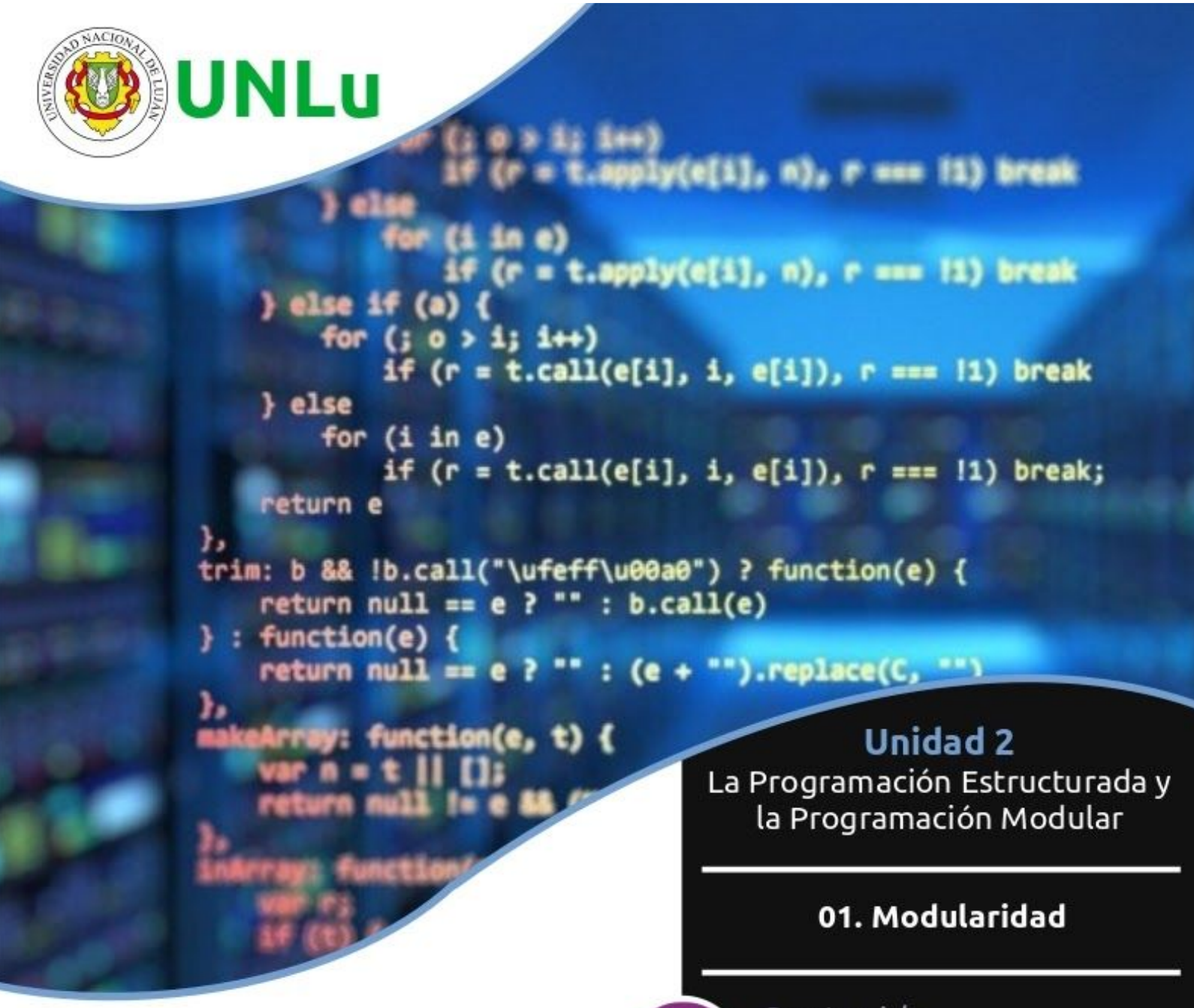
UNLu



Unidad 2

La Programación Estructurada y la Programación Modular

01. Modularidad

11074 – Programación I

Material didáctico preparado para el dictado de la actividad académica.

Contenido

Hablamos aquí sobre los conceptos básicos de la modularidad y sus propiedades

Actividades

Te ofrecemos actividades a desarrollar para fijar mejor los conceptos

Participación

Te proponemos participar en foros de discusión y compartir tu aprendizaje

Universidad Nacional de Luján
Departamento de Ciencias Básicas
División Computación
www.unlu.edu.ar

Introducción

En algunas oportunidades ya hicimos mención a la Programación Estructurada. En esta guía comenzamos a formalizar este tema porque, a partir de aquí dejamos atrás los temas de Introducción a la Programación y comenzamos a hablar específicamente de Programación I.

Cuando hablamos de Programación Estructurada nos estamos refiriendo específicamente a un conjunto de técnicas que, con el transcurrir del tiempo han progresado, pero no debemos olvidar que su aparición, fue un antes y un después para la programación. Este paradigma permitió mejorar la productividad en el desarrollo de programas, haciéndolos más comprensibles y modificables. El aporte que realiza es limitar las estructuras de control con lo que se reduce la complejidad de los problemas y se minimizan los errores.

