5 CICLOS

Las instrucciones condicionales (Capítulo 4) permiten controlar el flujo de ejecución en un programa, ya que a partir de una condición la ejecución de las acciones puede tomar un camino u otro.

En este capítulo el lector encontrará otro grupo de instrucciones de control de flujo, que se conocen como **instrucciones de ciclo** porque están relacionadas con la repetición de acciones. Precisamente la solución por computación a muchos problemas requiere de repetir una secuencia de acciones para lograr un objetivo, con ello se aprovechan las cualidades en que las computadoras aventajan a los humanos: velocidad, confiabilidad y "obediencia" (a diferencia de los humanos las computadoras no protestan, ni se aburren, ni se equivocan, ni se distraen porque tengan que estar repitiendo lo mismo). Cualquier programa que resuelva algún problema de interés tendrá que hacer cálculos repetidamente.

Las razones de proceso repetitivo pueden ser varias: porque se quiere ejecutar repetidamente un grupo de acciones mientras se cumpla una cierta condición, o para repetir el grupo de acciones una cantidad de veces, o para aplicar una secuencia de acciones a cada uno de los elementos de una colección de datos. Para cada una de estas diferentes causas y formas de repetición existen instrucciones de ciclos en C# que se estudian en este capítulo.

5.1 Ciclo do while

Retomemos el código que pretende medir el tiempo de teclear el nombre. La solución (Listado 5.1) que se plantea en el Capítulo 4 es muy drástica: si nos equivocamos lo que se ha hecho es dar un mensaje de error, terminar la ejecución y pedir al usuario que vuelva a ejecutar el programa si quiere probar de nuevo.

```
Console.WriteLine("Teclea tu nombre");
string nombre1 = Console.ReadLine();
Console.WriteLine("\nVuélvelo a teclear");
string nombre2 = Console.ReadLine();
if (nombre1 == nombre2)
  Console.WriteLine("\nHola {0}, has tecleado bien tu nombre", nombre1);
else
  Console.WriteLine("\nERROR has tecleado mal.
                     Vuelve a ejecutar el programa");
```

Listado 5.1. Reacción drástica ante un potencial error

Lo que realmente sería deseable es darle la oportunidad al usuario de repetir la acción de teclear su nombre. Esto se puede lograr con una instrucción de ciclo `do while` (Listado 5.2) que repetirá el conjunto de instrucciones mientras los nombres tecleados sean diferentes.

```
using System;
```

```
namespace WEBOO.Programacion
{
  class Program
  {
    static void Main(string[] args)
    {
      string nombre1, nombre2;
      do
      {
        Console.WriteLine("Teclea tu nombre");
        nombre1 = Console.ReadLine();
        Console.WriteLine("\nVuélvelo a teclear");
        nombre2 = Console.ReadLine();
      }
      while (nombre1 != nombre2);
      Console.WriteLine("\nHola {0},
                         ya has tecleado bien tu nombre", nombre1);
    }
  }
}
```

Listado 5.2. Ciclo do while

Si el nombre no se teclea dos veces consecutivas, igual se pedirá que se vuelva a teclear. Una posible ejecución sería:

```
Teclea tu nombre
Juana

Vuélvelo a teclear
Juan

Teclea tu nombre
Juana

Vuélvelo a teclear
Juana

Hola Juana, ya has tecleado bien tu nombre
```

Esto aún nos deja insatisfechos porque al cometer el error no se ha dado ningún mensaje de aviso al usuario, simplemente se le ha vuelto a escribir el mensaje:

```
Teclea tu nombre
```

La instrucción `do while` tiene la forma

```
do
  <bloque de instrucciones>
while <condicion>
```

La ejecución del *<bloque de instrucciones>* se hace al menos una vez y se repetirá mientras la *<condición>* evalúe verdadero. Es de esperar que en algún momento la

ejecución de las instrucciones produzca algún cambio que provoque que en la próxima evaluación la condición evalúe falso, de lo contrario la ejecución del ciclo sería infinita. Un ciclo "infinito" producto de una mala programación es un error frecuente que el programador debe cuidarse de no cometer.

5.2 Ciclo while

En el caso del ejemplo anterior, el ciclo `do while` se ejecuta al menos una vez porque se espera que los nombres se tengan que teclear.

