

ALGORITMIA Y PROGRAMACIÓN

I. EXPRESIONES

Se entiende como expresiones, sentencias que involucran el uso de constantes, variables y operadores. Las expresiones se clasifican de acuerdo al tipo de datos y operadores que emplean, lo cual implica también el tipo de resultado que se obtiene en su desarrollo o evaluación.

OBSERVACIÓN:

- Se definen como **constantes** datos que no cambian su valor, por ejemplo un número (3 ó 5.34).
- Se definen como **variables** datos que pueden cambiar su valor, lo que generaría un cambio en el valor de la expresión que los involucra, a menudo se representan con letras.

Expresiones aritméticas

Una expresión aritmética es aquella que incluye variables numéricas (enteras o reales), constantes numéricas y operadores aritméticos.

Operadores aritméticos

+	Operador de suma
-	Operador de resta
*	Operador de multiplicación
/	Operador de división

Algunos lenguajes pueden emplear otros símbolos para representar las diferentes operaciones o pueden incluir nuevos operadores como el de módulo o residuo.

Prioridad: Un aspecto importante para revisar cuando se estudian los operadores aritméticos en cada lenguaje es la prioridad. La prioridad corresponde a las reglas que indican cuál operación debe realizarse primero en una expresión aritmética; aunque estas reglas pueden variar de un lenguaje a otro, en general la mayor prioridad la tienen los operadores de multiplicación y división, después seguirían los de suma y resta.

Cuando se encuentran dos o más operadores del mismo nivel de prioridad, las operaciones se resuelven de izquierda a derecha.

El uso de parentesis permite saltar las reglas de prioridad para obligar a que se desarrolle una operación primero, dado que las operaciones incluidas en parentesis tendrían la mayor prioridad en la expresión.

Ejemplo:

$4 + 8 * 2 - 6 / 2$	Para resolver esta expresión se realizaría primero la multiplicación $8 * 2 (=16)$, luego la división $6 / 2 (=3)$, luego la suma $4 + 16 (=20)$ y finalmente la resta $20 - 3$, dando como resultado 17.
$(4 + 8) * (2 - 6) / 2$	En este caso se ha alterado la prioridad gracias al uso de parentesis, aquí se realiza primero la suma $4 + 8 (=12)$, luego la resta $2 - 6 (= -4)$, luego la multiplicación $12 * -4 (= -48)$ y finalmente la división $-48 / 2$ dando como resultado -24.

Como se puede observar en el ejemplo anterior, el uso de parentesis cambia completamente el resultado de una expresión, así que debe tenerse mucho cuidado al momento de escribir expresiones aritméticas para verificar que las prioridades harán que se tenga el resultado que realmente se espera.

Para pensar....

¿ La prioridad que emplean los lenguajes de programación, es la misma que emplean las calculadoras?

OBSERVE QUE: En los lenguajes de programación, se tiene muy en cuenta los tipos de datos empleados en las operaciones, aunque puede variar, en términos generales si se operan números enteros (sin parte decimal) el resultado será un número entero; pero si se operan números reales (que incluyen parte decimal) o una combinación entre reales y enteros, el resultado será un numero real.

Considerando lo anterior, el resultado de operaciones como la división puede variar, así:

10 / 4 dará como resultado **2**, ya que los dos números son enteros

10.0 / 4.0 dará como resultado **2.5**, ya que los números son reales.

Expresiones Lógicas

Las expresiones lógicas pueden emplear datos de diferentes tipos (lógicos o numéricos), emplean operadores lógicos o relacionales y obtienen como resultado un valor lógico (verdadero o falso).

Operadores relacionales

Estos operadores son los que permiten establecer algún tipo de relación entre dos operandos, las relaciones comunes son las siguientes:

Relación	Operador
Igual	==
Diferente	!=
Mayor	>
Menor	<
Mayor o igual	>=
Menor o igual	<=

Los operandos (datos) que se pueden emplear, en general corresponden a números, aunque algunos lenguajes permiten comparar otro tipo de datos como caracteres o cadenas.

El resultado de una operación que involucra operadores relacionales será de tipo lógico (verdadero o falso)

Ejemplo:

3 > 4	tendrá como resultado Falso
5 < 20	tendrá como resultado verdadero
10 >= 10	Tendrá resultado Verdadero
3 < 4 < 10	no es una expresión válida porque en algún momento quedaría algo como verdadero < 10, lo cual no puede resolverse.

Operadores lógicos

Los operadores lógicos permiten enlazar expresiones lógicas a través de las conexiones lógicas que se emplean comúnmente. Los operadores lógicos comúnmente empleados son:

Conexión lógica	Operador	Símbolo en C#
Conjunción	AND	&&
Disyunción	OR	
Negación	NOT	!

Estos operadores trabajan siguiendo la lógica de las tablas de verdad que se presentan a continuación:

AND	Verdadero	Falso
Verdadero	Verdadero	Falso
Falso	Falso	Falso

OBSERVE QUE: para la conjunción (AND) el único caso en que puede ser verdadero, es que las dos partes de la expresión sean verdaderas, cualquier otra combinación sería falsa.

OR	Verdadero	Falso
Verdadero	Verdadero	Verdadero
Falso	Verdadero	Falso

OBSERVE QUE: para la disyunción (OR) el único caso en que puede ser falso, es que las dos partes de la expresión sean falsas, cualquier otra combinación sería verdadera.

El operador de Negación trabaja sobre un único operando, cambiando su valor de verdad (o valor lógico)

NOT	
Verdadero	Falso
Falso	Verdadero

Ahora bien, regresando al tema de las expresiones lógicas, estas podrían involucrar solamente operadores lógicos, operadores relacionales o combinación de ambos; dependiendo de lo que se quiera presentar.

Ejemplo:

Si se quiere verificar que la variable nota se encuentre en el rango [0.0, 5.0] la expresión correcta sería:

(Nota >= 0.0) && (nota <=5.0)

En este caso, se resuelven primero las expresiones en parentesis, de la siguiente forma.

Si la variable Nota obtiene el valor de 2.6 tendríamos:


(2.6 >= 0.0) && (2.6 <=5.0)
Verdadero && verdadero
Verdadero

Si la variable Nota obtiene el valor de 6.2 tendríamos:

(6.2 >= 0.0) && (6.2 <=5.0)
Verdadero && Falso
Falso

Al combinar operadores lógicos y aritméticos, las prioridades establecidas varían, quedando de la siguiente forma:

/		*	
+		-	
>	<	>=	<=
==		!=	
&&			



Como siempre, los paréntesis permiten saltar los niveles de prioridad.

Ejemplo:

Para el siguiente caso:

$3 + 4 < 10 \ \&\& \ 56 - 3 > 100 \ || \ 4 * 6 - 2 <= 22$

La solución, atendiendo las prioridades establecidas, quedaría de la siguiente forma:

1. Se resuelven divisiones y multiplicaciones:
 $3 + 4 < 10 \ \&\& \ 56 - 3 > 100 \ || \ 24 - 2 <= 22$
2. Se resuelven sumas y restas (de izquierda a derecha):
 $7 < 10 \ \&\& \ 53 > 100 \ || \ 22 <= 22$
3. Se resuelven los operadores relacionales:
Verdadero && Falso || Verdadero
4. Se resuelven las operaciones de conjunción (AND, &&)
Falso || verdadero
5. Se resuelven las operaciones disyunción (OR, ||)
Verdadero

La solución de la expresión es: Verdadero.

Operaciones matemáticas

Algunas operaciones matemáticas comunes, no tienen representación en los lenguajes de programación como un operador, en lugar de ello debe emplearse un método que ya está incorporado en el lenguaje. En el caso de C#, muchas de estas operaciones se incorporan en la clase *Math*, entre las más comunes se encuentran las siguientes:

Nombre del método	Función
Cos(double x)	Retorna el coseno del ángulo especificado.
Exp (double x)	Retorna e elevado a la potencia especificada.
Log (double x)	Devuelve el logaritmo natural (en base e) de un número especificado.
Log10(double x)	Devuelve el logaritmo en base 10 de un número especificado.
Pow(double x, double y)	Retorna un número (x) elevado a la potencia especificada (y)
Round (decimal x)	Redondea un número decimal al entero más próximo.
Sin (double x)	Retorna el seno del ángulo especificado.
Sqrt(double x)	Retorna la raíz cuadrada del número indicado.
Tan(double x)	Retorna la tangente del ángulo especificado.

Para emplear cualquiera de estas funciones se debe anteponer *Math.* al nombre del método correspondiente, esto le especifica al lenguaje que el método que se desea usar esta en una clase llamada *Math*.

Ejemplo:

```
X = Math.sqrt(64);
```

Esta expresión dejaría en x el valor de 8 (raíz cuadrada de 64)

Estos métodos pueden ser involucrados dentro de una expresión.

Ejemplo:

```
58 – Math.sqrt(64) < 100
```

En este caso, se resuelve primero el método, luego la resta y finalmente la comparación; lo que daría como resultado Verdadero

Para pensar....

Algunos símbolos que comunmente empleamos para un propósito tienen un significado totalmente diferente en el lenguaje de programación, por ejemplo, el símbolo de circunflejo (^) corresponde a un operador particular en C# y no es el exponente como comúnmente lo usamos en otros campos.

II. Introducción a la Programación

Un **algoritmo**, se define como “*un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema*”¹. Si se lee con atención la definición, no se mencionan computadores, lenguajes de programación o dispositivos de procesamiento alguno y esto es debido a que el énfasis de un algoritmo no está en la programación de computadores sino en la definición de los pasos o instrucciones, que deben ser claros y completos para permitir alcanzar un objetivo.

Además del proceso definido (pasos), los algoritmos tienen otros dos elementos que los caracterizan: las entradas y la(s) salida(s). Las entradas o elementos de entrada o datos de entrada son aquellas cosas o datos que se requieren para poder realizar el algoritmo a cabalidad. La salida son los elementos o datos que se obtienen una vez se ejecuten los pasos del algoritmo.

La diferencia básica entre un **algoritmo** y un **programa** es que este último está escrito en un lenguaje de programación, de forma que pueda ser interpretado y ejecutado por un computador.

-Escribir un programa, implica necesariamente, hacer un algoritmo-

Cuando vamos a programar, nuestro cerebro ejecuta la misma secuencia de pasos que aplica en cualquier situación problemática: primero, entender el problema con todas sus condiciones y características; segundo, buscar una solución que se ajuste a dichas condiciones y características; tercero, poner en ejecución la solución hallada; cuarto, verificar si la solución cumple con lo que se indicó que debería hacer².

Ejemplo:

Tenemos una situación problema: “Es necesario llegar desde la UAO hasta Chipichape”.

