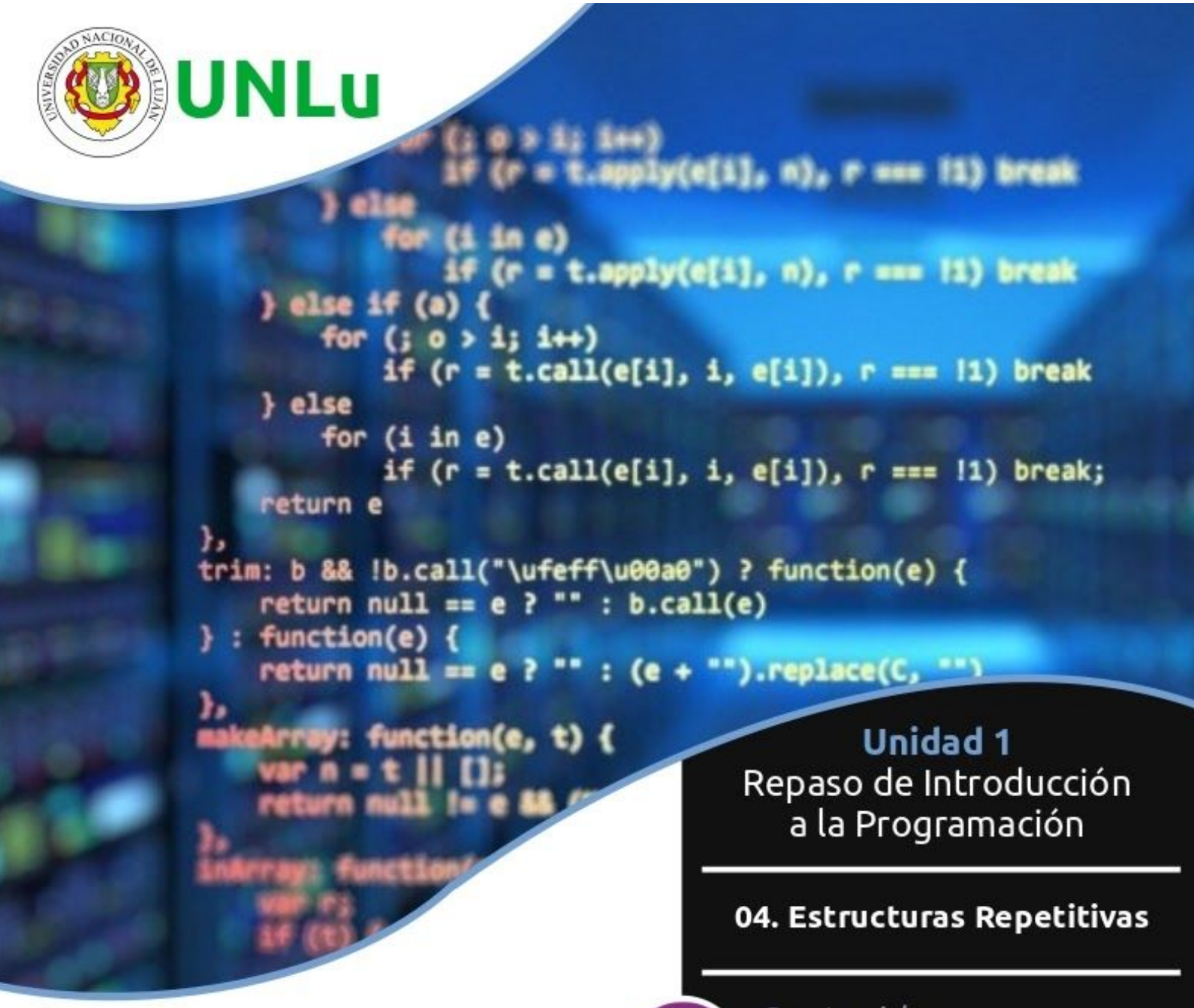
UNLu



Unidad 1
Repaso de Introducción a la Programación

04. Estructuras Repetitivas

11074 – Programación I

Material didáctico preparado para el dictado de la actividad académica.