Teorema de la Programación Estructurada

Extractamos aquí la parte fundamental del teorema de la Programación Estructurada:

Un programa propio puede ser escrito utilizando ***sólo tres tipos de estructuras de control:***

- *Secuenciales*
- *Selectivas*
- *Repetitivas*

Un programa es ***propio*** *si cumple con las siguientes características:*

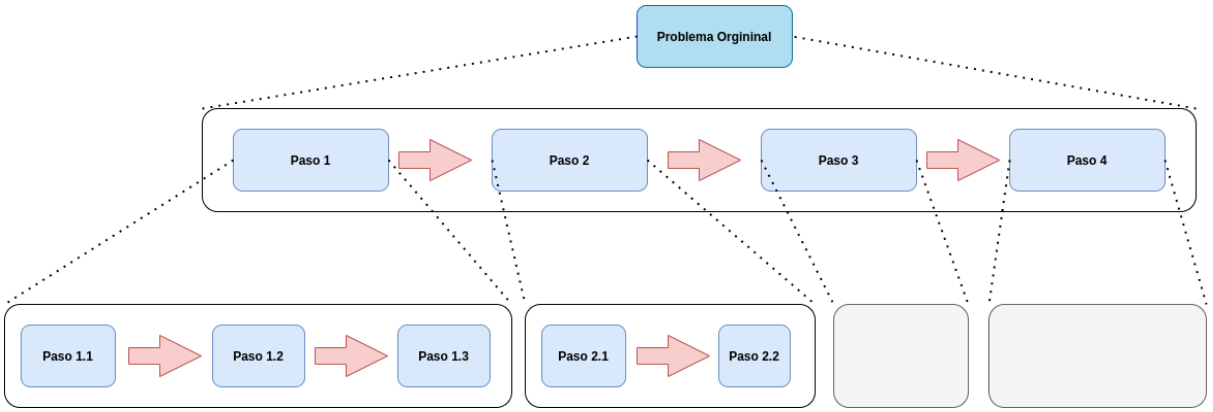
1. *Tiene un solo punto de entrada y uno de salida o fin de control del programa.*
2. *Existen caminos desde la entrada hasta la salida que se pueden seguir y que pasan por todas partes del programa.*
3. *Todas las instrucciones son ejecutables y no existen lazos o bucles infinitos (sin fin).*

Esta programación incorpora, además, un conjunto de técnicas muy valiosas como el procesos de abstracción que permite representar un problema del mundo real a través de su representación y el diseño descendente, también denominado top-down.

En esta figura se ve un proceso de abstracción. Fijate como la fotografía de la oveja se va transformando en cada vez más abstracta, quitando detalles pero como siempre seguís pensando que es una oveja.

En los procesos descendentes el problema se descompone en una serie de pasos sucesivos (stepwise). Los pasos pueden tener diferente nivel de abstracción. Si la abstracción conseguida no posee el nivel necesario para poder ser representado por un conjunto de instrucciones ejecutables, se lo vuelve a descomponer y así sucesivamente hasta lograr que todos los procesos de abstracción

puedan ser representados en forma de conjunto de instrucciones o estructuras de control (¿te acordas? : secuencial, selectiva o repetitiva).



Esta descomposición genera secuencias útiles que realizan tareas con un propósito definido. Estas tareas pueden, de alguna manera, encapsularse en lo que llamamos módulo. ¿Por qué conviene realizar este encapsulamiento? Cuando contamos con un módulo (conjunto de código encerrado o encapsulado), podemos representarlo con un nombre genérico y convocarlo cada vez que haga falta. Te lo explicamos con una receta:

Vamos a representar todo este proceso con un ejemplo simple y corto para no aburrir, vos podés extenderlo con ejemplos más complicados.

Tomamos para eso un extracto de una receta de cocina:

Calentar a baño María hasta que el flan esté cocido.

Vamos a explicar para los no culinarios que es esto (te encargamos una porción de flan cuando lo practiques ;))