Primero: entendamos el problema, es claro hay que encontrar una forma de llegar desde la UAO hasta Chipichape, pero hay algunas restricciones que tenemos que considerar: ¿hay restricción de dinero para el transporte? ¿hay restricción de tiempo para el transporte? ¿cuántas personas deben transportarse?

Digamos que debemos transportar a 3 personas, no importa cuánto tiempo se demoren pero no deberían gastar más de \$10.000.

¹ Real Academia Española. Diccionario. 2015. Disponible en: <http://www.rae.es/recursos/diccionarios/drae> [consultado: Julio 20 de 2015]

² En desarrollo de software, esta característica se conoce como corrección, un algoritmo correcto, es un algoritmo que efectivamente cumple lo que se supone que debería hacer.

Ahora si, teniendo claro el problema, entramos al paso dos: buscar alternativas de solución.

- **Alternativa 1:** Tomar un taxi. Permite transportar a 3 personas pero con seguridad costaría más de 10.000 pesos. ¡Descartada!
- **Alternativa 2:** Ir en MIO. Permite transportar a 3 personas y cuesta menos de 10.000 pesos. ¡es una alternativa válida!
- **Alternativa 3:** Ir caminando. Permite el transporte de 3 personas y cuesta menos de 10.000 pesos. ¡es una alternativa válida!

Una vez encontramos alternativas válidas podemos seleccionar una, vamos a seleccionar la Alternativa 2, porque aunque la Alternativa 3 es más barata, tardaría mucho tiempo y las personas llegarían muy cansadas.

Una vez seleccionada la alternativa, procedemos al Tercer paso, implementar la solución. En este caso, la implementación consiste en determinar cuáles rutas del MIO se deberán tomar desde la UAO hasta Chipichape y en cuales estaciones se debe hacer cambio de ruta.

Antes de enviar a las tres personas a aventurarse por las calles de Cali, podemos adelantar el paso Cuatro: Probar la solución. Probar es verificar si con las entradas que hemos definido y los pasos que se han establecido logramos el resultado esperado; entonces, se puede hacer una revisión sobre el mapa, para verificar si las rutas seleccionadas con intercambio en las estaciones definidas permiten que las personas lleguen desde la UAO hasta Chipichape.

Una vez se comprueba que la solución diseñada es apropiada, podemos generalizarla, cualquier persona que tenga el mismo problema con las mismas restricciones, podrá utilizar nuestra solución.

Este es un ejemplo bastante simple, pero representa nuestra forma de pensar y actuar ante cualquier situación problema, lo que sucede es que no siempre somos concientes de este proceso, nuevamente, porque lo tenemos tan interiorizado y lo realizamos de forma tan automática que no lo pensamos paso por paso; sin embargo, si lo piensas por un minuto, ante cualquier situación actúas de la misma forma, bueno, con algunas variaciones, porque algunas situaciones requieren que la etapa uno sea de mayor duración que las otras, pero otras situaciones requieren que la etapa dos sea la de mayor duración, etc.

Estas mismas etapas son las que seguiremos en la elaboración de algoritmos, lo que sucede, es que como todo proceso en Ingeniería, tiene sus particularidades en la forma de realizarlo y comunicarlo.

Actividad 1 : El análisis

La primera etapa en el proceso de programación tiene diferentes nombres, en nuestro caso, la denominaremos **Análisis**, en esta etapa se busca comprender la funcionalidad del programa, así como identificar las restricciones que se tengan sobre la operación del mismo.

Otra forma de comprender la etapa de análisis es preguntarse ¿Qué debe hacer el programa? y ¿Qué condiciones se deben cumplir durante la ejecución para lograrlo?

Ahora, ¿De dónde se obtiene esa información?, la respuesta es muy simple, de la persona que necesita el programa. Durante el proceso de aprendizaje, es común que los profesores entreguen por escrito la definición del problema, y de allí básicamente se obtiene la información que se requiere. En situaciones reales, debe preguntarse a la(s) persona(s) que requieren el programa sobre todos los aspectos que este deberá incluir.

Para efectos del curso, en la etapa de análisis se establecerán los datos de entrada y los datos de salida.

- **Datos de entrada:** Son aquellos datos que se requiere conocer para que el programa pueda funcionar apropiadamente. Generalmente se asocian con las “cosas” que se preguntará al usuario.

Un buen programador es conciente que al preguntar un dato, el usuario puede contestar de cualquier forma, por ello, debe asegurar que la información ingresada sea apropiada para el desarrollo del programa, para ello se emplea la validación de datos, por ahora solamente nos centraremos en establecer cuales son los valores válidos para un dato de entrada, más adelante veremos cómo se hace la validación.

- **Datos de salida:** Corresponden a los datos o información que el usuario espera que el programa le presente.

Ejemplo:

En la Universidad de la Region se quiere desarrollar un programa que permita determinar rápidamente si un estudiante ha perdido una asignatura por inasistencia y en cualquier caso, su nota definitiva. Para esto debe tenerse en cuenta que un estudiante pierde una asignatura si ha faltado al 20% o más de las horas programadas en un curso, en ese caso, su nota definitiva será 1.5; de otra forma, su nota se calcula como el promedio de cuatro exámenes que se presentan durante el semestre.

Datos de Entrada: Para el desarrollo del programa, es necesario conocer los siguientes datos:

- Número de horas totales programadas en el curso.
- Número de horas que ha faltado el estudiante
- Calificación del primer examen
- Calificación del segundo examen
- Calificación del tercer examen
- Calificación del cuarto examen

Los dos primeros datos permitirán calcular el porcentaje de faltas que ha tenido el estudiante y de esta manera determinar su nota; si no ha perdido la asignatura por faltas, los otros cuatro datos permitirán hacer el cálculo de la definitiva.

Observe que datos como el porcentaje que determina que el estudiante ha perdido la asignatura por faltas o la nota que se asigna en caso de pérdida por faltas, no se incluyen en los datos de entrada porque ya se conocen (20% y 1.5 respectivamente).

Para efectos posteriores de validar los datos, vamos a incluir el rango de valores válidos para cada dato:

- Número de horas totales programadas en el curso (mayor a 0)
- Número de horas que ha faltado el estudiante (mayor o igual a 0)
- Calificación del primer examen (entre 0.0 y 5.0)
- Calificación del segundo examen (entre 0.0 y 5.0)
- Calificación del tercer examen (entre 0.0 y 5.0)
- Calificación del cuarto examen (entre 0.0 y 5.0)

Datos de Salida: Al finalizar el programa se espera que se presente el siguiente dato:

- Nota definitiva del estudiante

OBSERVE QUE: Aunque durante el proceso (programa) deberá calcularse el porcentaje de inasistencias que ha tenido el estudiante, este dato no será mostrado al usuario, por lo tanto no se considera dato de salida.

Actividad 2: El diseño

El diseño de un programa, es realmente el algoritmo del mismo y como tal, contempla las tres secciones mencionadas: entradas, proceso y salidas. Las entradas y salidas se definieron en el paso anterior, así que ahora nos concentraremos en el proceso.

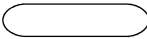


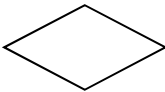


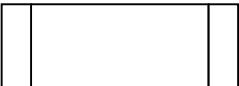

Proceso: Se refiere a todas las instrucciones necesarias para que el computador pueda cumplir el objetivo propuesto para el programa. Este proceso debe describirse en forma


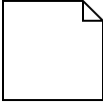
de algoritmo, es decir pasos ordenados, claros y completos que al ser seguidos de forma rigurosa permiten la ejecución del objetivo propuesto.

Existen múltiples formas de representar el proceso de un algoritmo, se puede hacer con lenguaje natural (español o inglés), con lenguaje estructurado (pseudocódigo) o lenguaje gráfico (flujograma). Para nuestro caso se empleará el flujograma, que tiene dos grandes ventajas: es estándar y se gráfico, lo que permite mayor comprensión para los lectores.

FlujoGrama

Durante este curso se empleará la notación de flujograma o diagrama de flujo para presentar el diseño de un programa. El flujograma es una representación gráfica de las instrucciones que se llevarán a cabo en el programa. A continuación se presentan los símbolos a emplear.

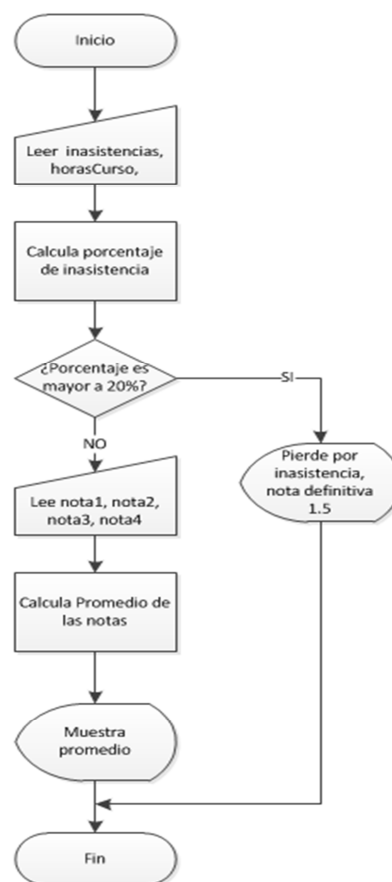
Símbolo	Utilización
	Inicio / Fin Indica los extremos del proceso principal. Dentro del símbolo se deberá escribir la palabra Inicio o Fin. Si este símbolo es utilizado en un subproceso, en el Inicio se deberán escribir las entradas, y en el Fin, se deberá escribir la salida. Solamente deberá aparecer un símbolo de inicio y un símbolo de fin para cada proceso.
	Lectura de Datos Este símbolo se emplea para especificar los datos que se deben leer o ingresar para la realización del algoritmo. Dentro del símbolo se escriben los datos a leer (incluyendo la restricción de los datos, por ejemplo: >0, entre 0 y 20, etc.).
	Proceso Especifica los cálculos requeridos, tales como: evaluación de fórmulas, incrementos, acumulaciones, entre otros. Para cada cálculo se debe especificar la variable donde quedará almacenado el valor.
	Decisiones Este símbolo permite mostrar decisiones que se deben tomar durante la realización del algoritmo, dentro del rombo se escribirá la pregunta o expresión lógica que genera la decisión. Este símbolo se puede emplear para tomar decisiones, en cuyo caso, saldrán del símbolo dos caminos, uno marcado como SI y otro como NO, para indicar las instrucciones que se deben realizar cuando la expresión sea verdadera o falsa, respectivamente.
	Salida Se emplea este símbolo para indicar la información que se va a presentar al usuario.
	Línea de Flujo Indica la secuencia de las instrucciones. Debe tenerse cuidado que las flechas sean rectas (horizontales o verticales) y que no se crucen unas con otras en el diagrama.
	Subproceso Con este símbolo se realiza la invocación o llamado a un subproceso, al interior del cuadro se escribe el nombre del subproceso que se va a invocar. Adicionalmente, se debe especificar los parámetros enviados al subproceso entre paréntesis y la variable en la cual es asignado el valor que éste devuelve.
	Conector Este símbolo permite conectar varias flechas en la misma página.