Otro tipo de ciclo es el conocido como ciclo `while` que es de la forma

```
while <condicion>
  <bloque de instrucciones>
```

En este caso, la *<condicion>* se evalúa antes de comenzar el ciclo y el *<bloque de instrucciones>* se ejecutará mientras la condición evalúe verdadera. Es decir que si la condición evalúa falso desde la primera vez, el bloque de instrucciones no se ejecutará. Se supone entonces que la ejecución del bloque de instrucciones deberá provocar que en algún momento la condición evalúe true para que el ciclo termine y no se ejecute infinitamente.

El código del Listado 5.3 muestra un método `DecimalABinario` que recibe un número entero positivo y devuelve un valor de tipo `string` con la representación en binario del número.

Si el número es 0 se retorna `"0"` (línea 4 del listado), que es la representación en binario de `0`. De lo contrario, la variable `binario` mantiene su valor de inicialización `""` (es decir, la cadena vacía) y se entra en un ciclo que se repite mientras el número almacenado en la variable `n` sea mayor que cero. Si el resto de la división por `2` es `0`, entonces el número es par y su representación en binario termina en `0` y se concatena `binario = "0" + binario`; si el resto de la división es `1`, entonces el número es impar y su representación en binario termina en `1`, por eso se concatena `binario = "1" + binario`. Se toma ahora como nuevo número el cociente entero de la división por `2`, es decir `n = n/2`, y se vuelve a repetir el ciclo mientras ese cociente sea mayor que `0`.

```
static string DecimalABinario(int n)
{
  string binario = "";
  if (n == 0) return "0";
  while (n > 0)
  {
    if (n % 2 == 0) binario = "0" + binario;
    else binario = "1" + binario;
    n = n / 2;
  }
  return binario;
}
```

Listado 5.3. Método para llevar de decimal a un string con la representación en binario

No hay preferencia sobre cuándo usar un tipo de ciclo u otro. C# incluye ambas sintaxis para que el programador utilice la que más sencilla le resulte y se acomode mejor a la forma en que se plantea la repetición. Pruebe a reprogramar el método `DecimalABinario` usando un ciclo `do while`.

5.3 Abortar un ciclo. Instrucciones continue y break

Nos habíamos quedado insatisfechos con el código del Listado 5.2 porque se repetía el ciclo sin haber dado un mensaje de advertencia de que se había tecleado mal. El problema en este caso es que el mensaje de advertencia lo debemos dar antes de mandar a repetir el ciclo. Esto se podría solucionar reescribiendo el código como se muestra en el Listado 5.4.

```
1   string nombre1, nombre2;
2   do
3   {
4     Console.WriteLine("Teclea tu nombre");
5     nombre1 = Console.ReadLine();
6     Console.WriteLine("\nVuélvelo a teclear");
7     nombre2 = Console.ReadLine();
8     if (nombre1 != nombre2)
9       Console.WriteLine("Te has equivocado, vuélvelo a intentar");
10  }
11  while (nombre1 != nombre2)
12    Console.WriteLine("\nHola {0},
13                       ya has tecleado bien tu nombre", nombre1);
```

Listado 5.4. Código mejorado con mensaje de advertencia

Pero esta solución no es sintácticamente elegante, note que dentro del ciclo se ha hecho la pregunta `if (nombre1 != nombre2)` (línea 8) para que inmediatamente después se repita la misma en el `while (nombre1 != nombre2)` (línea 9). Además de que implica evaluar innecesariamente dos veces la misma expresión.

5.3.1 Instrucción break

El problema aquí es que la decisión para repetir o no el ciclo se quisiera tomar dentro el cuerpo del ciclo. Una solución mejor se muestra en el Listado 5.5 usando la instrucción `break` (línea 8). La instrucción `break` se puede usar dentro del bloque de instrucciones de un ciclo y significa abortar la ejecución del ciclo para continuar en la instrucción siguiente a la del ciclo. De modo que el hecho de que un ciclo se exprese en la forma `while (true)` no quiere decir que su ejecución sea infinita (porque la expresión `true` siempre dará valor `true`), sino que posiblemente su ejecución se interrumpirá por la ejecución de una instrucción `break` dentro del cuerpo del ciclo.

```
1   string nombre1, nombre2;
2   while (true)
3   {
4     Console.WriteLine("Teclea tu nombre");
5     nombre1 = Console.ReadLine();
6     Console.WriteLine("\nVuélvelo a teclear");
7
```

```
8     nombre2 = Console.ReadLine();
9     if (nombre1 == nombre2) break;
10    else
11      Console.WriteLine("Te has equivocado, vuélvelo a intentar");
12  }
13  Console.WriteLine("\nHola {0},
                      ya has tecleado bien tu nombre", nombre1);
```

