

```
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 clase padre.

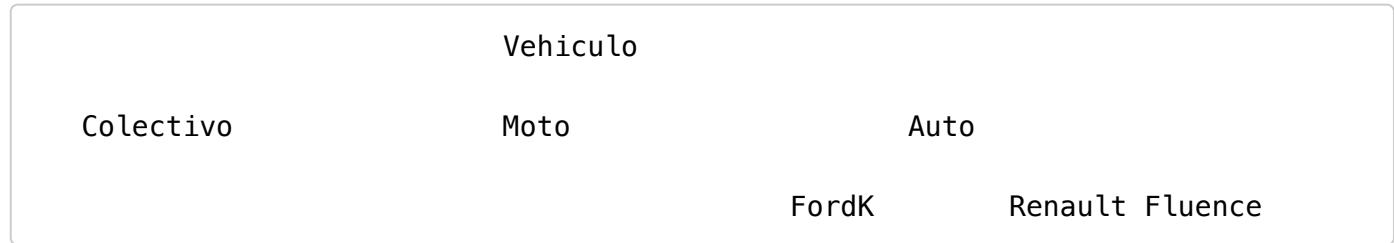
Subclase

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.

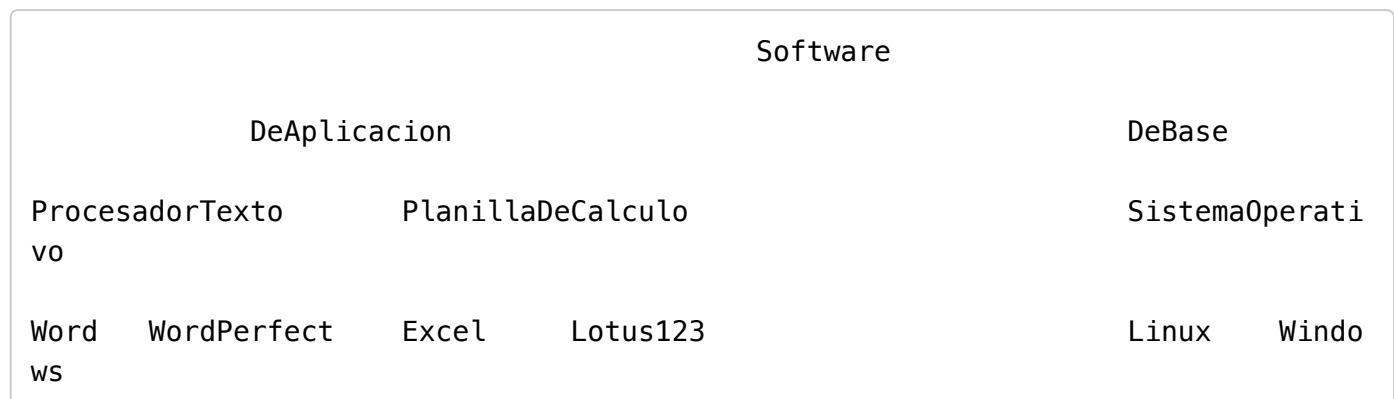
Veamos algunos ejemplos teóricos de herencia:

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 relación 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, edad) {
    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 personal = Persona("Jose", 22)
    println("Datos de la persona")
    personal.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 Científica (suma de dos números y el cuadrado y la raíz 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 CalculadoraCientífica y llamando a algunos de sus métodos:

```

println("Prueba de la clase Calculadora Científica (suma de dos números y
el cuadrado y la raíz 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 asteriscos. 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 sumal = Suma(10, 4)
    sumal.operar()
    sumal.imprimir()
    val restal = Resta(20, 5)
    restal.operar()
    restal.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("Perímetro: ${fig.calcularPerímetro()}")  
    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 componentes 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 subíndice
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 elementos de tipo Articulo.