Contenido
Aquí hablamos de las **estructuras repetitivas** que es otra de las formas de controlar el flujo de ejecución de un programa

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 la guía anterior hablábamos de la Programación Estructurada como paradigma y cubrimos dos de las tres estructuras que establece este paradigma: La estructura secuencial y la de selección. En esta guía nos enfocamos en las estructuras repetitivas.

Estructuras de repetición

Las computadoras se diseñaron inicialmente para realizar tareas sencillas y repetitivas. En contrapartida, el ser humano no es bueno realizando estas tareas repetitivas: pronto le falla la concentración y comienza a tener descuidos. Las computadoras, en cambio, pueden realizar la misma tarea muchas veces por segundo durante años y nunca se aburren (o, al menos, hasta hoy no se ha tenido constancia de ello).

La estructura repetitiva, por tanto, reside en la naturaleza misma de las computadoras y consiste, simplemente, en repetir varias veces un conjunto de instrucciones. Las estructuras repetitivas también se llaman bucles, lazos o iteraciones.

Los bucles tienen que repetir un conjunto de instrucciones un número finito de veces. Si no, nos encontraremos con un bucle infinito y el algoritmo no serviría para nada. En rigor, ni siquiera será un algoritmo, ya que no cumplirá la condición de finitud.

El bucle infinito es un peligro que acecha constantemente a los programadores y nos toparemos con él muchas veces a lo largo de este curso. Para conseguir que el bucle se repita sólo un número finito de veces, tiene que existir una condición de salida del mismo, es decir, una situación en la que ya no sea necesario seguir repitiendo las instrucciones.

Por tanto, los bucles se componen, básicamente, de dos elementos:

- **un cuerpo del bucle**: que es el conjunto de instrucciones que se ejecutan repetidamente
- **una condición de salida** : para dejar de repetir las instrucciones y continuar con el resto del algoritmo

Esencialmente se pueden definir tres tipos de bucles. Cada uno posee características propias que se especializan en diferentes situaciones contextuales:

- Bucle “mientras“ (o “while”)
- Bucle “repetir hasta“ (o “repeat ... until”)
- Bucle “para“ (o “for”)

El bucle “mientras”

El bucle “mientras” o “while” es una estructura que se repite mientras una condición dada se mantenga como verdadera. La condición se prueba ANTES de comenzar el bucle y si esta resulta falsa el bucle no se ejecuta. Si por el contrario es verdadero, ejecuta las sentencias incluidas en el cuerpo del bucle y vuelve a probar la condición. Si es verdadera repite de nuevo el cuerpo del bucle y si es falsa termina.

Es importante que comprendas que cuando la condición es falsa, la siguiente línea que se ejecuta es la que le sigue a la última instrucción incluida en el cuerpo del bucle de repetición

Formato de un ciclo "mientras" en pseudocódigo

Código ...

```
mientras <condición> hacer
inicio
     {cuerpo del bucle}
     acciones 1;
     acciones 2;
     …
     acciones n;
fin;
```

Entonces repasemos los conceptos principales de cómo funciona este ciclo y porque es un ciclo: El programa viene ejecutándose secuencialmente. Cuando se llega a una instrucción mientras, primero, evalúa la condición. Si es verdadera, se realizan las acciones secuenciales incluidas en el cuerpo del ciclo. Al terminar el bloque de acciones, se regresa a la instrucción mientras (he aquí el concepto de bucle o lazo). En este punto, se vuelve a evaluar la condición y, si sigue siendo verdadera, vuelve a repetirse el bloque de acciones. Y así, sin parar, hasta que la condición se haga falsa.