Listado 5.5. Ciclo usando instrucción break

5.3.2 Instrucción continue

La instrucción `continue` también interrumpe la iteración de un ciclo, pero a diferencia del `break`, en lugar de salir del ciclo y seguir en la instrucción siguiente, lo que hace es continuar con una nueva iteración desde el comienzo del ciclo. El código del Listado 5.6 prueba el método `DecimalABinario` con los números que se vayan tecleando. La instrucción `break` se ha usado para abortar la prueba cuando se dé la cadena vacía (teclear *Enter*) en lugar de un número. La instrucción `continue` hace que se continué con otra iteración cuando se teclee un número negativo.

```
while (true)
{
  Console.WriteLine("\nEntre un número entero positivo");
  string s = Console.ReadLine();
  if (s.Length == 0) break;
  int k;
  if (TryParse(s, out k))
  {  int k = int.Parse(s);
     if (k < 0)
     {
       Console.WriteLine("El número debe ser >= 0");
       continue;
     }
  }
  else
  {
    Console.WriteLine("Debe teclear un número >= 0");
    continue;
  }
  Console.WriteLine("{0} en binario es {1}", k, DecimalABinario(k));
}
```

Listado 5.6. Ciclo con break y continue

5.4 Ejemplos de ciclos while

5.4.1 Método para determinar si un número es primo

Un número es primo si solo es divisible por él mismo y por `1`, de modo que una solución sencilla para determinar si un número es primo es recorrer todos los números desde `2` y

menores que el número, y si nos encontremos uno que lo divide entonces sabemos que el número no es primo y abortamos el recorrido. El método `Primo` (Listado 5.7) devuelve `bool` para indicar si el número que recibe como parámetro `n` es un número primo. Aquí la instrucción `return false` significa salir del método ha servido también para abortar el ciclo

```
static bool Primo(int n)
{
  int k = 2;
  while (k < n)
  {
    if (n % k == 0) return false;
    else k++;
  }
  return true;
}
```

Listado 5.7. Método para determinar si un número es primo

Note que en el ciclo se hacen iteraciones innecesarias. Si un valor mayor que la raíz cuadrada del número dividiese al número, entonces el cociente de esa división también dividiría al número y por consiguiente el ciclo ya se hubiese abortado antes con el `return false`. Es decir, si `n%q` es igual a `0` siendo `q` mayor que `Sqrt(n)`, entonces `n/q` es `m`, donde `m` sería menor o igual que `Sqrt(n)`. Por consiguiente, no hay que probar si son divisores los números mayores que `Sqrt(n)` y el ciclo puede terminar antes. La solución mejorada se muestra en el Listado 5.8.

```
static bool Primo(int n)
{
  int k = 2;
  int raiz = (int)Math.Sqrt(n);
  while (k <= raiz)
  {
    if (n % k == 0) return false;
    else k++;
  }
  return true;
}
```

Listado 5.8. Método Primo mejorado

5.4.2 Método para determinar el Máximo Común Divisor de dos números

El máximo común divisor (MCD) entre dos números enteros positivos es el mayor número entero positivo que divide a ambos números. Por ejemplo, el MCD entre 12 y 8 es 4, ya que cualquier otro número natural mayor que 4, no divide a 12 y 8. Se quiere escribir un método que reciba dos parámetros enteros positivos y devuelva el MCD.

Evidentemente, el MCD no puede ser mayor que el menor de ambos números. Por lo tanto, una solución obvia es comenzar con una variable que tenga como valor inicial el menor de ambos números y hacer un ciclo probando con todos los valores menores que

ese valor inicial hasta encontrar el primero que los divida a ambos. El ciclo siempre terminará porque en el peor caso, si los números son primos entre sí, se llegaría a 1, que los divide a ambos.