Símbolo	Utilización
	Conector de página Si el diseño de un algoritmo ocupa varias páginas, se puede emplear este conector para indicar la secuencia de instrucciones, si existen varios conectores, es conveniente numerarlos para evitar alguna confusión.
	Comentario Este símbolo permite incluir comentarios o notas adicionales sobre el algoritmo. Se sugiere utilizar este símbolo (junto al símbolo de Inicio) para incluir el nombre de los procesos.

Ejemplo:

Continuando con nuestro ejemplo y considerando los datos de entrada y salida definidos, pasaremos a desarrollar el diseño del mismo.

A manera de ejemplo, se presenta el siguiente diagrama como una propuesta de solución³:



³ Los flujogramas de ejemplo que se presentan en este documento, así como los que se presentarán posteriormente durante el curso, han sido realizados empleando la herramienta Microsoft Visio 2010. Los estudiantes son libres de emplear esta u otra herramienta e incluso hacer sus diagramas a mano, siempre y cuando respeten los símbolos definidos.

NOTA: como todo proceso de diseño en Ingeniería, pueden presentarse diferentes propuestas para la solución de un problema, sin embargo, debe asegurarse que todos los pasos se cumplan y la salida sea la esperada.

Actividad 3: La implementación

El siguiente paso es realizar la implementación del algoritmo, es decir, traducir las instrucciones definidas a un lenguaje de programación. Si bien cada lenguaje tiene una sintaxis diferente, todos manejan los mismos elementos conceptuales, los cuales si son claros para el programador, le permitirán emplear cualquier lenguaje con un mínimo tiempo de aprendizaje.

Para el desarrollo de nuestro curso, se empleará el lenguaje de programación Visual C#, por lo cual, se presentará la sintaxis específica de este lenguaje al tiempo que se presentan los conceptos generales.

Los elementos básicos de programación se describen a continuación:

1. **Variables**

Las variables en programación son equivalentes a las variables que se emplean en matemáticas, son palabras (a veces de una sola letra) que representan valores que pueden variar a lo largo del programa. Las variables en programación tienen dos características propias: nombre y el tipo.

El nombre de la variable, conocido también como identificador, es el que permite hacer referencia a la variable en cualquier parte del programa. Cada lenguaje define algunos limitantes para la selección del nombre de las variables, sin embargo, algunas normas generales de buena programación son las siguientes:

- Una variable debe nombrarse con una o varias palabras que indiquen el dato que contiene, *por ejemplo: nombre, telefono, salario, temperatura, etc.*
- El nombre de una variable no puede incluir espacios ni caracteres especiales. Si el nombre se compone de varias palabras, pueden enlazarse con subrayado o simplemente poner las palabras seguidas una de otra, *por ejemplo: salario_total, temperaturaMedia, promedio_Semestral.*

- El nombre de la variable debe empezar por una letra, posteriormente puede incluir numeros. Si el nombre de una variable inicia con un número, el computador puede entender que se trata de un número y no de una variable. *Por ejemplo, semestre5, login123, etc.*

El tipo de la variable, se refiere al tipo de dato que la variable podrá almacenar. La mayoría de los lenguajes requieren que se establezca el tipo de la variable y que no se cambie durante toda la ejecución del programa. Adicionalmente, cada lenguaje tiene limitaciones propias para cada tipo de datos (como el rango de datos válidos). Los tipos de datos más generales son los siguientes:

- Entero: indica que las variables guardarán números negativos, positivos o cero SIN parte decimal. En C# existen varios tipos enteros: sbyte, short, int, long; la diferencia entre un tipo y otro es el tamaño disponible para almacenar datos. El tipo de dato entero más común es el **int**, que tiene un tamaño de 4 bytes y permite almacenar números entre -2.147'483.648 y 2.147'483.647
- Reales o decimales: indica que las variables guardarán números negativos, positivos o cero incluyendo parte decimal. En C# existen dos tipo de datos reales: float y double. Al igual que con los enteros, la diferente entre ellos se basa en el tamaño disponible. El tipo más comun en los reales es el **float** que tiene 7 dígitos de precisión.
- Booleano: es un tipo de dato especial que permite almacenar valores de verdad (verdadero o falso). En C# se emplea el tipo **bool** y los únicos valores que puede tomar son True o False.
- Caracteres: indica que la variable almacenará caracteres únicos, que pueden ser letras ('a','F','z'), números como símbolos y no como cantidad ('3', '9', '0') o caracteres especiales, que corresponden a esos caracteres que se disponen y que no son letras ni números (')', '#', '['). Cuando se emplea un número como símbolo (carácter) no es posible hacer operaciones aritméticas con él. En C#, este tipo se denomina **char** y sus valores, siempre se presentan entre comillas simples.

- Cadenas: algunos lenguajes disponene un tipo de dato especial, denominado cadena, que corresponde a un conjunto de caracteres, por ejemplo “perro”, “mi casa es bonita”, “Maria Martinez”, “ el promedio es 3.45”. En C# este tipo de dato, se denomina **string** y los datos que almacena siempre se representan entre comillas dobles.

2. Constantes

Las constantes se refieren a datos que no cambiarán su valor durante la ejecución del programa, no requieren nombre y solamente se trabajan con el dato correspondiente (3, 6.7, “hola”).

3. Operador de asignación

La definición de una variable supone que en alguna parte del proceso establecido, la variable debe tomar un valor o cambiar el que tiene, una forma de realizar esta operación es asignandole dicho valor, para ello se emplea un operador específico. El operador de asignación empleado por cada lenguaje puede diferir pero el más común es el operador =.

A través del operador de asignación se asigna el valor a la derecha del operador (que podría ser una variable o constante) a la variable que se encuentra a la izquierda del operador.

Ejemplo:

Total = 35.6	asigna el valor 35.6 a la variable total
Nombre = “Miguel”	asigna la cadena Miguel a la variable nombre.
Promedio = nota	asigna a la variable promedio el valor que tenga (en ese momento) la variable nota.
3 = x	es una expresión no válida, porque no se puede asignar un valor a otro valor.

4. Expresiones

Como se explicó en la unidad anterior, las expresiones son sentencias que toman un valor particular; en programación se distinguen las expresiones aritméticas, que dan como resultado un valor numérico y las expresiones lógicas que dan como resultado un valor de verdad (dentro de las expresiones lógicas se incluyen las relacionales).

5. Operaciones

El último elemento constituyente de un programa de computadora son efectivamente las operaciones o instrucciones que se pueden emplear.

Operaciones de lectura: son las operaciones con las que se permite al usuario hacer el ingreso de los datos para que el programa funcione.

Operaciones de escritura: son las operaciones que permiten presentar información en pantalla.

Decisiones: son operaciones que permiten definir un conjunto de instrucciones que se ejecutarán bajo una condición específica.

Repeticiones o ciclos: Permiten definir un conjunto de instrucciones que se repetirán mientras se cumpla una condición establecida.

6. Método

Un método o procedimiento se puede entender como una porción de programa que realiza una tarea específica. Un método puede llamarse o invocarse en cualquier momento dentro de un programa, en ese momento se ejecutan las instrucciones del método y una vez termine se regresa a continuar la ejecución del programa.

Al realizar el llamado o invocación al método, es posible enviar datos que este requiera para realizar su operación, de forma similar, al terminar la ejecución del método, se puede retornar un dato al programa principal para que continúe su labor.

NOTA: Aunque pueden existir variaciones, en general, los métodos (procedimientos, rutinas o funciones) en todos los lenguajes de programación tienen la posibilidad de retornar un solo dato.

Actividad 4: Las pruebas

La última actividad del proceso de desarrollo de software es la de pruebas, en esta parte se ejecuta el programa bajo condiciones controladas para verificar su corrección, es decir, verificar que cumple los requerimientos definidos previamente.

Contrario a lo que se puede pensar, las pruebas no están diseñadas para probar que el programa funciona bien, por el contrario, las pruebas se diseñan intentando que el

programa falle. Si al ejecutar las pruebas, el programa no falla, se puede asegurar que bajo esas condiciones, el programa funciona bien, pero no se puede asegurar que el programa funcionará bien bajo todas las circunstancias.

Cada prueba incluye:

- Una descripción de la característica o funcionalidad que se desea probar.
- Los datos y valores que se deben emplear como entrada al programa.
- Los datos de salida que se deberían obtener.
- Para aplicaciones más complejas, deben indicarse las condiciones del sistema al momento de la prueba.

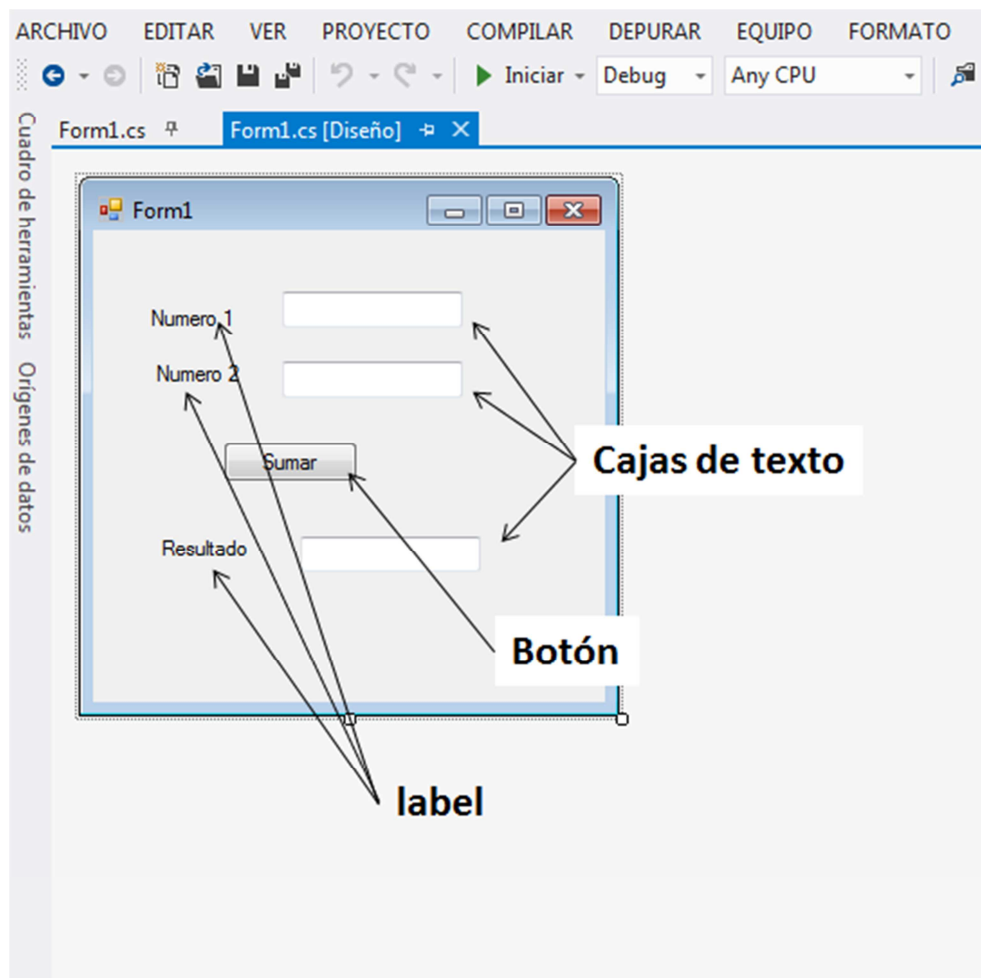
Cada prueba se ejecuta verificando si la salida obtenida es la esperada, si es así, se indica que la prueba es aprobada, de no ser así, debe empezar un proceso de revisión del programa para encontrar la causa del error y corregirla, una vez realizado este proceso se ejecutan nuevamente todas las pruebas para verificar que se corrigió efectivamente el error y que no se insertó otro error en el proceso.