Veamos un ejemplo un tanto más complejo y completo: vamos a escribir un algoritmo que realice una división por restas sucesivas. La idea es que, el usuario nos pasa dos números y el programa debe mostrar cuántas veces entra el primero en el segundo (ese es básicamente el concepto de división entera)

Código ...

```
Variables ENTERO: dividendo, divisor, resto, resultado;


dividendo <- Pedir("Dime el número que quieres dividir : ");
divisor <- Pedir("Dime el número por el cual quieres dividirlo : ");
resto <- dividendo
resultado <- 0
diviOriginal <- dividendo
Mientras (dividendo >= divisor) Hacer
      dividendo = dividendo - divisor
      resultado <- resultado + 1
      resto <- dividendo
Fin Mientras
Escribir("El resultado de dividir ",diviOriginal," por ",divisor," es
",resultado," y el resto es ",resto);
```

En esta estrategia, lo que se ha considerado es lo siguiente: Supongamos que el usuario pone un número más pequeño en el dividendo que el divisor, por ejemplo 2 y 6 respectivamente. En este caso la división es 0 y el resto es 2. Es decir, no entra ninguna vez el 6 en el 2 y sobran dos. Si no fuera así y, por ejemplo, si las entradas fueran 5 y 3, diríamos que entra 1 vez (qué es lo que se almacena en el contador de veces que resta llamado ***resultado***) y el resto sería 5 - 3 = 2. Debe notarse que, como la variable *dividendo* va a ser alterada para el cálculo y el resultado deberá ser mostrado al final, se debe preservar el valor original de la variable *dividendo*, en nuestro caso, almacenándola en *diviOriginal*

Observemos de cerca la evolución del algoritmo:

1. Pedimos al usuario el valor para el dividendo
2. Pedimos al usuario el valor para el divisor
3. El resto se iguala al dividendo por si fuera menor al divisor.
4. El resultado se iguala a 0 porque es el valor correcto en caso de que no se ingresara nunca al ciclo y, además, porque es un contador.
5. Mientras que el dividendo sea >= al divisor sea verdadera
6. restar del dividendo, el divisor
7. sumar uno al resultado (pudo restarse otra vez)
8. igualar resto al divisor (por si fuera la última que se resta)
9. Después, el flujo del programa regresa a la instrucción mientras (punto 5), ya que estamos en un bucle, y se vuelve a evaluar la condición.
10. Escribir el resultado

Si pudiéramos ejecutar este programa, se vería algo así (se muestra la salida con dos entradas distintas):

Ejecución

```
<<Primera Ejecución>>
Dime el número que quieres dividir : 4
Dime el número por el cual quieres dividirlo : 3
El resultado de dividir 4 por 3 es 1 y el resto es 1

<<Segunda Ejecución>>
Dime el número que quieres dividir : 6
Dime el número por el cual quieres dividirlo : 6
El resultado de dividir 6 por 6 es 1 y el resto es 0
```

Una buena observación que podemos hacer aquí es que, a priori, no se sabe cuántas veces el ciclo va a ser ejecutado (cuantas vueltas va a dar) y por eso, como el resultado es igual a las veces que se produce el ciclo, hay un contador que las cuenta. Lo que sí se sabe es bajo qué condición se debe

continuar repitiendo el ciclo. También se sabe que, si la condición es falsa la primera vez (que sería cuando el divisor es mayor al dividendo) no se debe ejecutar ninguna vez este ciclo.

Te animamos a pensar … ¿Que pasaría en este algoritmo si el divisor fuera 0? Tal vez, en esta respuesta entiendas el porqué la división por 0 en los números reales da como resultado infinito.

Resumiendo y Destacando!

Este ciclo se usa cuando no es posible de antemano anticipar cuántas veces se debe ejecutar el cuerpo del ciclo y, además, puede que no sea necesario ejecutarse ni una vez

Lo más problemático a la hora de diseñar un bucle es, por lo tanto, pensar bien su condición de salida, porque si la condición de salida nunca se hiciera falsa, caeríamos en un bucle infinito. Por lo tanto, la variable implicada en la condición de salida debe sufrir alguna modificación en el interior del bucle; si no, la condición siempre sería verdadera y generaría lo que se llama **un bucle infinito**. En nuestro ejemplo, la variable **dividendo** se modifica en el interior del bucle disminuyendo su valor, por eso llega un momento, después <n> repeticiones que la condición se hace falsa y el bucle termina.