El código del Listado 5.9 implementa el método `Mcd`. En las líneas 3 y 4 se verifica que ambos números sean positivos y en caso contrario se dispara una excepción `ArgumentException` con el mensaje correspondiente. Luego al hacer `int mcd = Math.Min(m, n);` se inicializa la variable `mcd` con el menor de los dos parámetros `m` y `n` (usando el método `Min` de la clase `Math`). El ciclo `while` se repite mientras el valor de la variable `mcd` no divida a algunos de los dos números `m` o `n` (línea 6). Como en cada iteración se decrementa el valor de `mcd` el ciclo terminará. Cuando termine el ciclo el valor en `mcd` será el máximo común divisor.

```
static int Mcd(int m, int n)
{
  if (m<=0 || n<=0)
    throw new System.ArgumentException("los números deben ser positivos")
  int mcd = Math.Min(m, n);
  while (m % mcd != 0 || n % mcd != 0)
    mcd--;
  return mcd;
}
```

Listado 5.9. Máximo Común Divisor

Este tipo de solución se denomina de **fuerza bruta** porque para encontrar la solución se prueban todos los casos posibles.

Una solución de fuerza bruta puede no ser eficiente, porque para encontrar la respuesta la solución prueba tal vez casos que podrían haberse obviado. Sin embargo, en compensación por lo general una solución de fuerza bruta es más simple de entender. Encontrar una solución "más eficiente" (que obvie probar algunos casos y que llegue de modo más rápido al resultado) puede ser más complicado y el código resultante menos claro de comprender. No hay una posición absoluta sobre hacia dónde inclinar la balanza. Sobre este tema del costo de los algoritmos y distintas estrategias de solución se trata en el Capítulo 9.

Una solución mejor es la del método de Euclides.

Este método concebido, por el célebre pensador griego **Euclides**, se considera como el "más antiguo algoritmo conocido" aunque por supuesto que en aquellos tiempos aún no se había planteado el concepto de algoritmo

Este método se basa en que si *n* es mayor que *m*, entonces el MCD de *n* y *m* es el mismo que el de *m* y *n* - *m*.

Note que si *p* divide a *m* y a *n-m* es porque existen *q* y *r* tales que

*(1) m = p*q* y
*(2) n-m = p*r*

Sustituyendo (1) en (2) se tiene que *n = p*r + p*q*, de modo que *n = p*(r+q)*, lo que dice que *p* es también divisor de *n*.

Quedaría por demostrar que todo número que divida a *n* y *m* divide también a *n-m*.

Si *k* divide a *n* y *m* es porque existen *j* e *i* tales que *n = k*j* y *m=k*i*, de modo que *n-m = k*(j-i),* lo que quiere decir que *k* es también divisor de *n-m*.

Por lo tanto *n*, *m* y *n-m* tienen los mismos divisores, y por consiguiente calcular el MCD de *n* y *m* puede reducirse a calcular el MCD de *m* y *n-m*. Si se repite este proceso siempre quedándonos con el menor y la diferencia entre el mayor y el menor, llegará un momento en que esa diferencia será cero y en tal caso el otro será el MCD. El código del Listado 5.10 implementa este algoritmo.

```
1   static int Mcd(int m, int n)
2   {
3     if (m <= 0 || n <= 0)
4       throw
5         new System.ArgumentException("Los números deben ser positivos");
6     int t;
7     while (n > 0)
8     {
9       //Siempre tener en n el mayor de los dos
10      if (n < m)
11      {
12        t = n; m = n; n = t;
13      }
14      n = n - m;
15    }
16    return m;
    }
```

Listado 5.10. Máximo Común Divisor por el algoritmo de Euclídes

Note que en las líneas 9-12 se garantiza que siempre se tenga en `n` el mayor de los dos. Y en la línea 13 al mayor se le resta el menor. Esta resta *n-m* se seguirá haciendo, dejando en *n* el mayor hasta que se llegue a un número menor que *m*, lo que equivale al valor resultante de haber aplicado el operador `%` (resto de la división). De manera que el código del Listado 5.10 puede ser mejorado por el código del Listado 5.11.

```
static int Mcd(int m, int n)
{
  if (m <= 0 || n <= 0)
    throw
      new System.ArgumentException("los números deben ser positivos");
  int r = m % n;
  while (r != 0)
  {
    m = n;
    n = r;
    r = m % n;
  }
  return n;
}
```

Listado 5.11. Máximo Común Divisor mejorado usando el operador %

Seguramente al lector le resultará más difícil comprender las soluciones presentadas en Listado 5.10 y Listado 5.11, que la solución de fuerza bruta del **Error! Reference source not found.**.

