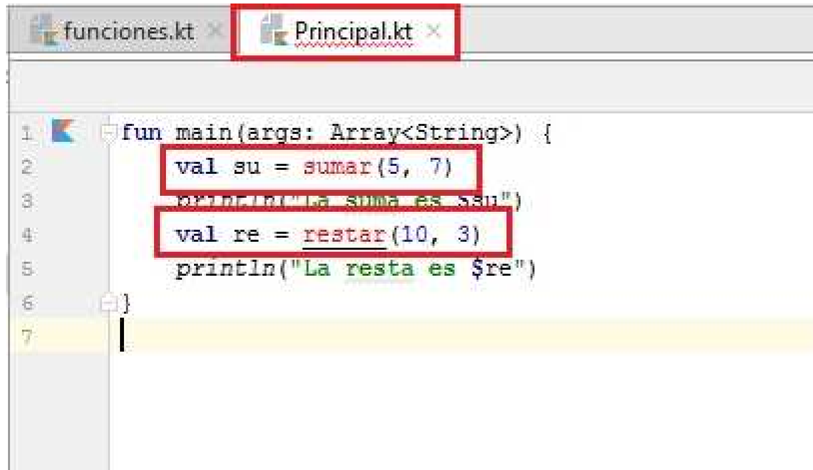
Para importar todas las funciones definidas en el paquete matematica utilizamos la palabra clave import seguida del nombre del paquete y luego un punto y el asterisco:

```
import matematica.*
```

Una vez importado el paquete completo podemos llamar a sus funciones:

```
    val su = sumar(5, 7)
    ...
    val re = restar(10, 3)
```

Si no hacemos el import se presenta un error sintáctico cuando queremos compilar el programa:



Hay varias variantes cuando hacemos uso del import. La primera es que podemos importar por ejemplo solo la función de restar del paquete matematica:

```
import matematica.restar

fun main(args: Array<String>) {
    val re = restar(10, 3)
    println("La resta es $re")
}
```

Como vemos indicamos luego del nombre del paquete la función a importar. Solo podremos llamar en nuestro algoritmo a la función restar.

Cuando importamos una funcionalidad podemos crear un alias, en el ejemplo siguiente importamos la función restar y la renombramos con el nombre restaEnteros:

```
import matematica.restar as restaEnteros

fun main(args: Array<String>) {
    val re = restaEnteros(10, 3)
    println("La resta es $re")
}
```

Esto es útil cuando importamos por ejemplo funciones de dos paquetes que tienen el mismo nombre.

Si no disponemos el import la otra posibilidad es indicar el paquete previo al nombre de la función (esto es muy engorroso si tenemos que acceder a muchas funcionalidades de un paquete):

```
fun main(args: Array<String>) {
    val su = matematica.sumar(5, 7)
    println("La suma es $su")
    val re = matematica.restar(10, 3)
    println("La resta es $re")
}
```

Problema propuesto

- Crear un paquete llamado entradateclado. Dentro del paquete crear un archivo llamado entrada.kt y definir las siguientes funciones:

  ```
  package entradateclado

  fun retornarInt(mensaje: String): Int {
      print(mensaje)
      return readLine()!!.toInt()
  }

  fun retornarDouble(mensaje: String): Double {
      print(mensaje)
      return readLine()!!.toDouble()
  }

  fun retornarFloat(mensaje: String): Float {
      print(mensaje)
      return readLine()!!.toFloat()
  }
  ```

  En el programa principal (Principal.tk) importar el paquete entradateclado y llamar a varias de sus funciones.

Solución

Retornar (index.php?inicio=45)