El bucle "repetir"

El bucle de tipo "repetir" es muy similar al bucle "mientras", con la salvedad de que la condición de salida se evalúa al final del bucle, y no al principio, como veremos a continuación.

La forma de la estructura "repetir" es la que sigue:

Código …

```
repetir
     {cuerpo del bucle}
     acciones 1;
     acciones 2;
     …
     acciones n;
hasta que <condición>;
```

Cuando la computadora encuentra un bucle de este tipo, ejecuta las acciones escritas entre el inicio (repetir) y fin (hasta que) y, después, evalúa la condición, que debe ser de tipo lógica. Aquí deberás apreciar la segunda diferencia con el bloque mientras y la más importante: Como no tiene condición en la entrada, el bloque **repetir** se ejecuta siempre, al menos una vez. Dependiendo del lenguaje, suele ser que el repetir se haga: ***Repetir Hasta*** y en este caso la condición del final del ciclo funciona al revés que en el caso anterior (debe ser verdadera para salir) o ***Repetir Mientras*** (donde la condición debe evaluarse como falsa para terminar).

Para nuestro ejemplo en pseudo código, tomaremos la implementación original de este algoritmo que es del tipo repetir hasta, es decir que para finalizar el ciclo, la condición debe ser verdadera (justo lo contrario del ciclo Mientras).

Este tipo de ciclos son muy útiles para validar datos de entrada y que el programa insista preguntando hasta que el dato sea correcto. Tomemos por caso el ejemplo de escribir el día de la semana en función del número de día, donde 1 era Lunes, 2 era Martes, etc. que fue desarrollado en la guía sobre selección múltiple.

Código ...

```
Variables ENTERO: dia;

Repetir
      dia <- Pedir("Dime el número de dia (de 1 a 7) : ");
Hasta que (dia >= 1 .Y. dia <= 7) //nótese el operador lógico .Y. o and aquí

Según dia Sea
      1 : Escribir('Lunes');
      2 : Escribir('Martes');
      3 : Escribir('Miércoles');
      4 : Escribir('Jueves');
      5 : Escribir('Viernes');
      6 : Escribir('Sábado');
      7 : Escribir('Domingo');
Fin Según
Escribir("Fin del Programa");
```

Si pudiéramos ejecutar el programa tres veces veríamos :

Ejecución

```
<<Primera Ejecución>>
Dime el número de dia (de 1 a 7) : 5
Viernes
Fin del Programa

<<Segunda Ejecución>>
Dime el número de dia (de 1 a 7) : 7
Domingo
Fin del Programa

<<Tercera Ejecución>>
Dime el número de dia (de 1 a 7) : 9
Dime el número de dia (de 1 a 7) : -5
Dime el número de dia (de 1 a 7) : 4
Jueves
Fin del Programa
```

Como se puede observar, cuando la entrada es válida (el valor del día es mayor o igual a uno y menor o igual que siete, el ciclo no se repite, pero si fuera falso, se continúa repitiendo hasta que sea verdadero.

Resumiendo y Destacando!

Este ciclo se usa cuando no es posible de antemano anticipar cuántas veces se debe ejecutar el cuerpo del ciclo y, además, **debe** ser ejecutado al menos una vez

El bucle "para"

En muchas ocasiones se conoce de antemano el número de veces que se desean ejecutar las acciones del cuerpo del bucle. Cuando el número de repeticiones es fijo, lo más apropiado es usar un bucle "para".

La estructura "para" repite las acciones del bucle un número prefijado de veces e incrementa automáticamente una variable contador en cada repetición. Su forma general es:

Código ...

```
Para contador := <valor_inicial> hasta <valor_final> hacer
inicio
     {cuerpo del bucle}
     acciones 1;
     acciones 2;
     …
     acciones n;
fin;
```

contador, como su nombre lo indica es la variable de tipo contador, es decir que se incrementa de uno en uno. Lo interesante de este tipo de bucles es que la variable contador se incrementa automáticamente cada vez que se termina de ejecutar el bloque repetitivo.