Seguramente la diferencia en tiempo de las tres soluciones no es perceptible al ojo humano. Mida el tiempo de ejecución de las tres soluciones usando un objeto `Stopwatch`. Posiblemente la diferencia en tiempo tampoco sea apreciable porque a nivel de milisegundos (que es lo que mide el `Stopwatch`) en todos los casos el tiempo medido sea cero. Coloque una variable `iteraciones` en cada uno de los tres códigos para contar la cantidad de iteraciones que se hacen en cada una de las tres soluciones.

5.5 Ciclo for

El método del Listado 5.12 recorre en el ciclo `while` los números de 1 a *n*.

```
1   public static int SumaNPrimeros(int n)
2   {
3     int suma = 0;
4     int k = 1;
5     while (k<=n)
6     {
7       suma += k;
8       k++;
9     }
10    return suma;
11  }
```

Listado 5.12. Suma de los n primeros números naturales usando un ciclo while

Note que en este ciclo se usa una variable `k` (líneas 4, 5, 7 y 8) para recorrer todos los números naturales del 1 al *n* e irlos sumando. Distingamos tres partes de este código: en la línea 4 se **inicializa** la variable en `1`, en la línea 5 se **pregunta por la variable para decidir si terminar o repetir el ciclo** y en la línea 8 se **modifica** el valor de la variable.

Es decir, éste es un ciclo en el que se usa una variable para recorrer una secuencia de valores mientras se cumpla una condición (que involucra a la variable). C# dispone de una sintaxis especial para escribir este tipo de ciclos: el ciclo `for`. El Listado 5.13 nos muestra un método `SumaNPrimeros` equivalente al del Listado 5.12 pero utilizando un ciclo `for`.

```
1   public static int SumaNPrimeros(int n)
2   {
3     int suma = 0;
4     for (int k = 1; k <= n; k++)
5       suma += k;
6     return suma;
7   }
```

Listado 5.13. Suma de los n primeros números usando un ciclo for

El código es más simple que el del Listado 5.12, ya que el uso de una variable para recorrer los valores, su inicialización antes de comenzar el ciclo, el criterio por el cual se

repite el ciclo y cómo cambia la variable en cada iteración queda evidenciado en el encabezado del ciclo:

```
for (int k = 1; k <= n; k++)
```

al que debe seguir el bloque de instrucciones que se repite, en este caso la simple instrucción `suma += k;`

Este tipo de ciclo tiene la sintaxis:

```
for (<inicialización>; <condición>; <actualización>)
      <instrucción>
```

Tanto *<inicialización>* como *<condición>* y *<actualización>* pueden omitirse. De modo que los siguientes ciclos son sintácticamente correctos:

```
for (int k = 0; ;) Console.Write(".");
for (; ; ) Console.Write(".");
int j = 0;
for (; ; j++) Console.Write(".");
```

Todos estos ciclos ejecutan infinitamente; la ausencia de *<condición>* se interpreta como el valor `true`, de modo que los anteriores son ciclos que nunca terminan.

Note que un ciclo `while` como:

```
while (<condición>)
  <instrucción>
```

se puede escribir de forma equivalente con el ciclo `for` siguiente:

```
for (;<condición>;) <instrucción>
```

Por supuesto que la utilidad del ciclo `for` es expresar con una sintaxis más clara y sencilla un ciclo en el que una variable recorre una secuencia de valores enteros.

El uso del ciclo `for` está mayormente asociado para recorrer mediante índices enteros los elementos de un array o de otras estructuras de datos que sean "indizadas" (los arrays se estudian en el Capítulo 7).

El lector puede comprobar que los tres tipos de ciclos `do while`, `while` y `for` son sustituibles entre sí haciendo un uso adecuado de las condicionales `if` y las sentencias `break` y `continue`. El programador debe utilizar el ciclo que le resulte más simple para la iteración.