Implementar dos funciones, una que le envíemos 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  $\rightarrow$  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 mayorEstados Unidos(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 personal1 = Persona("juan", 18)
    if (personal1.esMayor(::mayorArgentina))
        println("${personal1.nombre} es mayor si vive en Argentina")
    else
        println("${personal1.nombre} no es mayor si vive en Argentina")
    if (personal1.esMayor(::mayorEstados Unidos))
        println("${personal1.nombre} es mayor si vive en Estados Unidos")
    else
        println("${personal1.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 mayorEstados Unidos(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 "mayorEstados Unidos":

```
if (persona1.esMayor(::mayorEstados Unidos))  
    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 mayorEstados Unidos(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(::mayorEstados Unidos))
        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 elevarCuarto = operar(2, 4, {x, y ->
        var valor = 1
        for(i in 1..y)
            valor = valor * x
        valor
    })
    println(elevarCuarto)
}
```

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 primera expresión lambda podemos identificarla en la primera 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  $\rightarrow$  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  $\rightarrow$

```
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 elevarCuarto = 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 elevarCuarto = operar(2, 4) {x, y ->
        var valor = 1
        for(i in 1..y)
            valor = valor * x
        valor
    }
    println(elevarCuarto)
}
```

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  $\geq 50$ :

```
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 carácter 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 carácter 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 función 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 sumal = Vector()
        for(i in arreglo.indices)
            sumal.arreglo[i] = arreglo[i] + 1
        return sumal
    }

    operator fun dec(): Vector {
        var restal = Vector()
        for(i in arreglo.indices)
            restal.arreglo[i] = arreglo[i] - 1
        return restal
    }
}

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 personal1 = Persona("Juan", 30)
    personal1.imprimir()
    val persona2 = Persona("Ana", 30)
    persona2.imprimir()
    println("Persona mayor")
    when {
        personal1 > persona2 -> personal1.imprimir()
        personal1 < 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 Tateti a medida que le asignamos piezas:

The screenshot shows the IntelliJ IDEA interface with the following details:

- Title Bar:** Projeto166 - [C:\programaskotlin\Proyecto166] - [Proyecto166] - ...src\Principal.kt - IntelliJ IDEA 2017.1.4
- Menu Bar:** File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
- Toolbar:** Standard IntelliJ toolbar with icons for file operations.
- Project View:** Shows the project structure with 'Proyecto166' containing '.idea', 'out', 'src', and 'Proyecto166.iml' files, and 'External Libraries'.
- Code Editor:** Displays 'Principal.kt' with the following code:

```
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  
}
```
- Run Tab:** Shows the run configuration for 'PrincipalKt' with the command 'C:\Program Files (x86)\Java\jdk1.7.0\bin\java' ...
- Output:** The run tab displays the output of the program, which is a 3x3 matrix:

```
1 0 0  
0 0 0  
0 0 0  
  
1 0 2  
0 0 0  
0 0 0  
  
1 0 2  
0 0 0  
1 0 0  
  
1 0 2  
1 0 2  
1 0 0
```

## 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 <sub>1</sub> , ..., i <sub>n</sub> )	a.invoke(i <sub>1</sub> , ..., i <sub>n</sub> )

## Problema 6

Plantear una clase Dados que administre 10 valores de datos 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 curso")
    else
        println("El alumno Marcos no está inscripto en el curso")
}
```