La primera vez que se ejecutan las acciones situadas dentro del ciclo, a la variable *contador* se le asigna el valor especificado en la expresión <valor_inicial>. En la siguiente repetición, *contador* se incrementa en una unidad, y así sucesivamente, hasta alcanzar el <valor_final>. Cuando esto ocurre, el bucle se ejecuta por última vez y después el programa continúa en la instrucción que haya a continuación de "fin". El incremento de la variable *contador* siempre es de 1 en cada repetición del bucle (en algunos lenguajes esto se permite variar agregando un modificador en la instrucción). El ciclo "para" se puede escribir también de manera que en lugar de ir incrementándose en 1 el valor de *contador*, se vaya decrementando en 1. Por esta razón, la estructura "para" tiene una sintaxis alternativa:

Código ...

```
Para contador := <valor_inicial> bajando hasta <valor_final> hacer
inicio
     {cuerpo del bucle}
     acciones 1;
     acciones 2;
     …
     acciones n;
fin;
```

De esta forma, se puede especificar si la variable *contador* debe incrementarse o decrementarse en cada repetición.

Ejemplo 1: Escribir un algoritmo que escriba todos los números enteros entre 1 y 100. En este caso, se ve claramente que la cantidad de repeticiones necesarias se sabe de antemano. Entonces, el bucle apropiado es el "para". De esta forma, el código en pseudocódigo sería como sigue:

Código ...

```
Programa escribirNumeros;
variables
  contador : entero;
inicio
  para contador := 1 hasta 100 hacer
  inicio
    escribir (contador);
  fin;
fin.
```

De nuevo, lo más interesante es observar las diferencias de este algoritmo con los dos ejemplos anteriores. Advierte que ahora no es necesario asignar un valor inicial de 0 al contador, ya que se hace en el mismo bucle; y tampoco es necesario incrementar el valor de *contador* en el cuerpo del bucle (*contador* = *contador* + 1), ya que de eso se encarga el propio bucle "para". Por último, no hay que escribir condición de salida, ya que el bucle "para" se repite hasta que *contador* alcance el valor de 100 (inclusive)

Cuadro Destacado

El ciclo "para" es quizás el que más varía entre lenguajes y lenguajes. Casi todos los lenguajes lo incluyen pero no todos de la misma forma. Algunos, además, incorporan un ciclo "para" especial para elementos especiales como colecciones o arreglos. Veremos algún ejemplo de este ciclo cuando lleguemos a la parte de arreglos.

Comparando los tres ciclos

Habrás observado que hay pocas diferencias entre el ciclo "Mientras" y el "Repetir" pero hay algunas más entre éstos y el ciclo "para".

Te brindamos un cuadro a modo resumen final :

Ciclo	Uso

Mientras	Se utiliza cuando la cantidad de repeticiones a realizar no se conoce antes de comenzar el ciclo. Además cuando es posible que no se ejecute tan siquiera una vez.
Repetir Hasta	Se utiliza cuando la cantidad de repeticiones a realizar no se conoce antes de comenzar el ciclo pero es necesario ejecutar el ciclo al menos una vez
Para	En este ciclo es necesario saber con antelación la cantidad de veces que es necesario repetir el cuerpo del bucle. Se considera una muy mala práctica manipular el índice de control del ciclo (la variable contador en nuestros ejemplos) dentro del cuerpo.

Lamentablemente no todos los lenguajes de programación implementan los tres tipos de ciclo y si lo hacen, lo hacen de manera muy distinta. Cuando te aboques a aprender un lenguaje de programación deberás investigar cómo se definen los ciclos de repetición en él.

Para ejemplificar esto te mostramos la implementación de los ciclos en algún lenguaje en particular.

Empezamos con **Python**.