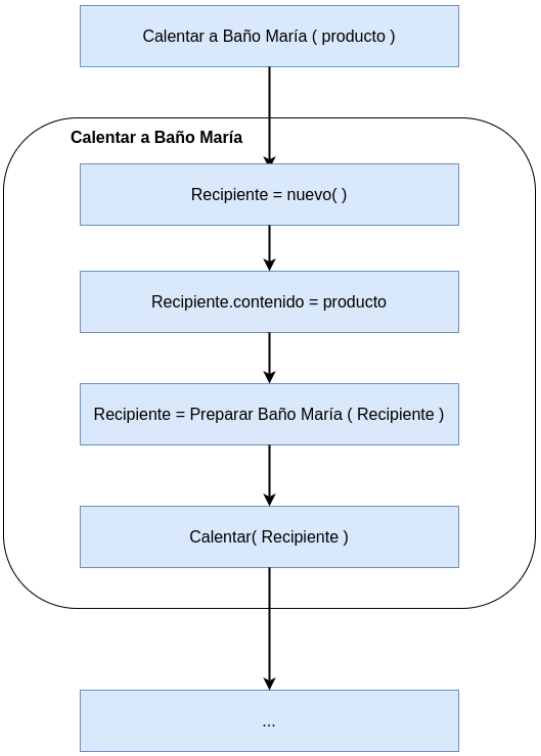
- ¿Qué significa Calentar a "baño María" o "baño de María"?
- "Baño María" significa que tenés que poner el recipiente del producto a calentar dentro de otro recipiente más grande y lleno de agua. Este último es el que recibe el fuego directo y el de adentro se va calentando por el calor del agua de manera que no supera nunca los 100 grados centígrados.
- Ahora: ¿Qué significa calentar?
- Calentar significa encender un fuego y poner el elemento sobre él para que eleve su temperatura..
- ¿Podemos representar eso con un algoritmo?
- Bueno, podríamos decir que : recibido un recipiente, se abre la llave de gas de la cocina, se le acerca un fósforo o elemento que haga chispa hasta que la llama se encienda. Si pasado cinco segundos de intentar no se encendió, cerrar la llave de gas, esperar cinco segundos y reintentar.

Este último algoritmo podríamos representarlo con un módulo llamado "cocinar" que recibe un recipiente.

"Preparar baño María" podría ser otro módulo que recibe un recipiente y genera un recipiente de mayor volumen con agua dentro y el recipiente metido en él.

Entonces, “Calentar a baño María” dado un recipiente sería: Pasar este recipiente al módulo “Preparar baño María” para recibir otro recipiente que se lo pasamos a “Calentar”.

Nuestro esquema sería:



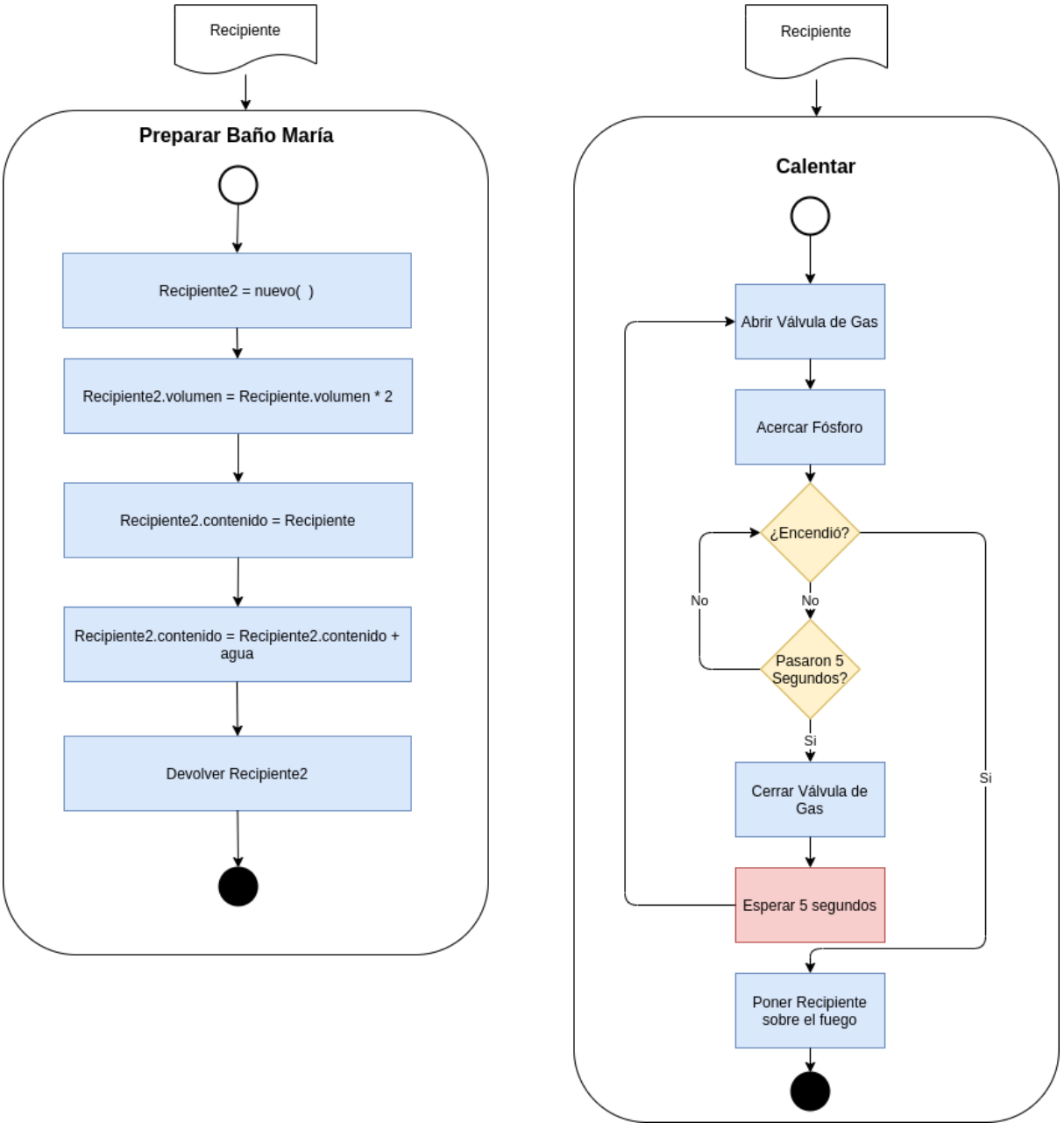
En este esquema, “Preparar Baño María” y “Calentar” son dos módulos que se utilizan en el módulo “Calentar a Baño María”. ¿Lo ves?