III. INTRODUCCIÓN AL LENGUAJE VISUAL C#

Visual C# es un lenguaje gráfico orientado a eventos, lo que significa que las acciones se ejecutan en respuesta a eventos.

Un programa desarrollado en un lenguaje visual, tendrá dos componentes: la interfaz gráfica (conocida como GUI por sus siglas en inglés – *Graphic User Interface*) y el código o instrucciones del programa.

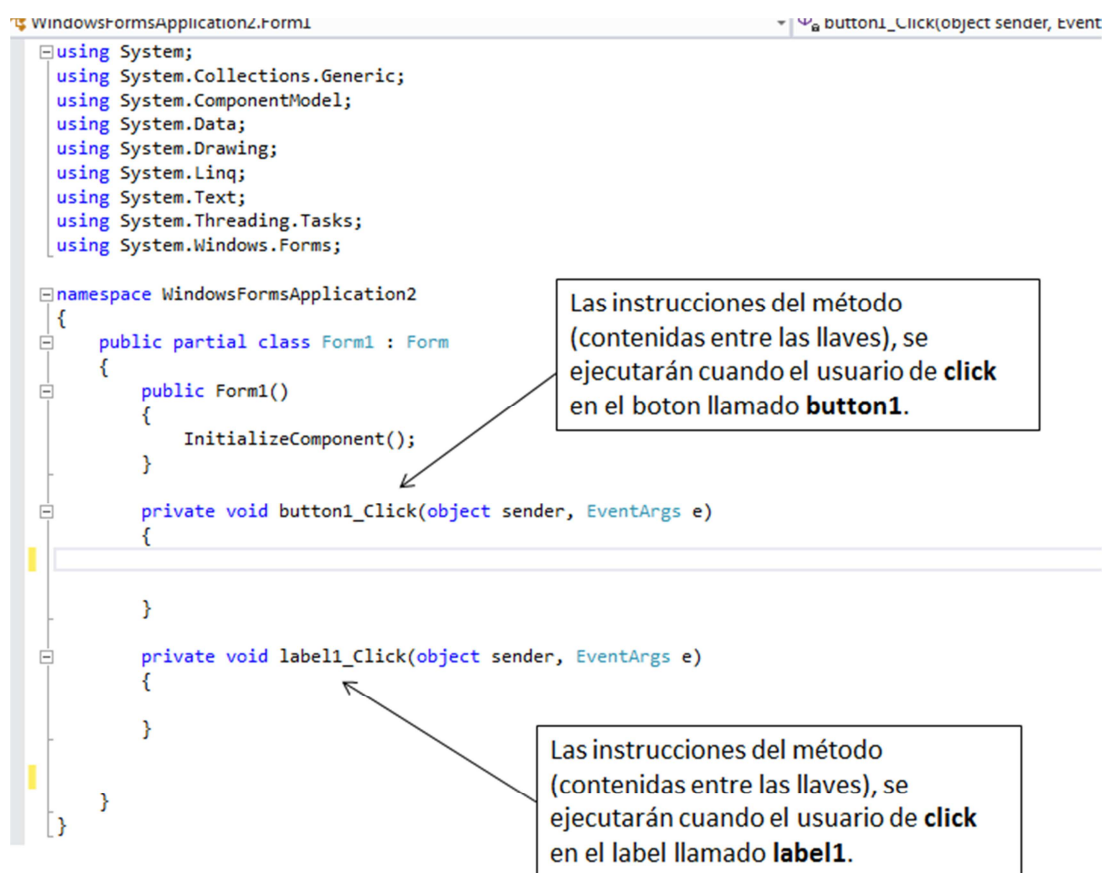
Inicialmente se desarrollará la GUI, agregando esencialmente: *labels* (etiquetas), *cajas de texto* (para leer o mostrar datos) y *botones*.



El código del programa aparece en otra ventana y tiene una estructura similar a la siguiente: inicialmente se encuentra una sección de varias instrucciones **using**, esta sección

le indica al lenguaje cuales librerías⁴ se van a emplear en el programa; posteriormente aparece una definición del **namespace** que corresponde a la aplicación (si observa, este nombre es el mismo que usted dio a su programa al crear el proyecto); luego continúa la definición de clase, que corresponde a la estructura empleada por el lenguaje y en la cual por ahora no ahondaremos.

Lo que sí es importante, es que dentro de la clase se ubican diferentes métodos que responden a los eventos de los elementos incorporados en la GUI. En general, serán de la misma forma: tienen el nombre del elemento (button1, label1, etc.) y la acción a la que responden (click).

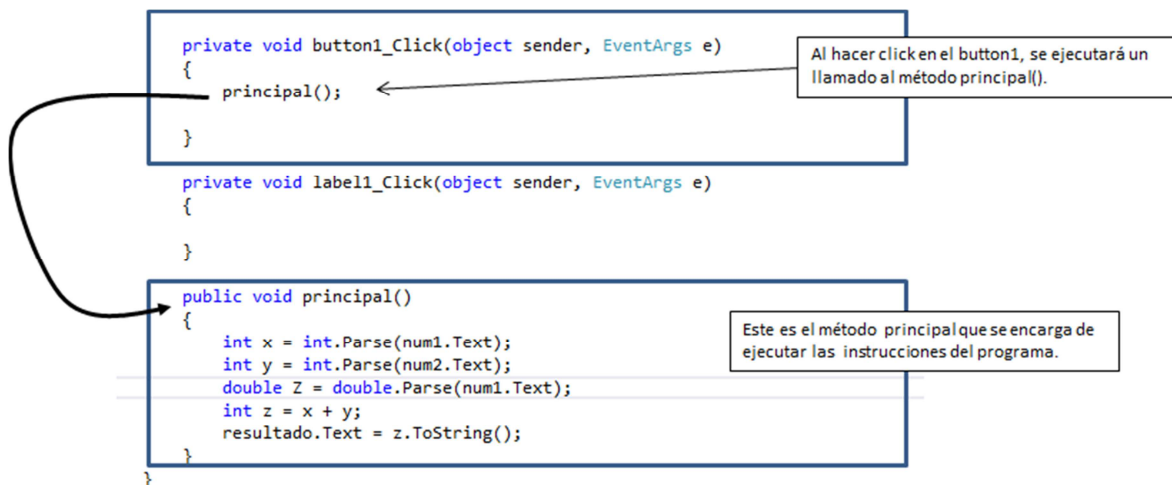


Estructura General:

Como convención en nuestro curso, la estructura general de los programas es que hay un botón que inicia la ejecución del programa (botón de calcular, ejecutar, etc.) y dentro del

⁴ Una librería es un conjunto de módulos o programas que ya se encuentran escritos en el lenguaje de programación y pueden ser empleados por los programadores cuando lo requieran.

evento correspondiente se realizará un llamado al método principal(), en el cual se ubicarán las instrucciones necesarias:



NOTA: En programas complejos, el método o módulo principal se encarga de operaciones de lectura y escritura y delega las instrucciones de procesamiento a otro(s) método(s).

Operaciones:

Lectura de variables de entrada desde una GUI.

Cuando se trabaja con cajas de texto, debe tenerse en cuenta que son eso: cajas **de texto** y por tanto, lo que se ingresa o se muestra en ellas es texto (o cadenas). Para trabajar con los datos que se han ingresado, deben transformarse al tipo requerido, de la siguiente forma.

Suponiendo que la caja de texto que contiene el dato se llama Control:

Lectura de cadenas	<i>String variable = Control.Text;</i> Ej: <i>String nombre = textBoxNombre.Text;</i>
Lectura de enteros	<i>int variable = int.Parse(Control.Text);</i> Ej: <i>int edad = int.Parse(textBoxEdad.Text);</i>
Lectura de reales	<i>double variable = double.Parse(Control.Text);</i> Ej: <i>double estatura = double.Parse(textBoxEstatura.Text)</i>
Lectura de caracteres	<i>char variable = char.Parse(Control.Text);</i> Ej: <i>char seguir = char.Parse(textBoxEstadoCivil.Text);</i>

NOTA: Recuerde que todas las variables que se utilicen, deben ser declaradas⁵ previamente.

⁵ La declaración de una variable implica darle un nombre e indicar el tipo de dato que va a almacenar.

Escritura (salida) de variables de entrada desde una GUI.

Para escribir o mostrar datos en la pantalla, generalmente usaremos cajas de texto deshabilitadas (Enabled=false), para evitar que el usuario ingrese datos en ellas.

Suponiendo que la caja de texto que va a contener el dato se llama Control:

Mostrar una cadena fija o constante	Control.Text = "Constante de cadena"; Ej: textBoxSaludo.Text = "Hello World";
Mostrar una variable de tipo cadena	Control.Text = <i>variable</i> ; Ej: textBoxNombre.Text = <i>nombre</i> ;
Mostrar una variable de tipo diferente a cadena (debe transformarse a texto)	Control.Text = <i>variable</i> .ToString() ; Ej: textBoxNota.Text = nota.ToString() ;

Algunas operaciones básicas con la GUI.

Hay dos operaciones de uso común con las interfaces gráficas. La primera permite cerrar la ventana o pantalla que se está ejecutando y la segunda permita limpiar los datos contenidos en una caja de texto.