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"), Alumno(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.documento}
```

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:

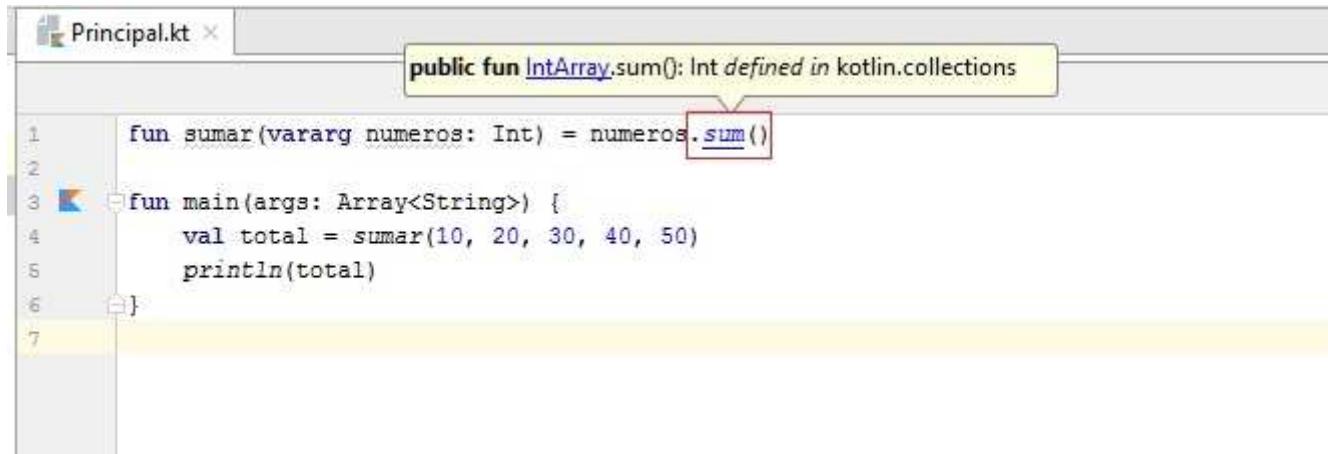
```
val total = sumar(10, 20, 30, 40, 50)
```

Recordemos que Kotlin tiene como principio permitir implementar algoritmos muy concisos, la función sumar podemos codificarla en forma más concisa con la sintaxis:

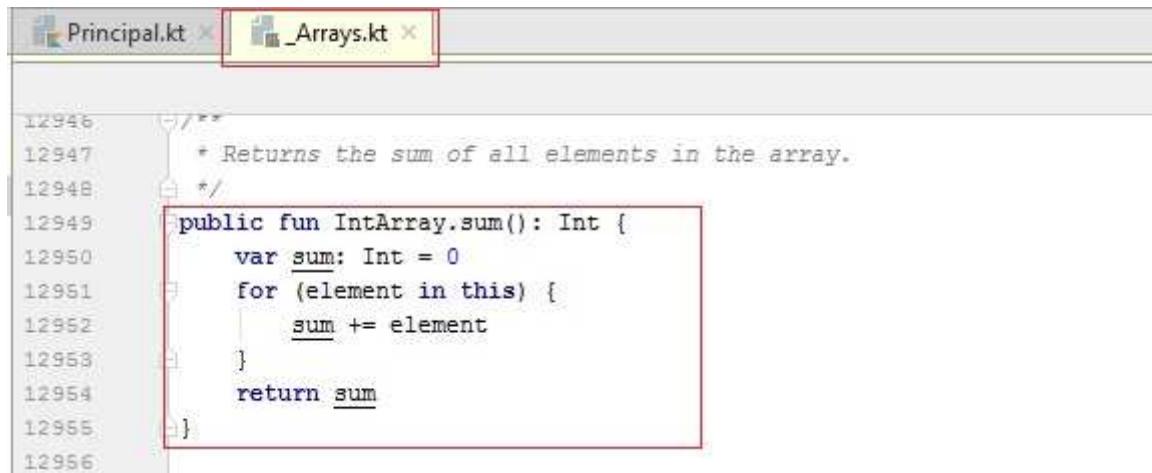
```
fun sumar(vararg numeros: Int) = numeros.sum()
```

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:



```
12946     /**
12947      * Returns the sum of all elements in the array.
12948     */
12949     public fun IntArray.sum(): Int {
12950         var sum: Int = 0
12951         for (element in this) {
12952             sum += element
12953         }
12954         return sum
12955     }
12956 }
```

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 vararg 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 carácter \* 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

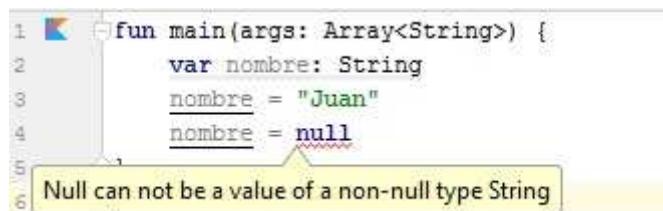
Retornar (index.php?inicio=30)

# 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:



A screenshot of the IntelliJ IDEA code editor. The code is as follows:

```
1 fun main(args: Array<String>) {
2     var nombre: String
3     nombre = "Juan"
4     nombre = null
}
```

The line `nombre = null` is underlined with a red squiggly line, indicating an error. A tooltip window appears below the line with the text `Null can not be a value of a non-null type String`.

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 carácter "?" 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: DatosPersonales?)

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?.telefono}")
}
```

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: DatosPersonales?)
```

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.datosPersonales?.telefono}")
```

Analizando si tienen almacenado valores distintos a null hasta llegar a la propiedad del teléfono 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?.telefono}")
```

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 crear 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).toInt() }
    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("Marcos", 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.  
Aregar 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 país.

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 completa:

```

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 recuperaremos 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 existe la clave buscada en el mapa definimos la variable cantHabitantes1 de Int? ya que puede almacenar un null si no existe el país 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 países 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 completa 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 inglés/castellano. La clave es la palabra en inglés y el valor es la palabra en castellano.

Crear las siguientes funciones:

- 1) Cargar el diccionario (se ingresan por teclado la palabra en inglés y su traducción en castellano)
- 2) Listado completo del diccionario.
- 3) Ingresar por teclado una palabra en inglés 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 inglés:")
        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 inglés 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 inglés 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 inglés:")
        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: ${producto.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[codigo]?.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 siguientes 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 siguientes 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.

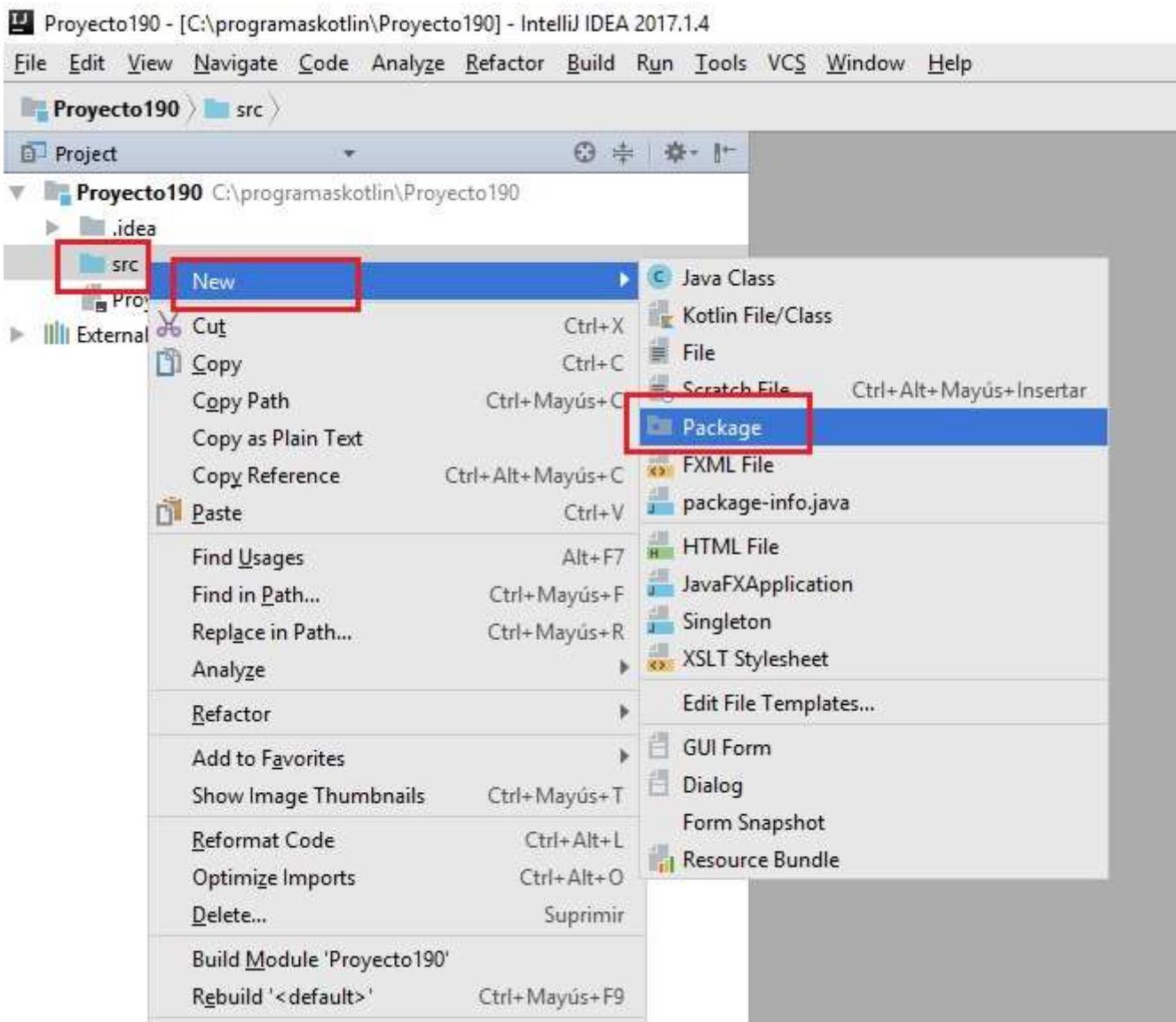
Veremos ahora como crear nuestro propio paquete y luego consumir su contenido.