Este ejemplo se ha incluido solo con propósitos ilustrativos, ya que no hace falta realizar ningún proceso repetitivo para calcular la suma de los *n* primeros números. Se cuenta que cuando Gauss, uno de los matemáticos más grandes de inicios del siglo XIX, asistía a sus primeros años en la escuela, mostraba muy poco interés por lo aburrido de las matemáticas que le impartían. En una ocasión en que el niño hiciera una travesura durante sus clases de matemáticas, el profesor le puso como castigo que no podía irse a casa hasta que no *sumase todos los números del uno al cien*. El profesor quedó muy sorprendido cuando al poco tiempo el niño le entregara la respuesta de 5050. Aunque no disponía de computadoras ☺, Gauss había descubierto que no tenía que realizar las cien sumas porque el valor se podía calcular directamente con la fórmula $(n * (n-1))/2$

5.5.1 Ciclo cantidad de veces. El elemento n-ésimo de la sucesión de Fibonnaci

Uno de los usos que se le suele dar al ciclo `for` es cuando un proceso iterativo se quiere repetir una **cantidad** de veces. El ciclo utilizado en el método `Fibonnaci` del Listado 5.14 calcula el elemento *n*-ésimo de la sucesión de Fibonnaci. La sucesión de Fibonnaci es de la forma:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
```

es decir, los dos primeros elementos son `1,` y a partir del tercero en adelante cada elemento es igual a la suma de los dos anteriores.

```
1   static long Fibonnaci(int n)
2   {
3     long anterior = 1;
4     long ultimo = 1;
5     for (int k = 3; k <= n; k++)
6     {
7       long t = anterior;
8       anterior = ultimo;
9       ultimo = t + ultimo;
10    }
11    return ultimo;
12  }
```

Listado 5.14. Cálculo del elemento n-ésimo de la sucesión de Fibonnaci

Note que el ciclo `for` (línea 5) comienza con `k = 3` y se repite mientras `k<=n,` de modo que para valores de `n` que sean `1` o `2` el ciclo no se ejecuta y el valor devuelto es `1`. En cada iteración del ciclo se calculan los dos últimos elementos de la sucesión a partir de los dos últimos calculados en la iteración anterior (líneas 7, 8 y 9). Como la última vez que se ejecuta el ciclo es cuando `k` es igual a `n`, entonces al terminar el ciclo el valor en la variable `ultimo` es el elemento *n*-ésimo en la sucesión.

5.6 Ciclos dentro de ciclos

Se pueden tener ciclos anidados dentro de otros ciclos. El código del método `PrimoMasCercano` (Listado 5.15) determina cuál es el número primo más cercano al valor que recibe como parámetro. Note como dentro del ciclo que va recorriendo los valores mayores y menores que el número se ejecuta el código que antes hemos presentado como parte del método `Primo` (Listado 5.7).

```
public static int PrimoMasCercano(int n)
{
  int i = n + 1;
  int j = n - 1;
  int k, raiz;
  while (true)
  {
    raiz = (int)Math.Sqrt(i);
    k = 2;
```

```
    while (k <= raiz)
    {
      if (i % k == 0) return i;
      else k++;
    }
    raiz = (int)Math.Sqrt(j);
    k = 2;
    while (k <= raiz)
    {
      if (j % k == 0) return j;
      else k++;
    }
    i++; j--;
  }
}
```

Listado 5.15. Ciclos anidados para calcular el primo más cercano

En tales situaciones el ciclo más interno suele depender de alguna manera del ciclo que lo contiene, como en el caso de este ejemplo en que las variables `i` y `j` que cambian en el ciclo más externo se usan para calcular la `raiz` y determinar la condición de parada dentro de los ciclos `while` más internos.

Realmente este código se podría haber escrito de forma más abreviada y simple si utilizamos a su vez el método `Primo` como se muestra en el Listado 5.16. Pero esto no significa que no se estén ejecutando dos ciclos de forma anidada.

```
public static int PrimoMasCercano(int n)
{
  int i = n + 1;
  int j = n - 1;
  while (true)
  {
    if (Primo(i)) return i;
    if (Primo(j)) return j;
    i++; j--;
  }
}
```

Listado 5.16 Ciclos anidados camuflados mediante el uso de métodos

Note que cuando se tienen ciclos anidados el tiempo de cómputo aumenta, porque por cada iteración del ciclo más externo se hacen todas las iteraciones del ciclo más interno. El programador debe cuidar por tanto de que tal anidamiento sea necesario para la solución del problema que se quiere resolver.