Cerrar GUI		Close();
Limpiar componentes gráficos	Cajas de texto	Control.Clear(); Ej: textBoxNombre.Clear() ;

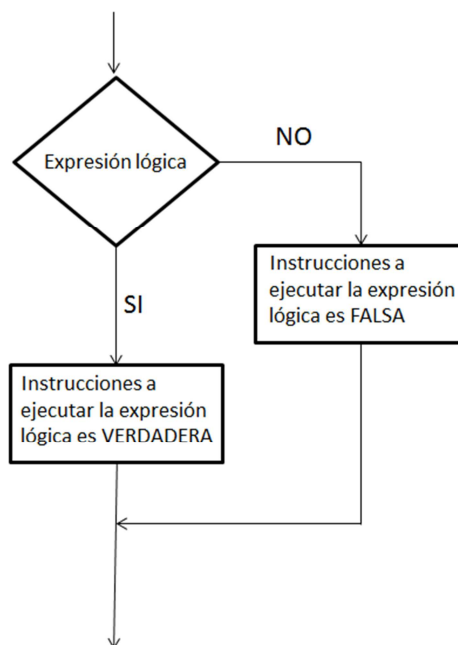
IV. PROGRAMACIÓN ESTRUCTURADA

En programación estructurada se emplean tres tipos de estructuras: las de **secuencia**, que implica escribir una instrucción después de otra; las de **decisión** que permiten representar las situaciones donde hay que decidir por un camino u otro y las de **repeticón** que permiten ejecutar un conjunto de instrucciones varias veces.

1. ESTRUCTURAS DE DECISIÓN

Una estructura de decisión se emplea cuando en el desarrollo de un programa debe tomarse una decisión que implica realizar una(s) instrucción(es) cuando la condición sea verdadera y otra(s) cuando la condición sea falsa.

En el diagrama de flujo, se representa la decisión con un rombo, de forma que por un lado (en general a la derecha) se presentan las instrucciones que se realizarán cuando la expresión sea Falsa y por abajo se presentan las instrucciones que se realizarán cuando la expresión sea Verdadera.



OBSERVACIONES: Es importante considerar los siguientes aspectos:

- El contenido del rombo debe ser una expresión lógica, que tenga como resultado Falso o Verdadero.
- Del rombo, salen solamente dos caminos; cada uno de los cuales debe estar marcado con SI o NO.
- En cada lado del rombo se pueden incluir todo tipo de instrucciones (lectura, proceso, salida; incluso otras decisiones).

- En algún punto y en todo caso, antes de terminar el programa, deben unirse nuevamente los dos caminos que se originaron en la decisión.

En **C#**, una estructura de decisión se define con la palabra clave **if**. La estructura general de la instrucción es la siguiente:

```

if ( condición )
{
    // instrucciones que se ejecutan si la condición es verdadera.
}
else
{
    // instrucciones que se ejecutan si la condición es falsa.
}

```

Algunas recomendaciones para tener presentes:

- La condición debe estar escrita entre paréntesis y debe corresponder a una expresión lógica (cuyo resultado sea Verdadero o Falso).
- Una vez se abre una llave, todas las instrucciones que se escriban se consideran dentro del mismo bloque (verdadero o falso) hasta que se cierre la llave respectiva.
- La palabra clave **else** y el bloque respectivo, no son obligatorios. Ya que una decisión puede no tener instrucciones para realizar cuando la condición sea Falsa.
- La palabra clave **else** (si se incluye) debe escribirse inmediatamente después de la llave que cierra el bloque de instrucciones verdadero.

A continuación se presentan algunos ejemplos de utilización de la estructura de decisión.

1. *Calcular la longitud de una circunferencia si el radio es positivo.*

```

if ( radio > 0 ) {
    longitud = 2 * 3.1416 * radio ;
}

```

2. *Dada la edad de una persona, imprimir un mensaje sólo si ella cumple con la mayoría de edad.*

```

if ( edad >= 18 ) {
    salida.Text = "La persona tiene la mayoría de edad" ;
}

```


3. *Calcular la longitud de una circunferencia si el radio es positivo y, en caso contrario, se proporciona un mensaje de error.*

```
if ( radio > 0 ) {  
    longitud = 2 * 3.1416 * radio ;  
}  
else {      // si radio <= 0  
    salida.Text = "Radio incorrecto" ;  
}
```

4. *Dada la edad de una persona, imprimir un mensaje indicando si es mayor de edad o no.*

```
if ( edad >= 18 ) {  
    salida.Text = "La persona tiene la mayoría de edad" ;  
}  
else {  
    salida.Text = "La persona no tiene la mayoría de edad" ;  
}
```

Estructura de Decisión Múltiple

C#, como otros lenguajes, ha definido lo que se denomina una estructura de decisión múltiple, que permite evaluar los diferentes valores que puede tomar una variable (o expresión) y de acuerdo a ello, realizar diferentes operaciones.

NOTA: Toda operación que se pueda realizar empleando estructuras de decisión múltiple, igualmente puede presentarse con estructuras de decisión básicas, sin embargo la estructura de decisión múltiple es más eficiente en escritura y ejecución.

En C#, la instrucción **de decisión múltiple** se presenta así:

```
switch( variable o expresión a evaluar ) {  
    case valor1:  
        // Bloque de instrucciones que se ejecuta si la expresión  
        // toma el valor valor1  
        break ;  
    case valor2:  
        // Bloque de instrucciones que se ejecuta si la expresión  
        // toma el valor valor2  
        break;  
    // .....  
    case valorN:  
        // Bloque de instrucciones que se ejecuta si la expresión  
        // toma el valor valorN  
        break;  
    default:  
        // Bloque de instrucciones que se ejecuta si la expresión  
        // NO toma ninguno de los valores arriba indicados.  
} // Llave de fin de la estructura de decisión múltiple
```

Algunas recomendaciones para tener presentes:

- La expresión o variable que se incluye en el paréntesis, debe ser del mismo tipo de cada uno de los valores que se incluyen en los diferentes casos.
- El uso de la palabra clave **break**, al final de cada caso es muy apropiado. Esta sentencia, rompe el proceso de ejecución y lo traslada después de la estructura; es decir, cuando se encuentre la palabra break, el programa continúa ejecutando las instrucciones que hay después de la llave que cierra la sentencia switch.
- Los casos no requieren ningún orden específico.
- Solamente se puede emplear esta estructura en casos de igualdad, es decir cuando la variable o expresión en el paréntesis sea igual al valor1, o igual al valor2, etc. Para casos de rangos (mayor que, menor que, etc.) no se podría emplear esta estructura.
- La sentencia **default** no es obligatoria, aunque es bastante útil para saber cuándo se ha obtenido un valor que no se consideró en la lista de casos definidos.
- El último caso incluido (sea default u otro), no requiere la sentencia break, ya que en ese punto termina la estructura de decisión.
- Observe que las únicas llaves incluidas son las que abren y cierran la sentencia switch. En cada uno de los casos no se requiere el uso de llaves.

Ejemplo:

Si *mes* es el cardinal de un mes del año (1 para enero, 2 para febrero, etc.), indique el número de días que tiene dicho mes.

```
switch ( mes ) {  
    case 1: salida.Text= "31" ;  
           break;  
  
    case 2: salida.Text = "28 días, 29 días si el año es bisiesto" ;  
           break;  
  
    case 3: salida.Text = "31" ;  
           break;  
  
    case 4: salida.Text = "30" ;  
           break;  
  
    case 5: salida.Text = "31" ;  
           break;  
  
    case 6: salida.Text = "30" ;  
           break;  
  
    case 7: salida.Text = "31" ;  
           break;  
  
    case 8: salida.Text = "31" ;  
           break;  
  
    case 9: salida.Text = "30" ;  
           break;  
  
    case 10: salida.Text = "31" ;  
            break;  
  
    case 11: salida.Text = "30" ;  
            break;  
  
    case 12: salida.Text = "31" ;  
            break;  
  
    default:  
        salida.Text = "Mes incorrecto" ;  
}
```

2. ESTRUCTURAS DE REPETICIÓN

Otra de las estructuras usadas habitualmente en programación corresponde a las **estructuras de repetición** o **ciclos**: son aquellas estructuras que permiten ejecutar una serie de instrucciones en varias ocasiones. En todos los lenguajes de programación existen instrucciones precisas que permiten construir este tipo de estructuras.

¿Cuándo hay un proceso de repetición?

La respuesta a esta pregunta es bastante fácil, cuando se requiere desarrollar el mismo conjunto de instrucciones (1, 2 o más instrucciones) más de una vez.

Por Ejemplo:

- Se requiere calcular la nota definitiva de 20 estudiantes de una asignatura.
- Se requiere calcular el valor del paquete turístico para todos los clientes que compren en un día.
- Se requiere que el programa permita al jugador realizar tantos intentos como desee, para pasar de un nivel a otro.
- Si el usuario digita 3 veces la clave incorrecta, debe bloquearse la cuenta.

La estructura

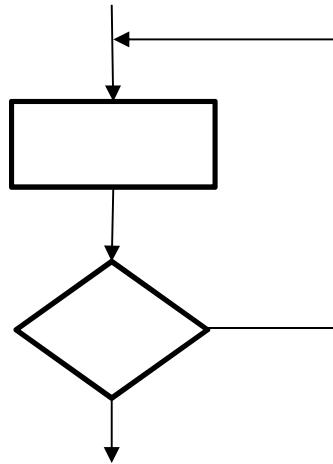
Una estructura de repetición o ciclo tiene tres elementos constituyentes:

- 1- **CONDICIÓN:** la repetición siempre es controlada por una condición. En algunos casos, se repite cuando la condición sea Verdadera y se termina el proceso cuando la condición es Falsa. En otros casos, se repite cuando la condición sea Falsa y se termina el proceso cuando la condición es Verdadera. Pero independiente de esto, se debe entender que la condición será la que controle el ciclo de repetición.
- 2- **INSTRUCCIONES:** Se entiende como el conjunto de instrucciones que se desea repetir, estas pueden ser de cualquier tipo: lecturas, escrituras, procesos de cálculo, llamado a métodos, estructuras de decisión e, incluso, otras estructuras de repetición.
- 3- **CAMBIO DE LA CONDICIÓN:** Para asegurar que las repeticiones terminen en algún momento y no se convierta en un ciclo infinito, debe asegurarse que en alguna instrucción, dentro del ciclo, se haga el cambio de una o varias variables que se encuentran involucradas en la condición que controla el ciclo; este cambio en la variable, puede deberse a que se lee nuevamente o simplemente por la asignación de un nuevo valor.

En Diseño (Flujograma)

Los ciclos en el flujograma se identifican fácilmente porque incluyen una condición y una flecha *dirigida hacia arriba*, que implica el reprocesamiento de algunas operaciones. Como se dijo anteriormente, la flecha *hacia arriba* puede corresponder al camino negativo de la decisión o por el camino positivo, dependiendo de la forma de la estructura.