## Problema 1

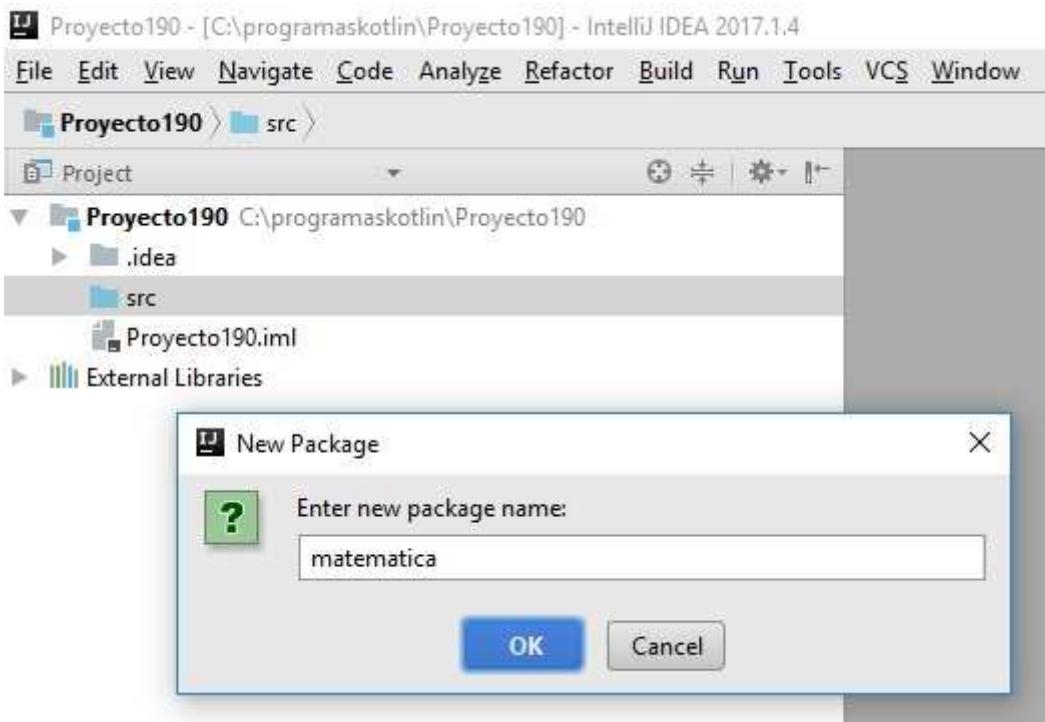
Crear un paquete llamado matematica y definir dos funciones en su interior que permitan sumar y restar dos valores de tipo Int.