Ahora mirá los esquemas de “Preparar Baño María” y el de “Calentar”.

Si bien en estos esquemas no se utiliza ninguna metodología de representación estándar, creemos que se entiende perfectamente que cada módulo puede utilizarse en infinidad de otros módulos y son independientes, lo que le aporta una ventaja cuantiosa a la efectividad de código por tres cuestiones muy simples:

- El código se escribe una sola vez y por lo tanto disminuye el riesgo de cometer errores de tipeo y aumenta la mantenibilidad, ya que un cambio en el módulo se implementa en TODOS los que lo usan.
- Aumenta la velocidad de escritura. Con una sola instrucción “ordeno” que se hagan en verdad muchas cosas.
- Permite reemplazar un módulo por otro y que las cosas se adapten a nuevos casos. Esto es más difícil de que te des cuenta pero pensalo y vas a ver que es genial! Te damos un ejemplo. Suponte que querés que en lugar de cocinar en una cocina, hacerlo en el horno. Solo hay que armar un módulo “Calentar En Horno” y cambiar “Calentar” por éste en el último paso de “Cocinar a Baño María” . ¿Te diste cuenta? No cambiás el módulo “Calentar” porque éste puede servirte para usar con una cocina sino que creas uno nuevo que use el horno y lo usás en lugar del de la cocina y todo funciona.

La última ventaja que mencionamos se potencia en otros paradigmas a través de lo que se denomina polimorfismo. Pero eso es "harina de otro costal". Nos ocuparemos de eso en Programación Orientada a Objetos.



De esta manera se entiende que la modularidad consiste en dividir un programa en partes, los cuales pueden trabajarse por separado.

En términos de programación, los módulos pueden ser programas o librerías separadas del programa principal. De esta manera se logra su objetivo que es reducir el costo de elaboración de programas al poder dividir el trabajo entre varios programadores, por ejemplo y reutilizar el código.

Procedimientos y Funciones

Una de las formas más habituales que toma la modularidad y la primera que vamos a usar nosotros, es permitirnos dividir nuestro código en pequeños módulos llamados procedimientos o funciones. Estas palabras no son sinónimos aunque hoy por hoy en varios lenguajes de programación las cosas se confunden un poco.

Un Procedimiento es un módulo que se invoca con un nombre y que puede o no enviar un conjunto de parámetros pero que no devuelve nada como resultado.

Una Función en cambio, es igual a un procedimiento pero devuelve un resultado como respuesta a su invocación.

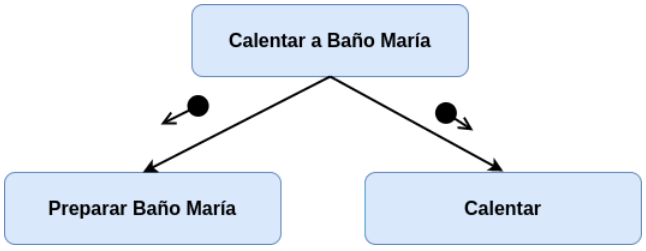
Originalmente, y así lo vamos a tratar en esta asignatura, una función devuelve uno y solo un valor. Nunca puede devolver más de uno. Con este concepto, cuando necesitamos más de un valor de retorno es porque, probablemente, tenemos la abstracción equivocada y habría que descomponer el problema en más de uno.

Para tener cuidado!!!

Hoy en día, lenguajes como **Go** y **Python** permiten que una función devuelva más de un valor. Insistimos en que nosotros no vamos a usar ese potencial porque el objetivo es atenernos (dentro de lo posible) al concepto original de la modularidad. Si nos acostumbramos a pensar con más limitaciones, mejoramos el potencial cuando ya no las tenemos.

El ejemplo de la cocción a "Baño María" - que esperamos que nos haya permitido hacernos entender - nos permite observar otra cosa muy interesante.