Por ejemplo:

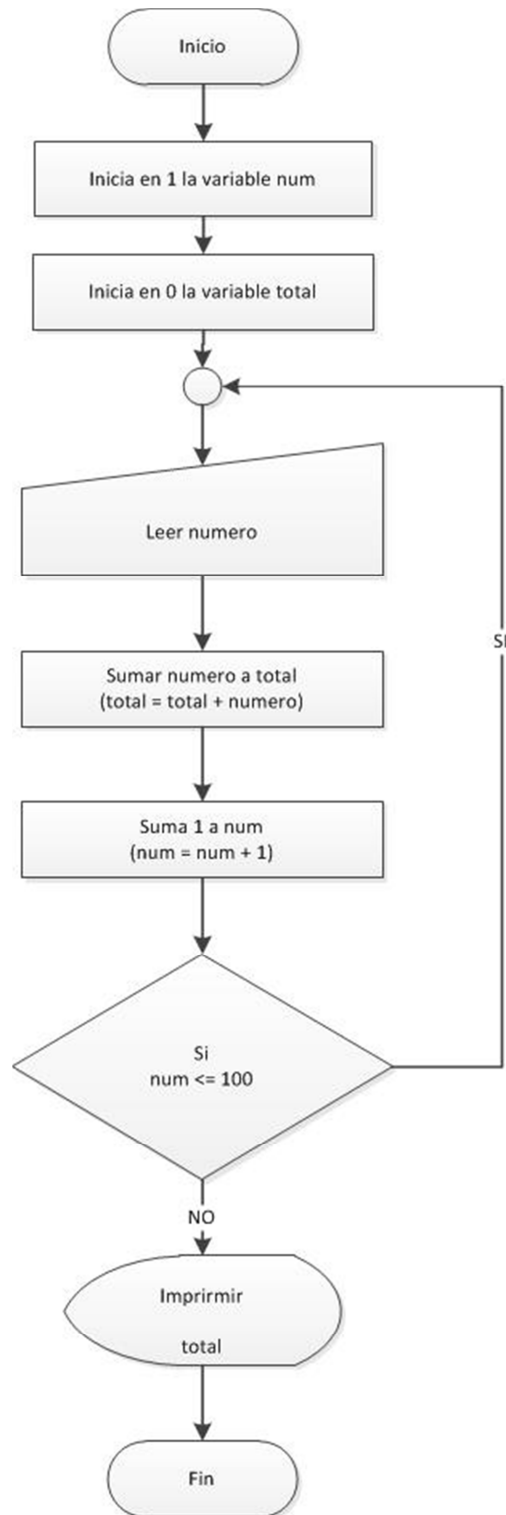


Ejemplo: Se requiere un programa que permita leer 100 números y obtener la suma total de ellos.

Para este caso los Datos de Entrada son los 100 números y el Dato de Salida es la suma de ellos.

NOTA: *hay varias formas de realizar este algoritmo, una que es dispendiosa pero válida, es leer 100 variables diferentes y sumarlas; pero como no se quiere hacer un trabajo tan arduo, lo que se puede hacer es leer un número, sumarlo sobre una variable y repetir las operaciones 100 veces.*

Para lograr las 100 repeticiones, emplearemos algo que todos usamos a diario: contar. Una variable nos ayudará a contar desde 1 hasta 100. Cuando termine, sabremos que se han completado 100 veces.



La variable **num** es empleada para contar desde 1 hasta 100. Observe que antes del ciclo la variable es iniciada en 1 y cada vez que se ejecuta el ciclo se le suma 1 (incrementa) y al final, se pregunta si es menor o igual a 100. Si la variable todavía es menor o igual a 100, se repite el ciclo, sino (cuando sea 101) se terminará el ciclo.

La variable **total** es empleada para guardar (sumar o acumular) los números que se van leyendo. Observe que antes del ciclo, la variable se inicia en 0 (para asegurar su valor inicial) y en cada pasada (iteración) del ciclo, se le suma el número leído. Una vez se termina el ciclo, esta variable se imprime.

NOTA: Para facilitar la transcripción al lenguaje (Visual C#), el ciclo se ha realizado de forma que se repita cuando la condición sea Verdadera y termine cuando la condición sea Falsa.

En Visual C#

En el lenguaje de programación Visual C# se han definido tres formas diferentes para escribir estructuras de repetición o ciclos. Aunque las tres formas son válidas y útiles para cualquier caso de repetición, hay unas formas que son más pertinentes ante ciertas situaciones que otras. La estructura de repetición establecida en el flujograma puede “traducirse” con cualquiera de las formas del lenguaje de programación.

Forma 1: Ciclo FOR

La sintaxis para un ciclo for, es la siguiente:

```
for (instrucciones iniciales; condición; instrucciones finales) {  
    // instrucciones a repetir  
}
```

La estructura incluye tres bloques dentro del paréntesis, cada uno de los cuales se separa con punto y coma (;):

- Instrucciones iniciales: son las primeras instrucciones que se ejecutarán en el ciclo. Generalmente se emplean para dar valor inicial a las variables. Pueden incluir varias instrucciones, en cuyo caso se separan con coma (,).
- Condición: Debe ser una expresión lógica, cuya solución sea Verdadero o Falso. En este caso, el ciclo se repetirá cuando la condición sea VERDADERA. En el momento que la condición sea Falsa, el ciclo terminará y continuará ejecutando las instrucciones que siguen después del bloque de repetición.
- Instrucciones finales: estas instrucciones se ejecutarán al final del ciclo (después de las instrucciones a repetir). Nuevamente, pueden ser varias instrucciones separadas por coma (,). Aunque se pueden incluir diferente tipo de instrucciones, generalmente esta sección se utiliza para modificar el valor de las variables que se incluyen en la condición.
- Instrucciones a repetir: se refiere a las instrucciones que conforman efectivamente *el cuerpo del ciclo*; es decir, las instrucciones que se requiere realizar múltiples veces y que dieron origen a la estructura de repetición. Como se mencionó, en este bloque se pueden incluir cualquier tipo de instrucciones: lectura de datos, cálculo y asignación de valores, escritura de datos, llamado o invocación de métodos, estructuras de decisión u otras estructuras de repetición.

OBSERVAR QUE: Cuando se emplee una estructura *for* debe tenerse cuidado con los siguientes aspectos:

- El bloque de instrucciones a repetir, tal como todos los bloques en Visual C#, debe estar entre llaves {...}
- En el encabezado de la estructura, se incluye entre paréntesis tres secciones, separadas por punto y coma (;). Las tres secciones deben mantenerse, aunque no tengan instrucciones.

Utilización: La estructura *for* es empleada comúnmente para los casos en los que se conoce previamente el número de repeticiones que se van a ejecutar.

Por ejemplo:

- Ejemplo 1: Se requiere un programa que lea y sume 100 números enteros.

```
int numero, suma = 0;
for (int i = 1; i <= 100; i=i+1) {
    numero = int.Parse( Interaction.InputBox("Digite el número", "Lectura de datos", null, 100, 100) );
    suma = suma + numero;
}
salida.Text = suma.ToString();
```

La instrucción ***Interaction.InputBox (String, String, String, int,int)*** permite que se lean datos a través de una ventana emergente, pero requiere que se incluya la instrucción ***using Microsoft.VisualBasic;*** en la parte inicial del programa y se incluya ***VisualBasic*** en ***References*** de la aplicación.

En este caso, la variable *i* es una variable que permite controlar el ciclo, como se puede observar en la primer parte del ciclo (instrucciones iniciales) se declara la variable y se inicia en 1; en la parte final (instrucciones finales) se incrementa el valor de la variable en 1, es decir, si tiene como valor 1 cambia a 2, si tiene como valor 2 pasa a 3 y así sucesivamente. La condición es que la variable sea menor o igual a 100. Recuerde que en este tipo de ciclos, se repiten las instrucciones cuando la condición sea verdadera (mientras la condición sea verdadera). En este caso, dado que la variable se inició en 1 y va incrementando de uno en uno, el ciclo se repetirá cuando *i* tenga los valores de 1, 2, 3, ..., 100. En el momento que la variable *i* tome el valor 101, la condición se torna falsa y por tanto, el ciclo termina.

- Ejemplo 2: Se requiere un programa que lea 10 números enteros y al final indique cuántos fueron positivos, cuantos negativos y cuántos cero.

```
int positivos=0;
int negativos = 0;
int ceros = 0;
int numero;
String numerotexto;

for (int num=0; num<10; num = num +1) {
    numerotexto= Interaction.InputBox("Digite el número", "Lectura de datos", null, 100, 100);
    numero = int.Parse(numerotexto);
    if (numero >0 ) {
        positivos = positivos +1;
    }
}
```



```

else {
    if (numero < 0) {
        negativos = negativos +1;
    }
    else { // si no es positivo ni negativo, debe ser cero.
        ceros = ceros +1;
    } //else
} // else
} //for

```

La instrucción **Interaction.InputBox(String, String, String, int, int)** permite que se lean datos a través de una ventana emergente, pero requiere que se incluya la instrucción **using Microsoft.VisualBasic;** en la parte inicial del programa y se incluya **VisualBasic** en **References** de la aplicación.

Observe que en este caso se tienen 4 variables que, bajo ciertas circunstancias, van aumentando de 1 en 1. A este tipo de variables se les conoce como contadores, porque su valor va incrementando cada vez en un valor constante, en este caso 1. La variable *num* incrementa en cada ejecución del ciclo y, por ello, se dice que va contando la cantidad de veces que se ha repetido el ciclo. De forma análoga, la variable *positivos* cuenta la cantidad de números positivos (solamente incrementa cuando el número es mayor a 0), la variable *negativos* cuenta la cantidad de números negativos (solamente incrementa cuando el número es menor a 0) y la variable *ceros* cuenta la cantidad de ceros que se digiten (solamente incrementa cuando el número no es mayor a cero y no es menor a cero).

NOTA: En este caso, el contador de los números (*num*) se inicia en 0. Por esto, la condición es <10 y no ≤ 10 , de forma que el ciclo se repetirá cuando *num* tenga valores de 0,1,2,3,4,5,6,7,8 y 9 (10 veces en total).

- Ejemplo 3: Se requiere un programa que lea las edades de 150 estudiantes de primer semestre de una carrera y calcule la edad promedio de dichos estudiantes.

```

...
int edad;
int edadTotal = 0;
double edadPromedio;
for (int cont =0; cont < 150; cont = cont + 1) {
    edad = int.Parse(Interaction.InputBox("Digite la edad", "Lectura de datos", null, 100, 100));
    edadTotal = edadTotal + edad;
}
edadPromedio = edadTotal / 150.0;
...