En este lenguaje**, el ciclo Repetir no está implementado**. Veremos entonces un ejemplo del Mientras

Ciclo Mientras en Python …

```
dividendo = int(input("Dividendo : "))
divisor = int(input("Divisor :"))
resto = dividendo
resultado = 0
dividendo_original = dividendo
while dividendo >= divisor:
   resultado = resultado + 1
   dividendo = dividendo - divisor
   resto = dividendo

print("El resultado de dividir ",dividendo_original," por ",divisor," es ",
resultado, " y el resto es ", resto)
```

Fijate que distinto es el for en Python a diferencia del que vimos en pseudo código:

Ciclo Para en Python …

```
for x in range(1,10):
   print(x)
print("Fin del programa")
```

En este caso, la variable x será asignada a 1 al comienzo y luego llegará hasta valer **9**, que es lo que indica el *rango (range)*. Como te darás cuenta, el último valor del rango no está incluído. También se puede indicar que querés que se incremente de dos en dos (o el número que desees) poniendo un

tercer número en la instrucción ***range***. El siguiente ejemplo imprime los números impares desde el 1 al 9.

Ciclo Para en Python con paso 2 ...

```
for x in range(1,10,2):
   print(x)
print("Fin del programa")
```

Que pasa en Go. En Go, no existe ni el ciclo Mientras ni el Repetir!!! Pero ... y entonces, ¿cómo programo? ¿Siempre debo saber la cantidad de veces que se va a repetir el ciclo?. La respuesta es "no". En Go, el ciclo Para es tan versátil que te permite, con la misma instrucción emular los tres comportamientos.

Empezamos por el ciclo Para de la forma "normal"

Ciclo Para en Go ...

```
package main

import "fmt"

func main() {
  for x:= 1; x <= 10; x++ {
         fmt.Printf("%d\n",x)
  }
}
```

Expliquemos un poquito cómo funciona este ciclo. Como Go es un derivado del lenguaje C y, al igual que C++ y Java, utiliza un patrón similar para este ciclo. La firma (o forma) completa cuenta con tres partes:

1. Declaración e inicialización : Esta parte corresponde a x:=1. La asignación con **:=** en lugar de solo el signo igual implica, en Go, que se está pidiendo que se declare la variable *x* utilizando un tipeo dinámico (si no sabes qué es esto vuelve a la guía sobre variables). Observa que no se indica el tipo de dato del cual es *x*, entonces el compilador debe asumirlo. Por otra parte se dice que *x* comienza valiendo 1.
2. Validación : La segunda parte debe ser de tipo *booleana*, es decir una condición y especifica qué es lo que debe ser verdad para reiniciar el ciclo repetitivo.
3. Incremento : En la tercera parte, se especifica el incremento del contador. En este caso x ++ es lo mismo que decir x = x + 1. Por lo tanto se incrementa en 1. Si pusiéramos x += 2 el salto se produciría de dos en dos (como en el ejemplo de Python)

Ok, ¿cómo puedo emular con este ciclo un Mientras?. Recordemos que en un Mientras, la variable de control no existe, sino que se construye y calcula dentro del ciclo. Esta comparación es equivalente a la sección de validación del ciclo for de Go. Pero en Go no nos obligan a poner la primera sección y tampoco la última. Por lo tanto podríamos escribir : **for x <= 10 {... }** ok .... y ¿cómo controlamos el comienzo y el fin? Ahora todo ese control lo debés hacer vos a mano. Aquí un ejemplo:

Ciclo Para en Go emulando un Mientras ...

```
package main

import "fmt"

func main() {
  var dividendo int32
  var divisor int32
  var resultado int32
  var resto int32
  var divOriginal int32
  fmt.Printf("Dividendo : ")
  fmt.Scanf("%d\n",&dividendo)
  fmt.Printf("Divisor : ")
  fmt.Scanf("%d\n",&divisor)
  resultado = 0
  resto = dividendo
  divOriginal = dividendo
  for dividendo >= divisor {
     resultado ++
     dividendo -= divisor
     resto = dividendo
  }
    fmt.Printf("El resultado de dividir %d por %d es %d y el resto es
%d\n",divOriginal,divisor,resultado,resto)
}
```

En este ejemplo, el **for** solo tiene el control del ciclo y, al igual que en el **Mientras**, si la condición es falsa antes de comenzar, no entra al ciclo.