En términos de módulo, yo puedo usar el de "Cocinar" y el de "Preparar Baño María" con independencia uno de otro. Es decir, puedo hacer un programa que use a "Cocinar" y no necesito tener a "Preparar Baño María". De la misma manera puedo hacer un programa que utilice a "Preparar Baño María" sin usar a "Cocinar". ¿Te queda claro el por qué? Pero no puedo usar nunca "Calentar a baño María" sin disponer además de los otros dos módulos. Porque "Calentar a Baño María" necesita "usar" o convocar a los otros dos módulos. ¿Lo ves?. Eso podríamos representarlo así:



Este es un gráfico de dependencias y estámos diciendo que "Calentar a Baño María" DEPENDE de "Preparar Baño María" y de "Calentar"

¿Para qué sirve esta noción de Dependencia y su contrapartida Independencia? Es fundamental para la programación y se estudia en varios paradigmas. El hecho es que cuanto más independiente sea un módulo, más oportunidad de reuso tiene y por lo tanto más útil es. Lo vamos a dejar aquí casi como un dogma de fé. En Programación Orientada a Objetos vamos a discutir mucho sobre este problema.

Parámetros de un módulo

Es altamente frecuente que, cuando invoquemos a un módulo como una función o un procedimiento, lo hagamos, además, enviándole un conjunto de datos para que trabaje. Pongamos un ejemplo para que se entienda mejor el concepto. Supongamos que queremos hacer un módulo que calcule la superficie de un rectángulo y lo llamamos superficie_rectángulo. A partir de esa premisa tendríamos dos opciones: el módulo se encarga de preguntar al usuario por el tamaño de cada uno de los dos lados necesarios para el cálculo o quien invoca a este módulo se los brinda para que el módulo realice solamente el cálculo. ¿Cuál de las dos opciones te parece más razonable?

Es obvio que cualquier camino que tomemos resuelve el problema pero las soluciones no son iguales. Un módulo es más útil que otro. Pensemos en esto: ¿Cómo vamos a preguntarle al usuario por los datos? Podemos hacerlo por consola, por una ventana con dos cajas de texto, por una web con dos cajas de texto, a través de un móvil, a través de un sensor que calcule distancias automáticamente en fin, hay un montón de formas para tomar datos. Lo mejor entonces sería generar independencia entre la captura de los datos y el cálculo en sí mismo. Para esto, el módulo debe recibir estos dos datos. A estos datos se les llama argumentos del módulo o parámetros del módulo.

Dicho esto, veamos algunos ejemplos en varios lenguajes de funciones y procedimientos con definiciones de argumentos o parámetros.

Vamos a los clásicos: **Pascal**

Procedimiento y Función en **Pascal**

```
// Procedimiento
procedure mostrar_mensaje(nombre : string);

begin
   writeln ('Hola ' + nombre + ', ¿qué tal estás?';);
end;

// Función
function calcular_area(base : integer, altura : integer) : integer;

begin
   calcular_area := base * altura;
end;
```

Observá como, en este lenguaje, existe un nombre diferente para definir funciones y procedimientos. También notá como en la función pasan dos cosas distintas al procedimiento: por un lado, luego de la lista de argumentos lleva dos puntos y el tipo de dato que devuelve la función. Eso no está en el procedimiento ¿Por qué? Porque el procedimiento NO DEVUELVE NADA.

Otra cosa a notar es que, en pascal, se utiliza la forma de igualar el nombre de la función (como si se tratase de una variable) al valor que se quiere devolver.

En **Visual Basic** por ejemplo se usa un formato parecido:

Procedimiento y Función en **Visual Basic**

```
' Procedimiento
sub mostrar_mensaje(nombre as String);
  debug.print ('Hola ' + nombre + ', ¿qué tal estás?';);
end sub;

' Función
function calcular_area(base as Integer, altura as Integer) as
Integer;
   calcular_area = base * altura;
end function;
```

En este lenguaje se usa sub para identificar procedimientos, la instrucción debug.print, el mensaje sale por un área tipo consola. En las funciones, el resultado también lo realiza asignando el nombre de la función el resultado.

Veamos ahora qué pasa en **C**

Procedimiento y Función en **C**

```
// Procedimiento
void mostrar_mensaje(char const *nombre){
   printf("%s",nombre);
}

// Función
int calcularArea(int base, int altura){
   return base * altura;
}
```

En C y los que heredan de C como C++ o Java para la definición de *métodos*[1], no hay un indicador de si se trata de una función o un procedimiento. La diferencia es que cuando se trata de un procedimiento, el "tipo" de dato se indica con **void** y no hay **return**. En cambio en las funciones hay **un tipo de dato indicado** y un **return** devolviendo el valor.

[1] Un método no es ni una función ni un procedimiento. Las palabras Procedimiento y Función quedan reservadas al ámbito de la programación estructurada y modular. Esto lo tratamos en Programación Orientada a Objetos

Veamos que pasa en **Python**

Procedimiento y Función en **Python**

```python
# Procedimiento
def mostrar_mensaje(mensaje):
    print(mensaje)

# Función
def calcular_area(base, altura):
    return base * altura
```

En **Python** pasa algo parecido pero, además, como es un lenguaje de tipado dinámico la única pista de la diferencia entre una función y un procedimiento te lo da que la función tiene un **return** y el procedimiento no.