```

Aquí se requiere hacer una sumatoria de todas las edades, para luego dividir las sobre el total de personas (150) y, de esa forma, obtener el promedio de las edades. Este programa ha incorporado un contador (estudiante) que ayuda a controlar la cantidad de veces que se va a repetir, en este caso, cuántas edades sea va a leer.

Además, se utiliza otra variable `edadtotal`, que va sumando los valores de las edades leídas. Esta variable inicia en 0. Si la primera edad leída es 20, `edadtotal` tomará el valor de 20; si la segunda edad leída es 15, `edadtotal` tomará valor de 35 (20+15); si la tercera edad leída es 18, `edadtotal` tomará el valor de 53 (35 + 18) y así sucesivamente. Las variables que tienen esta función, se conocen como *acumuladores*, ya que su propósito es *acumular* o hacer la sumatoria de un dato particular y su valor se incrementa en una cantidad variable (en este caso, la edad).

Forma 2: Ciclo WHILE

La sintaxis del ciclo `while` es la siguiente:

```
while (condición) {  
    // Instrucciones a repetir  
}
```

Al igual que en la estructura anterior, la estructura `while` se repite cuando (*mientras*) la condición es verdadera. En el momento que el valor de la condición cambia a falso, se termina el ciclo y el programa continúa ejecutando las instrucciones que se encuentren después de la estructura.

En un ciclo `while`, no se define un espacio *explícito* para las instrucciones que inician las variables involucradas en el ciclo (iniciación), ni para hacer el cambio de valor de dichas variables (finalización). No obstante, deben incluirse este tipo de instrucciones, para asegurar el correcto desarrollo del ciclo.

Utilización: Este tipo de ciclos resulta muy útil para los casos en que se conoce el número de repeticiones que se va a realizar, así como en los casos en los que no se conoce con antelación dicho número.

OBSERVACIÓN: Por regla general, los ciclos de repetición se pueden desarrollar con cualquiera de las estructuras del lenguaje. A continuación se presentan los mismos ejemplos que se revisaron previamente, pero empleando la estructura `while`.

Por ejemplo:

- Ejemplo 1: Se requiere un programa que lea y sume 100 números enteros.

```
int numero, suma = 0;  
int i = 1;  
while (i <= 100) {
```

```

        numero = int.Parse( Interaction.InputBox("Digite un número", "Lectura de datos", null, 100, 100) );
        suma = suma + numero;
        i=i+1;
    }
    salida.Text = suma.ToString();

```

En este caso, la operación es exactamente igual a la obtenida con la estructura *for*. Observe que la iniciación de la variable de control (*i*) se realiza *antes de* empezar el ciclo y el incremento (variación de valor) se hace dentro del ciclo.

- Ejemplo 2: Se requiere un programa que lea 10 números enteros y al final indique cuántos fueron positivos, cuantos negativos y cuántos cero.

```

int positivos=0;
int negativos = 0;
int ceros = 0;
int numero;
String numerotexto;

int num=0;
while (num<10) {
    numerotexto= Interaction.InputBox("Digite el número", "Lectura de datos", null, 100, 100);
    numero = int.Parse(numerotexto);
    if (numero >0 ) {
        positivos = positivos +1;
    }
    else {
        if (numero < 0) {
            negativos = negativos +1;
        }
        else {
            ceros = ceros +1;
        } //else
    } // else
    num = num + 1;
} //while

```

En este caso, debe prestarse especial atención a que el incremento de la variable de control (*num*) se realice *por fuera de cualquier decisión* para asegurar que, sin importar el resultado de las decisiones, se incremente la variable.

- Ejemplo 3: Se requiere un programa que lea las edades de 150 estudiantes de primer semestre de una carrera y calcule la edad promedio de dichos estudiantes.

```

...
int edad;
int edadTotal = 0;
double edadPromedio;
int cont = 0;

```

```

while (cont < 150){
    edad= int.Parse(Interaction.InputBox("Digite la edad", "Lectura de datos", null, 100, 100));
    edadTotal = edadTotal + edad;
    cont = cont + 1;
}
edadPromedio = edadTotal / 150.0;
...

```

Sin embargo, hay otro tipo de aplicaciones que resultan más fáciles de desarrollar con una estructura *while* que con una estructura *for*. Por ejemplo, aquellas donde *no* se sabe de antemano cuántas veces se van a repetir las instrucciones, sino que depende de alguna variable leída o calculada durante el ciclo. Por ejemplo:

- Ejemplo 4: Se quieren leer y sumar tantos números como el usuario desee.

```

int numero, numerototal=0;
char otro = 'S';
while (otro == 'S') {
    numero = int.Parse(Interaction.InputBox("Digite el número", "Lectura de datos", null, 100, 100));
    numerototal = numerototal + numero;
    otro = char.Parse(Interaction.InputBox("¿Desea leer otro número?", "Lectura de datos", null, 100, 100));
}

```

En este caso no se puede saber previamente (y no es necesario saberlo) cuántas veces se va a repetir el ciclo, se ejecutan las acciones y, al final, se pregunta al usuario si quiere ingresar otro número. Se repetirá mientras el usuario digite S. Si digita cualquier otro caracter, el ciclo terminará.

IMPORTANTE: Como se puede observar, esta clase de ciclo inicia preguntando sobre la validez de la condición. Por ello es importante que la condición sea verdadera al menos una vez, de otra forma *nunca* se realizaría el ciclo. En el caso anterior, la variable *otro* se inicia con el valor 'S' antes de iniciar el ciclo. Si no se hiciera este paso, el ciclo no se realizaría.

Forma 3: Ciclo DO - WHILE

La sintaxis del ciclo *do-while* es la siguiente:

```

do {
    //Instrucciones a repetir
} while (condición);

```

```
} while (condición);
```

Este ciclo maneja dos palabras clave: **do** que marca el inicio del ciclo y va seguido de la llave de apertura del bloque y **while**, escrito a continuación de la llave que cierra el bloque, va seguido de la condición y debe cerrarse con punto y coma (;).

Al igual que las otras estructuras presentadas, este ciclo se repite cuando (mientras) la condición sea verdadera. Sin embargo, tiene una diferencia fundamental con las otras estructuras y es que, en este caso, el ciclo siempre se ejecuta al menos una vez. Debido a que la condición se *escribe al final*, las instrucciones del ciclo siempre se ejecutarán *al menos por una vez*, después de lo cual se revisa la condición y entonces podría terminar el ciclo. En los ciclos *for* y *while*, la condición se encuentra *al inicio* de la estructura, por lo cual es *posible* que al empezar el ciclo la condición sea falsa y, por lo tanto, no se ejecuten ni siquiera una vez.

Utilización: Al igual que la estructura *while*, el *do-while* se puede emplear cuando se tienen ciclos en los que se conoce o no con antelación el número de veces que se va a repetir el ciclo. Adicionalmente, por su condición particular, esta clase de estructura es muy útil cuando se quiere ejecutar las instrucciones *al menos una vez*, por ejemplo para la *validación* de datos de lectura.

Retomamos los ejemplos anteriores, para presentar cómo se pueden desarrollar empleando esta estructura:

- Ejemplo 1: Se requiere un programa que lea y sume 100 números enteros.

```
int numero, suma = 0;
int i = 1;
do {
    numero = int.Parse( Interaction.InputBox("Digite un número", "Lectura de datos", null, 100, 100) );
    suma = suma + numero;
    i=i+1;
} while (i <= 100) ;
salida.Text = suma.ToString() ;
```

Para este caso, nuevamente tenga en cuenta que la iniciación de la variable de control (i) se realiza antes de empezar el ciclo y el incremento (variación de valor) se hace dentro del ciclo.

- Ejemplo 2: Se requiere un programa que lea 10 números enteros y al final indique cuántos fueron positivos, cuantos negativos y cuántos cero.

```
int positivos=0;
int negativos = 0;
int ceros = 0;
String numeroTexto;
int numero;
int cont = 0;
```

```

do {
    numeroTexto= Interaction.InputBox("Digite el número", "Lectura de datos", null, 100, 100);
    numero = int.Parse(numeroTexto);
    if (numero > 0) {
        positivos = positivos + 1;
    }
    else {
        if (numero < 0) {
            negativos = negativos + 1;
        }
        else {
            ceros = ceros + 1;
        } // else
    } // else
    cont = cont + 1;
} while (cont < 10);

```

- Ejemplo 3: Se requiere un programa que lea las edades de 150 estudiantes de primer semestre de una carrera y calcule la edad promedio de dichos estudiantes.

```

...
int edad;
int edadTotal = 0;
double edadPromedio;
int cont = 0;
do {
    edad= int.Parse(Interaction.InputBox("Digite la edad:", "Lectura de datos", null, 100, 100));
    edadTotal = edadTotal + edad;
    cont = cont + 1;
} while( cont < 150 );
edadPromedio = edadTotal / 150.0;
...

```

- Ejemplo 4: Se quieren leer y sumar tantos números enteros como el usuario desee.

```

int numero, suma = 0;
char otro;
do {
    numero = int.Parse(Interaction.InputBox("Digite el número: ", "Lectura de datos", null, 100, 100));
    suma = suma + numero;
    otro = char.Parse(Interaction.InputBox("¿Desea otro número? [S/N]: ", "Lectura de datos", null, 100, 100));
} while( otro == 'S' );

```

Para este caso, el uso de la estructura *do-while* tiene una ventaja: no es necesario asignar valor a la variable de control (*otro*) antes de iniciar el ciclo, ya que la lectura de la variable se realiza antes de terminar el ciclo y por tanto, en el momento que se evalúa la condición, la variable ya tiene un valor válido.

Un caso en el que es muy útil este tipo de estructuras es cuando se requiere **validar** un dato de entrada, para verificar que el valor ingresado se encuentre dentro del rango de datos aceptados para el programa; la idea es que si el usuario no ingresa el valor apropiado, se pida nuevamente el dato hasta tanto el valor ingresado se encuentre en el rango válido. Por ejemplo:

- Ejemplo 5: El número de estudiantes debe ser mayor que 0

```
int numEst;
do {
    numEst = int.Parse(Interaction.InputBox("¿Cuántos estudiantes hay?", "Número de estudiantes",
        null, 100, 100));
} while ( numEst <= 0 );
```

Debe recordarse que el ciclo se va a repetir cuando la condición establecida sea verdadera. Entonces, para estos casos conviene preguntarse *¿Si se requiere que el número sea mayor que 0, cuándo deberá repetirse el ciclo?* En este caso, el ciclo deberá repetirse, (es decir, volver a preguntar el número de estudiantes) cuando **no sea mayor que 0**, o sea, cuando el número **sea menor o igual que 0**.

- Ejemplo 6: La nota ingresada debe estar entre 0.0 y 5.0, ambos valores inclusive.