Importar luego el paquete desde nuestro programa principal y llamar a dichas funciones.

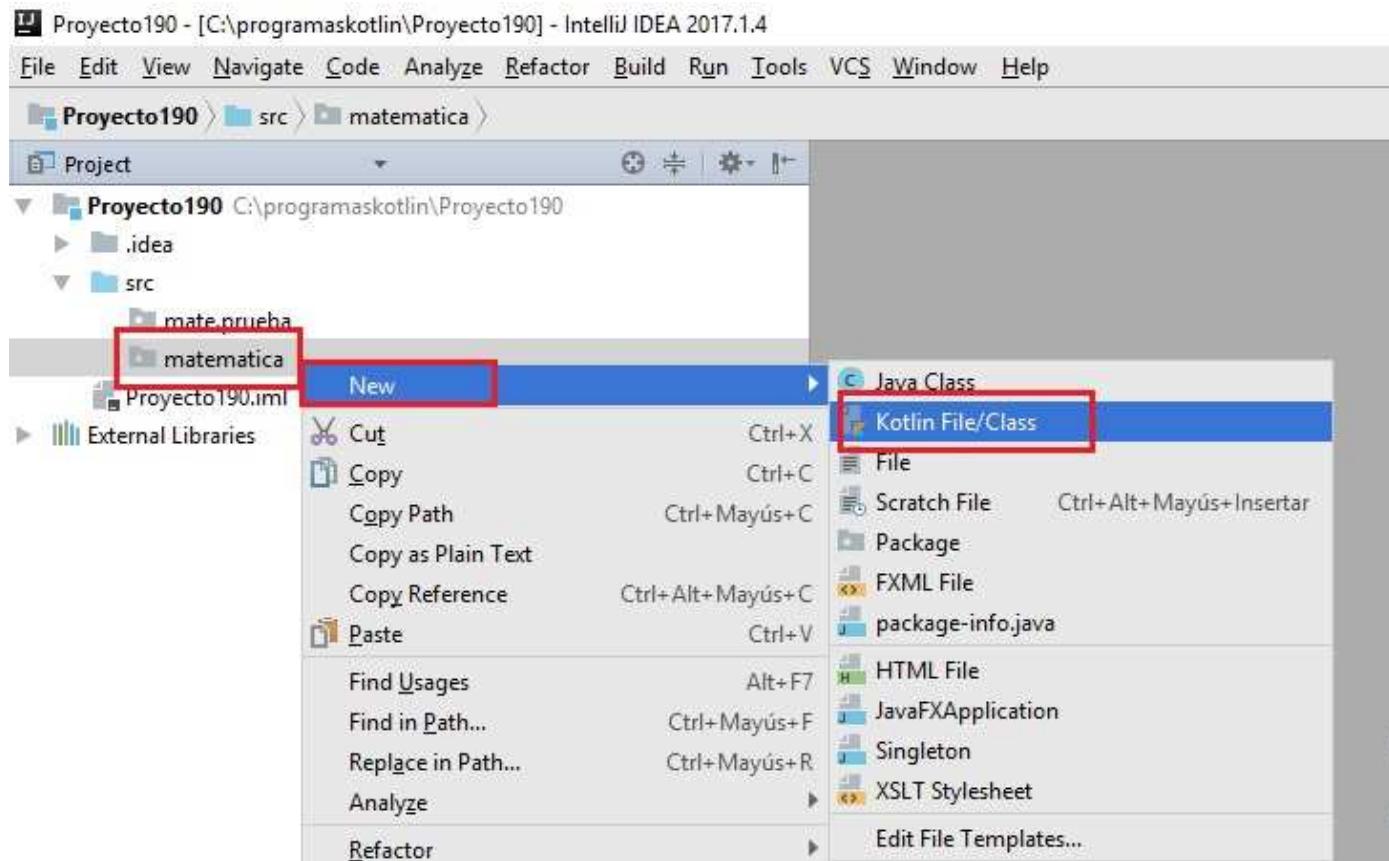
El primer paso luego de crear el Proyecto190 nos posicionamos en la carpeta src, presionamos el botón derecho del mouse y seleccionamos New -> Package:



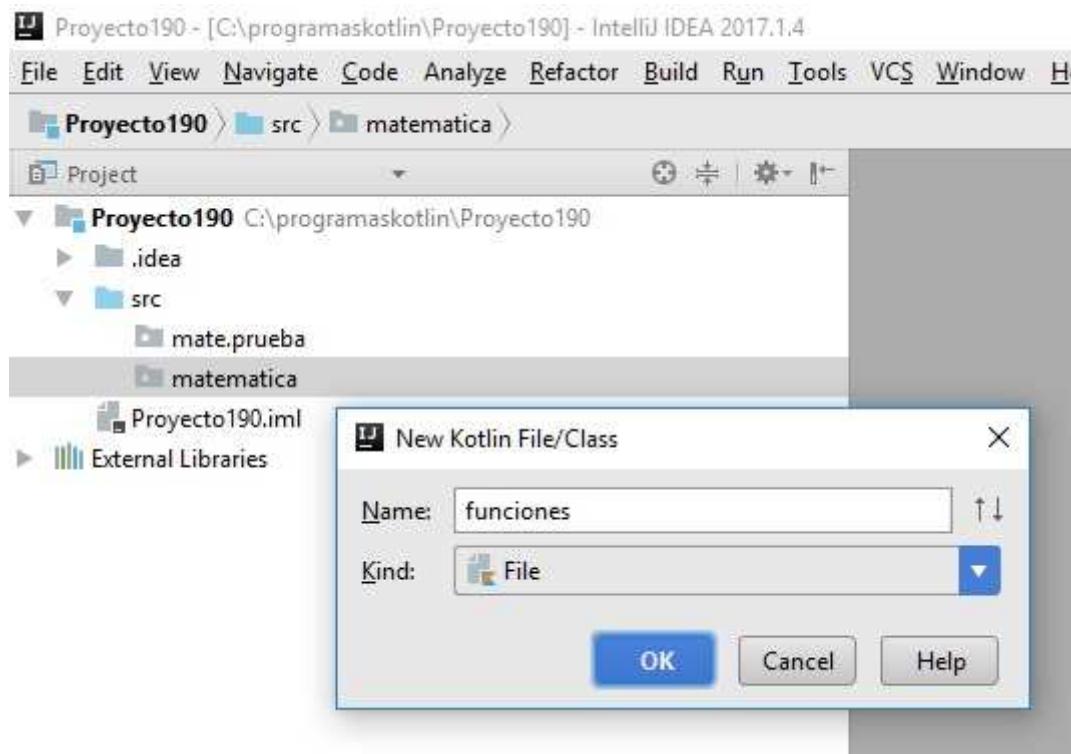
En el diálogo que aparece ingresamos el nombre del paquete a crear "matematica":