Fijate que en **Go** pasa más o menos lo mismo.

Procedimiento y Función en **Go**

```go
// Procedimiento
func mostrarMensaje(pregunta string) {
  fmt.Printf("%s",pregunta)
}

// Función
func pedirValor(pregunta string) int32 {
  var valor int32
  fmt.Scanf("%d\n",&valor)
  return valor
}
```

En lugar de **def** se una **func** y hay dos diferencias entre función y procedimiento: La definición del tipo de dato que devuelve (aunque puede omitirse) y la palabra **return** para devolver el valor.

Pasaje de Parámetros por Valor y Por Referencia

Para comprender mejor esta cuestión, vamos a plantear un escenario. Observemos el siguiente código escrito en **Go** y reflexionemos sobre los posibles valores que van a tomar determinados elementos.

Escenario para pensar en **Go**

```go
1. package main
2.
3. import "fmt"
4.
```

```
5. func main() {
6.   const ELEMENTOS = 5
7.   var arreglo[ELEMENTOS] int32
8.   var pregunta string
9.   for i := 0; i < ELEMENTOS; i++ {
10.        pregunta = fmt.Sprint("Dime el número ", i + 1, " : ")
11.        arreglo[i] = pedirValor(pregunta)
12.        fmt.Printf("%s",pregunta)
13.    }
14.    fmt.Printf("Los numeros ingresados fueron : \n")
15.    for i := 0; i < ELEMENTOS;i++ {
16.       fmt.Printf(" %d \t %d \n", i+1, arreglo[i])
17.    }
18.  }
19.
20.  func pedirValor(mensaje string) int32 {
21.    var valor int32
22.    fmt.Printf("%s",mensaje)
23.    mensaje = "aquí cambió el mensaje"
24.    fmt.Scanf("%d\n",&valor)
25.    return valor
26.  }
```

A esta altura, luego de leer varias guías y códigos en distintos lenguajes verás que la mayoría se parece bastante.

En Go, el programa empieza por la función main() (línea 5), luego define una constante llamada ELEMENTOS en 5 (línea 6), define un arreglo y una string (líneas 7 y 8 respectivamente), Acá viene la cuestión : en un **for** que comienza con **i** valiendo *0* hasta cantidad de *ELEMENTOS - 1*, asigna la string **pregunta** al valor : *"Dime el número "*, luego el número de iteración (índice más uno) y los dos puntos. Esta cadena, va como parámetro a la función **pedirValor** que lo recibe con el nombre de **mensaje**. En la línea 23, **mensaje** se re asigna a *"aquí cambió el mensaje"*, y la función termina devolviendo el valor ingresado por el usuario, que se almacena en **arreglo[i].** En la línea 12 se imprime por pantalla el valor de la variable **pregunta**. ¿Qué crees que se va a mostrar en esa línea?

Es obvio que la respuesta estará entre dos posibles:

a - Se imprime *"Dime el número x :"* (donde x será el número de iteración)

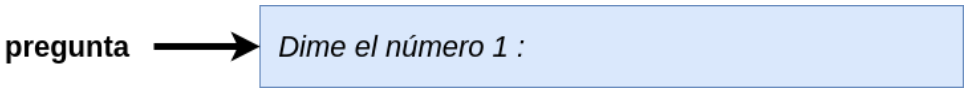
b - Se imprime *"aquí cambió el mensaje"*

Si la respuesta es (a), significa que la reasignación en la línea 23, no alteró el valor de la variable **pregunta**. Si, por el contrario, la respuesta es (b), la reasignación, surgió efecto en la variable **pregunta**

Acá viene una cuestión muy importante! Un módulo puede recibir sus parámetros de dos formas: por valor o por referencia.

¿Qué entendemos por "pasar una variable"? Básicamente es entregarle a una función o procedimiento, al momento de ser invocada, algún valor preexistente. En programación, este valor puede ser una constante (indicar el texto entre comillas directamente por ejemplo) o una variable. Una variable, como ya vimos y viste en otras asignaturas, no es más que un contenedor que permite

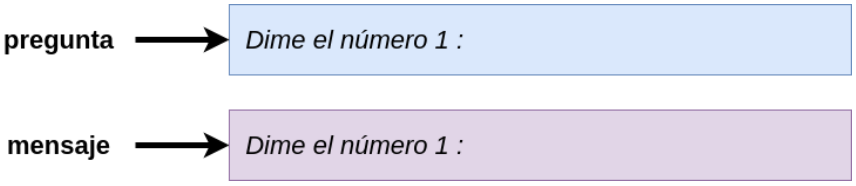
colocar valores por un tiempo bajo un nombre conveniente. La variable **pregunta** en nuestro caso es un ejemplo de ello. Esto puede ser representado con este esquema:



En este esquema queremos indicar que la variable **pregunta** es una forma conveniente de llamar a una localización de memoria donde se encuentra localizada esta cadena. Podríamos suponer que esa dirección es FAA5-00512[2].