Nuevamente, debe hacerse la pregunta, *¿en qué caso se debe repetir la lectura?* Respuesta: cuando la nota ingresada no esté entre 0.0 y 5.0; es decir, sea menor que 0.0 o mayor que 5.0

```
double nota;
do {
    nota = double.Parse(Interaction.InputBox("Digite la nota: ", "Lectura de notas", null, 100, 100));
} while( nota < 0.0 || nota > 5.0 );
```

- Ejemplo 7: El estado civil puede ser S, C o D.

Ante la pregunta sobre cuando repetir el ciclo, sería cuando el estado civil sea diferente de S, diferente de C y diferente de D.

```
char estado;
do {
    estado = char.Parse(Interaction.InputBox("¿estado civil?[S/C/D]:", "Estado civil", null, 100, 100));
} while( estado != 'S' && estado != 'D' && estado != 'C' );
```

- Ejemplo 8: El programa puede ser Ingeniería o Economía.

En este caso, el ciclo debe repetirse cuando el programa ingresado sea diferente a Ingeniería y diferente a Economía.

```
String programa;  
do {  
    programa = Interaction.InputBox("¿Programa académico? [Ingeniería/Economía]:",  
        "Programa académico", null, 100, 100);  
} while ( !programa.equals("Ingeniería") && !programa.equals("Economía") );
```


V. MÉTODOS

Como se mencionó previamente, los métodos se pueden considerar mini-programas, que tienen una función o propósito específico. En este sentido, los métodos tienen unos aspectos que los identifican:

- Un método tiene un único propósito u objetivo. Un método realiza una única tarea de forma completa.
- Un método puede ser llamado o invocado desde múltiples puntos del programa. Al llamar un método, lo que ocurre es que se ejecutan las operaciones que este incluye y, una vez termine, se continúa la ejecución del programa en el punto donde había quedado.
- No es común que se tengan operaciones de lectura o escritura en un método, ya que el método, en general, se emplea para realizar ciertas operaciones precisas. En caso de ser requerido, pueden hacerse operaciones de lectura y/o escritura en un método, pero no es común.

Utilización: Los métodos son bastante útiles cuando se dispone de una operación que es común y que puede ser empleada en múltiples ocasiones. Por ejemplo, las funciones que se pueden ejecutar con la calculadora (log, sen, cos, tan,...), es más fácil tener el método (la función matemática en este caso) y llamarlo, que tener que escribir las instrucciones cada vez que se quiere ejecutar la operación.

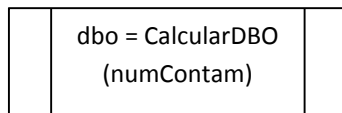
Elementos

Un método tiene los siguientes elementos:

- **Parámetros de entrada:** corresponde a los datos que se requieren que sean enviados al método para realizar su operación. Son equivalentes a los datos de entrada a un programa; solamente que en el método no se leen, sino que se reciben. Por ejemplo, si se considera como método SUMAR, se requiere que sean enviados dos números para sumar, los cuales serían los parámetros de entrada. Algunos métodos podrían operar sin datos de entrada.
- **Dato de Salida:** es el dato (variable) que se obtendrá como resultado después de ejecutar las instrucciones del método.
- **Instrucciones:** Corresponde a todas las instrucciones que se ejecutarán al interior del método. Un método puede incluir cualquier clase de instrucciones: cálculos, decisiones, ciclos, llamado a otro método, etc., aunque, como ya se especificó, no es común que existan operaciones de lectura y escritura en un método.

En el Flujograma

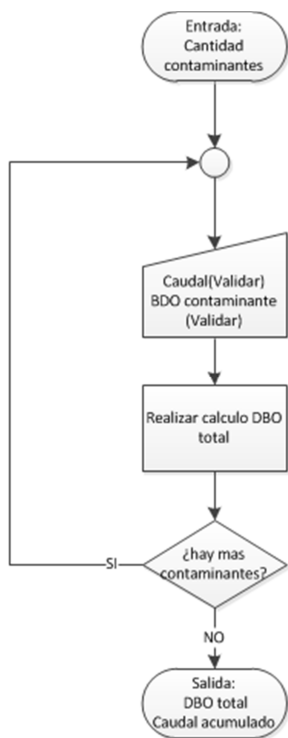
En el flujograma, el llamado a un método se representa como una caja con doble línea vertical a cada lado, dentro de la cual se escribe el nombre del método, con los datos (ó *argumentos*) que se le *envían* (entre paréntesis y separados por comas) y la variable en la cual es asignado el valor que éste devuelve. Si el método no devuelve un valor (método tipo *void*), se omite esta variable.



El método tiene su propio flujograma, que presenta el proceso que se lleva a cabo dentro del método. La única diferencia con el flujograma *principal*, es que el punto de entrada al método se nombra como Entrada, en lugar de Inicio, y el punto de salida del flujograma se nombra como Salida, en lugar de Fin.

En la entrada se muestran los datos requeridos que el método *recibe* (parámetros). En la salida, se indica el nombre de la variable que se devuelve como resultado de la ejecución del método. En todo caso, muestre siempre la palabra Entrada y la palabra Salida, así el método no reciba parámetros o no devuelva un resultado.

Ejemplo:



En Visual C#:

En Java los métodos tienen dos partes esenciales, el encabezado (también llamado cabezote, *signatura* o firma del método) y el cuerpo del método.

- La signatura del método incluye cuatro partes: **(1)** el modificador de acceso del método; **(2)** el tipo de retorno, que corresponde al *tipo de dato* que el método va a devolver. Si no devuelve ningún dato, se escribirá *void*; **(3)** el nombre del método, que sigue los mismos principios de cualquier identificador: debe iniciar con una letra, no tener espacios ni caracteres especiales; **(4)** tipo y nombre de los parámetros: para cada dato de entrada debe incluirse el tipo y un nombre de variable que será empleado en las instrucciones al interior del método.

La *signatura* entonces, tendría la siguiente estructura⁶:

```
public <tipo de retorno> <nombre del método> ( <tipo y nombre de los parámetros> )
```

- Cuerpo del método: corresponde a las instrucciones que se incluyen en el método para cumplir su función. Si el método retorna algún dato como resultado, deberá incluir la instrucción *return*, seguida de un valor o variable del *tipo de dato* definido para el <tipo de retorno>. Debe tenerse en cuenta que la instrucción *return* finaliza la ejecución del método y regresa el control al programa principal, en el punto en que fue llamado o invocado el método.

Por ejemplo:

- Ejemplo 1: Realizar un método que compare dos números enteros e indique el de mayor valor.

En este caso, el método requiere como parámetros de entrada, dos números enteros; el dato a regresar (o retornar) es, a su vez, un número entero. La signatura del método quedaría de la siguiente forma:

```
public int calcularMayor( int numero1, int numero2 )
```

después de la palabra clave *public* se incluye *int*, haciendo referencia al *tipo de dato* que el método va a retornar, después el nombre del método: *calcularMayor*. Finalmente, entre paréntesis y separados por comas, los parámetros de entrada: dos números enteros.

El método completo se podría ver así:

```
public int calcularMayor(int numero1, int numero2) {  
    int mayor;  
    if (numero1 > numero2) {  
        mayor = numero1;  
    }  
    else {  
        mayor = numero2;  
    }  
    return mayor;  
}
```

⁶ Los signos de < > se emplean para indicar que en este sitio se incluye un valor variable.

Observe que: las variables `numero1` y `numero2` no deben declararse nuevamente dentro del método, ya que quedan declaradas en la definición de parámetros y, por tanto, pueden emplearse al interior del método. Sin embargo, la variable `mayor` sí debe declararse dentro del método y al final será retornada. En este caso, la variable a retornar (*mayor*) debe ser *del mismo tipo* que se ha definido en el encabezado del método.

- Ejemplo 2: Se requiere un método que lea 10 números reales y retorne su promedio.

En este caso, se podrían ingresar los 10 números como parámetro pero es bastante engorroso. Es más fácil leerlos dentro del método y hacer el cálculo requerido.

```
public double promedio() {
    double numero;
    double promedio=0;
    for( int i = 0; i < 10; i=i+1 ) {
        numero = double.Parse(Interaction.InputBox("Digite un número: ", "Lectura de datos", null, 100,
            100));
        promedio = promedio + numero;
    }
    promedio = promedio / 10;
    return promedio;
}
```

Observe que: En este método no se requieren parámetros de entrada, sin embargo deben incluirse los paréntesis respectivos.

- Ejemplo 3: Se requiere un método que reciba dos números enteros y *muestre* el mayor.

Este es un caso, en el que se presentan los datos de salida dentro del método y por lo tanto, no es necesario retornar ningún valor.

```
public void mostrarMayor( int numero1, int numero2 ) {
    if( numero1 > numero2 ) {
        textBoxMayor.Text = numero1.ToString( );
    }
    else {
        if( numero2 > numero1 ) {
            textBoxMayor.Text = numero2.ToString( );
        }
        else {
            textBoxMayor.Text = "Los dos números son iguales";
        }
    }
}
```

Observe que: En este caso, no se debe retornar ningún valor, por lo que el método se declara como **void** en el encabezado y no hay sentencia que retorne un valor (no hay `return valor;`).

Invocación o llamado al método

Una vez definido el método, se puede llamar o invocar desde otra parte del programa. Para ello, se utiliza el nombre del método, enviándole valores o variables (ó argumentos) del mismo tipo de los parámetros de entrada definidos en el método. Adicionalmente, si el método retorna algún valor, debe asignarse el valor en una variable cuyo *tipo de dato* sea compatible con el *tipo de retorno* del método.

Ejemplo:

- Para el método que tiene el siguiente encabezado:

```
public int calcularMayor(int numero1, int numero2) { ... }
```

son válidas las siguientes invocaciones:

```
int resultado = calcularMayor(3, 7);  
  
int primero = 4;  
int segundo = 59;  
int x = calcularMayor(primeros, segundo);
```

OBSERVE QUE: Aunque los parámetros que el método recibe se denominan `numero1` y `numero2`, los datos usados en su llamado o invocación pueden ser constantes, o pueden ser variables *que tengan otros nombres*. Lo importante es que sean *del mismo tipo* que los esperados en el método.

- Para el método que tiene el siguiente encabezado:

```
public double promedio( ) { ... }
```

es válida la siguiente invocación:

```
double prom = promedio();
```

- Para el método que tiene el siguiente encabezado:

```
public void mostrarMayor( int numero1, int numero2) { ... }
```

son válidas las siguientes invocaciones:

```
mostrarMayor(34, 8) ;  
  
int n1 = 56, n2 = 58 ;  
mostrarMayor( n1, n2 ) ;
```

Observe que: en este último caso, debido a que el método no retorna valor (es tipo `void`), no es necesario asignar el llamado del método a una variable.