¿Cómo podemos emular el Repetir ... Hasta?

Recordemos que en este ciclo se debe garantizar que, al menos, se ejecute una vez.

Entonces, hay que hacer desaparecer la condición de entrada. El problema es que si hacemos esto, es como hacer un bucle infinito. Entonces hay que agregar una condición al final que, de cumplirse, no quiebre el ciclo. Es decir, fuerce la salida de él.

Aquí te mostramos un ejemplo:

Ciclo Para en Go emulando un Repetir .. Hasta ...

```
package main

import "fmt"

func main() {
  var dia int8

  for {
     fmt.Print("Dime el número de día ( entre 1 y 7 ) :")
     fmt.Scanf("%d\n", &dia)
     if dia > 1 && dia <= 7 {
```

```
            break
        }
    }
    switch dia{
    case 1:
        fmt.Println("Lunes")
    case 2:
        fmt.Println("Martes")
    case 3:
        fmt.Println("Miercoles")
    case 4:
        fmt.Println("Jueves")
    case 5:
        fmt.Println("Viernes")
    case 6:
        fmt.Println("Sábado")
    case 7:
        fmt.Println("Domingo")
    default:
        fmt.Println("No es un día correcto")
    }
    fmt.Println("Fin del Programa")
}
```

¿Observaste la instrucción break en el condicional que evalúa la entrada del día es correcta?. Eso no está bien visto en programación estructurada pero, dentro del lenguaje Go es la forma de resolver la ausencia del **Repetir Hasta**.

Actividad 1

Piensa en un algoritmo que te permita hallar el máximo común divisor y el mínimo común múltiplo de dos números naturales que son ingresados por el usuario.

Para hacerlo, busca la información que necesites en biblioteca o internet y observa que valores de los que usas pueden ser expresados y definidos como constantes para poder expresar tu algoritmo más apropiadamente.

Una vez obtenida toda la información pregúntate qué tipo de bucle es el más apropiado para resolver el problema. Para ello, deberás identificar qué partes de la resolución son repetitivas y cuál es la condición de salida de la repetición. Piensa también si hay alguna manera de saber de antemano qué cantidad de veces se deberá repetir el ciclo. Esto te ayudará a descartar o adoptar rápidamente la decisión.

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.

Puedes comentar, una vez puesta tu respuesta, las de tus compañeros. Es importante que compartamos apreciaciones y pareceres.

Actividad 2

Piensa en un algoritmo que te permita calcular la suma y el producto de los números pares comprendidos entre 20 y 30.

Para hacerlo, busca la información que necesites en biblioteca o internet y observa que valores de los que usas pueden ser expresados y definidos como constantes para poder expresar tu algoritmo más apropiadamente.

Al igual que en el caso anterior, una vez obtenida toda la información pregúntate qué tipo de bucle es el más apropiado para resolver el problema. Para ello, deberás identificar qué partes de la resolución son repetitivas y cuál es la condición de salida de la repetición. Piensa también si hay alguna manera de saber de antemano qué cantidad de veces se deberá repetir el ciclo. Esto te ayudará a descartar o adoptar rápidamente la decisión.

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.

Puedes comentar, una vez puesta tu respuesta, las de tus compañeros. Es importante que compartamos apreciaciones y pareceres.

Aprovecha la oportunidad INVALUABLE que te da la virtualidad de manejar tus tiempos para comparar las respuestas de tus compañeros con la tuya. Fijate cómo resolvieron los problemas y de esa forma aprenderás mucho más. El intercambio favorece enormemente la experiencia de enseñanza-aprendizaje.