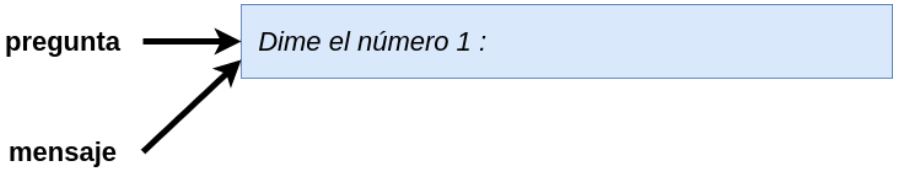
Como decíamos al principio, en el mundo de la programación existen dos formas de pasar variables a algún módulo: una se conoce como paso por valor y la otra es conocida como paso por referencia.

Si la variable **pregunta** se pasa al módulo por valor, no se pasa la variable sino el valor que ésta contiene. En nuestro ejemplo, no se pasa FAA5-0512 sino que se pasa *"Dime el número 1 :"*. Por lo tanto, la función **pedirValor** no tiene ni idea en qué lugar se guardó el contenido de la variable **pregunta**, sino que crea una nueva variable, en otro lugar, y la referencia por el nombre **mensaje.**



En ella pone el valor *"Dime el número 1 :"* Si así fuera, la reasignación de la línea 23 no afecta a la variable **pregunta**, solo se altera la variable **mensaje**.

Si la variable **pregunta** se pasa al módulo por referencia, no se pasa el valor sino la dirección de memoria donde la variable está guardada. En nuestro ejemplo, se pasa FAA5-0512. Por lo tanto, ahora, la función **pedirValor** al recibir el parámetro **mensaje**, referencia con otro nombre a la misma variable referenciada por pregunta. Observa el siguiente esquema



Como ves en el esquema, ambas variables apuntan a la misma variable, por lo tanto la reasignación de la línea 23 afecta a la variable **pregunta** y la altera.

Tal vez en este punto quieras saber cuál era la respuesta correcta en el ejemplo en Go. La respuesta correcta es (a), porque la variable es pasada por valor.

En **Go**, como en **C**, una variable se pasa por referencia añadiendo un **&** adelante. Con esta explicación, por ahí queda más clara la línea 24 (`fmt.Scanf("%d\n",&valor)`) Observá que la variable **valor** tiene un & adelante. Esto significa que al procedimiento Scanf se le pasa la variable **valor** por referencia. Si, Scanf es un procedimiento y para poder devolver un valor (que es el ingresado por el usuario) como no puede hacerlo como retorno lo que hace es alterar el valor de la variable enviada por referencia. Es un buen truco ¿no? Durante mucho tiempo, este fue un recurso

[2] es un ejemplo, el tema del direccionamiento a memoria se ve en otras asignaturas

muy usado y aún se sigue usando. Cuando se necesita que un módulo devuelva más de un valor, puedo enviar varias variables por referencia y de esta forma, al regresar el control, vuelven alteradas con los valores necesarios.

Ámbito de una variable

Volvé a mirar el ejemplo y Go y mirá la función **pedirValor**. Vas a ver que en ella hay dos variables (mensaje y valor) que son locales a la función. ¿Esto qué significa? Significa que fueron definidas DENTRO de la función y, por lo tanto, su existencia termina al finalizar la función.

El lugar donde una variable es visible (tiene sentido usarla) se lo conoce como ámbito de una variable. Una variable fuera de su ámbito no tiene significado alguno. En otras palabras, el ámbito se refiere a una región del programa en la que una variable existe.

El ámbito de una variable, su alcance y definición dependen fuertemente del lenguaje de programación que se utilice. Por ejemplo, en los lenguajes que utilizan bloques de código demarcados con inicios y fin, como las llaves en **Java**, **Go** o **C** o los *begin* y *end* de **Pascal**, las variables definidas dentro de un bloque suelen ser sólo visibles dentro del bloque. En la siguiente imagen se utilizaron colores diferentes para diferenciar ámbitos distintos en **Go**:

```
func main(){
        for i:=0;i<3;i++{
                fmt.Printf("i: %d\n", i)
                for j:=0;j<3;j++{
                        var num = 3
                        fmt.Printf("num: %d\n", num)
                        fmt.Printf("j: %d\n", j)
                }
                fmt.Println(num) /*Error*/ 1
        }
        fmt.Println(i) /*Error*/ 2
        fmt.Printf("Resultado almacenado en una variable: %d\n", res)
}

func mult(numero1, numero2 int) int{
        /*Variable local de la función*/
        var resultado int
        resultado = numero1 * numero2
        return resultado
}
```


Ámbito de variables (Go)[3]

- Las variables del ámbito **verde** pueden ser consultadas por los ámbitos **rojo** y **amarillo**.
- Las variables del ámbito **rojo** y **amarillo** no pueden ser consultadas por el ámbito **verde**.