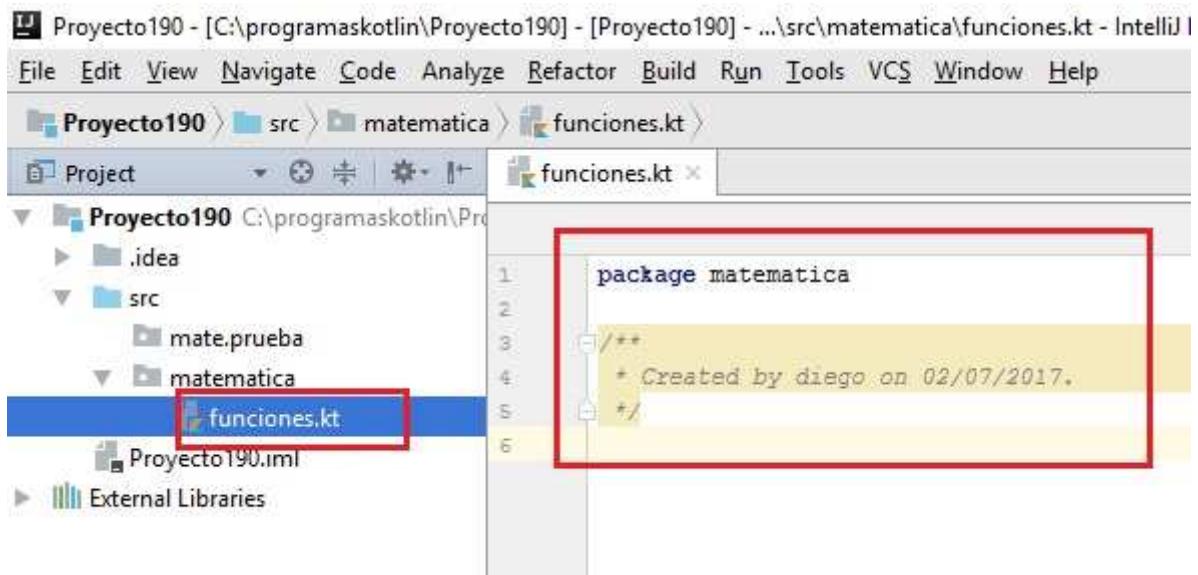
Ahora dentro de esta carpeta procedemos a crear un archivo llamado "funciones" presionando el botón derecho del mouse sobre la carpeta "matematica" que acabamos de crear:



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.kt 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")
}
```

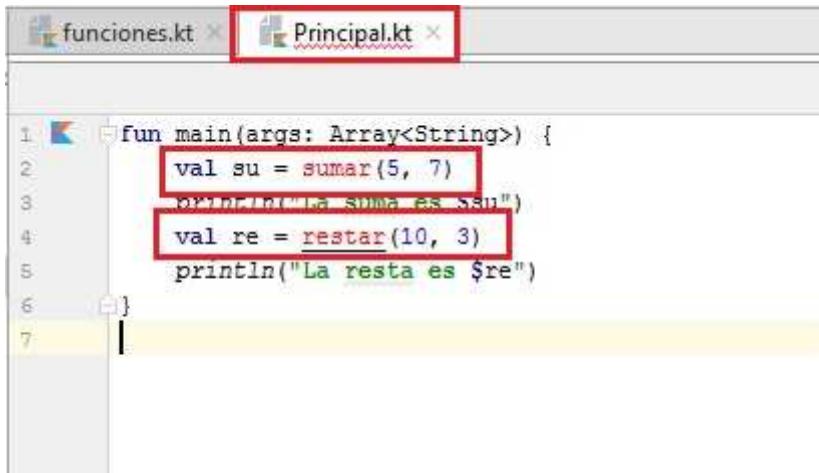
Para importar todas las funciones definidas en el paquete matematica utilizamos la palabra clave import seguida del nombre del paquete y luego un punto y el asterisco:

```
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:



```
funciones.kt ✘ Principal.kt ✘

1 fun main(args: Array<String>) {
2     val su = sumar(5, 7)
3     println("La suma es $su")
4     val re = restar(10, 3)
5     println("La resta es $re")
6 }
7
```

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)**