[3] Imágen y descripción extraída de https://codingornot.com/08-go-to-go-funciones-y-ambitos

- Las variables del ámbito **rojo** se pueden acceder desde el ámbito **amarillo**.
- Las variables del ámbito **amarillo** no se pueden acceder desde el ámbito **rojo**.
- Las variables del ámbito **cian** no pueden consultarse desde ningún otro ámbito y viceversa.
- Los número **1** y **2** marcan dos sentencias erróneas que intentan acceder a variables de un ámbito al que no pueden hacerlo.
- En el ámbito **amarillo** se utiliza la variable ***i*** que pertenece al ámbito **rojo**.

En Go, Las variables pueden ser declaradas en 3 lugares distintos:

1. Dentro de una función o bloque de código. En este caso las variables son conocidas como locales, y solamente se puede acceder a ellas dentro del ámbito al que pertenecen y desde otros internos a él.
2. Fuera de todas las funciones. En este caso se les conoce como variables globales y se puede acceder a ellas desde cualquier parte del programa.
3. En la definición de una función. Se les conoce como parámetros formales y solamente se puede acceder a ellas desde la función en la que se declaran.

Ámbito de variables en **Go**

```
package main

import "fmt"
// variable global
var variableGlobal int

func main() {
  /* variables locales a main */
  const ELEMENTOS = 5
  var arreglo[ELEMENTOS] int32
  var pregunta string
  variableGlobal = 5;
  for i := 0; i < ELEMENTOS; i++ {
     //i es variable local al bloque
     pregunta = fmt.Sprint("Dime el número ", i + 1, " : ")
     arreglo[i] = pedirValor(pregunta)
  }
  fmt.Printf("Los numeros ingresados fueron : \n")
  for i := 0; i < ELEMENTOS;i++ {
     fmt.Printf(" %d \t %d \n", i+1, arreglo[i])
  }
}

func pedirValor(pregunta string) int32 {
  var valor int32
  fmt.Printf("%d",variableGlobal)
  fmt.Printf("%s",pregunta)
  fmt.Scanf("%d\n",&valor)
  return valor
}
```

Veamos que pasa en **Python** para que veas que las cosas varían mucho de lenguaje en lenguaje

Ámbito de variables en **Python**

```
def procedimiento():
    c = 3
    print('La variable c dentro del procedimiento tiene por valor', c)

a = 1
b = 2
c = 5

print(a)
print(b)
procedimiento()
print('La variable c fuera del procedimiento tiene por valor', c)
```

Aquí tenés la salida de este programa:

Ejecución

```
1
2
La variable c dentro del procedimiento tiene por valor 3
La variable c fuera del procedimiento tiene por valor 5
```

Como ves, la variable c tiene un valor distinto dentro del cuerpo del programa que dentro del procedimiento.

Podemos decir que una variable definida dentro de una función o procedimiento tiene un ámbito local a ese procedimiento o función. ¿Se puede exceder ese ámbito en Python? Sorprendentemente SÍ

Observa este ejemplo (**muy feo ejemplo**)

Ámbito de variables en **Python**

```
def procedimiento():
    c = 3
    global b
    b = 5
    print('La variable c dentro del procedimiento tiene por valor', c)
    print('La variable b dentro del procedimiento tiene por valor', b)

a = 1
b = 2
c = 5
```

```
print(a)
print(b)
procedimiento()
print('La variable c fuera del procedimiento tiene por valor', c)
print('La variable b fuera del procedimiento tiene por valor', b)
```

Aquí tenés la salida de este programa:

Ejecución

```
1
2
La variable c dentro del procedimiento tiene por valor 3
La variable b dentro del procedimiento tiene por valor 5
La variable c fuera del procedimiento tiene por valor 5
La variable b fuera del procedimiento tiene por valor 5
```

Es espantoso ¿verdad? Observa como un procedimiento puede "volarte" el control de una variable que suponías controlada.

Actividad 1

Investiga cómo se define el ámbito de una variable en otro lenguaje distinto a Go y Python, elegí vos el lenguaje y escribí en el foro armado para tal fin : - ¿Cuales son los ámbitos?, ¿Cómo se definen? y cualquier otro dato que creas relevante.

Ahora, a participar!!

Escribe tu respuesta en el foro preparado a tal efecto. Recuerda que en este foro solo podrás ver las respuestas de tus compañeros luego de que hayas puesto la tuya.

Actividad 2

En el lenguaje elegido para la actividad 1, ¿Hay funciones y procedimientos claramente diferenciados?

Ahora, a participar!!

Escribe tu respuesta en el foro preparado a tal efecto. Recuerda que en este foro solo podrás ver las respuestas de tus compañeros luego de que hayas puesto la tuya.