4 INSTRUCCIONES CONDICIONALES

Retomemos el ejemplo para medir el tiempo de teclear nuestro nombre (Capítulo 2). ¿Cómo podemos mejorarlo para tener más certeza que se ha tecleado el nombre correcto y acercarnos a medir con más exactitud el tiempo de tecleo? Vamos a pedir teclear dos veces el nombre y suponer que si las dos veces se ha tecleado lo mismo es que se ha tecleado correctamente[1]. El código del Listado 4.1 nos permite lograr esto.

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

Listado 4.1 Validar que el nombre del usuario sea correcto

Este código introduce un nuevo tipo de instrucción en la línea 5 que se denomina **instrucción condicional**. Seguido de la palabra reservada `if` hay una expresión de tipo `bool` que compara los dos nombres. Si la expresión evalúa `true`, entonces se ejecuta la instrucción siguiente al `if`, y si evalúa `false` se ejecuta la instrucción que sigue al `else`.

A este tipo de instrucciones se les conoce como **instrucción de control de flujo** porque incide en el flujo de ejecución de las instrucciones. Sería muy reducido el tipo de problemas que podríamos resolver con las computadoras si las instrucciones solo pudieran ejecutarse secuencialmente una tras otra. En la mayoría de los algoritmos se deben tomar decisiones sobre qué alternativa seguir ante diferentes situaciones. Las instrucciones condicionales nos ayudan a expresar este tipo de situaciones.

4.1 Instrucción Condicional IF

La sintaxis de la instrucción condicional `if` tiene la forma:

```
if(<condición>)
    <parte if>
else
    <parte else>
```

Aquí *<condición>* tiene que ser una expresión de tipo `bool`. La *<parte if>* y la *<parte else>* pueden ser una instrucción simple o una secuencia de instrucciones si éstas se

[1] Claro que esto es una suposición simplista solo con propósito del ejemplo, lamentablemente el hombre es el único animal que tropieza dos veces con la misma piedra.

agrupan entre las llaves { y }. Si la expresión evalúa `true`, entonces se ejecuta la *<parte if>*, y si da como valor `false` se ejecuta la *<parte else*

Puede faltar la *<parte else>*, y en este caso si la condición evalúa `false` no se hace nada en especial, sino simplemente se continúa la ejecución del flujo de instrucciones.

4.1.1 Instrucciones if anidadas

La *<parte if>* o la *<parte else>* pueden ser a su vez otra instrucción condicional, lo que puede interpretarse como hacer una pregunta a continuación de otra pregunta. Suponga que se quiere mejorar el ejemplo del Listado 4.1 para que una vez que se sepa que las dos veces se ha tecleado lo mismo entonces se escriba el menor de los dos tiempos de tecleo.

El código del Listado 4.2 nos muestra la solución

```
1  Stopwatch crono1 = new Stopwatch();
2  Stopwatch crono2 = new Stopwatch();
3  Console.WriteLine("Teclea tu nombre");
4  crono1.Start();
5  string nombre1 = Console.ReadLine();
6  crono1.Stop();
7  Console.WriteLine("\nVuélvelo a teclear");
8  crono2.Start();
9  string nombre2 = Console.ReadLine();
10 crono2.Stop();
11 if (nombre1 == nombre2)
12 {
13   if (crono1.ElapsedMilliseconds < crono2.ElapsedMilliseconds)
14     Console.WriteLine("\nHola {0}, tu tiempo menor fue de {1} ms",
15                       nombre1, crono1.ElapsedMilliseconds);
16   else
17     Console.WriteLine(
18        "\nHola {0}\n, tu tiempo menor fue de {1} ms",
19             nombre1, crono2.ElapsedMilliseconds);
20 }
21 else
22   Console.WriteLine("\nERROR has tecleado mal. Repite");
```

Listado 4.2 If anidados

En este caso, la *<parte if>* que sigue al `if` de la línea 11 es a su vez otro `if`. En este ejemplo, para mayor claridad, el `if` más interno se ha encerrado entre llaves, aunque no es necesario. Como en este caso la *<parte if>* está formada por una sola instrucción, se puede prescindir del uso de las llaves y escribir el código como se muestra en el Listado 4.3.

```
1  if (nombre1 == nombre2)
2    if (crono1.ElapsedMilliseconds < crono2.ElapsedMilliseconds)
3      Console.WriteLine("\nHola {0}, tu tiempo menor fue de {1} ms",
4                        nombre1, crono1.ElapsedMilliseconds);
5    else
6      Console.WriteLine("\nHola {0}\n, tu tiempo menor fue de {1} ms",
7                       nombre1, crono2.ElapsedMilliseconds);
8  else
9
```

```
10    Console.WriteLine("\nERROR has tecleado mal. Repite");
```

Listado 4.3 If anidado sin necesidad de llaves

Para mayor legibilidad y claridad de los anidamientos, es conveniente dejar las sangrías adecuadas (lo que se conoce como "indentación"[2]). Visual Studio hace esto por nosotros. Las sangrías tienen como objetivo dar una mejor legibilidad al sugerir el anidamiento, pero no tienen un significado sintáctico; el código del Listado 4.3 podría haberse escrito en una única y larga línea[3].

4.1.2 Ejemplo de anidamiento. Hallar el mayor de tres números

Considere que se quiere implementar un método que reciba como parámetros tres valores enteros y devuelva como resultado el valor mayor de los tres (Listado 4.4). Los posibles flujos de ejecución se muestran en la Figura 4-1.

```
1  public static int Maximo(int a1, int a2, int a3)
2  {
3    if (a1 >= a2)
4      if (a1 >= a3) return a1;
5      else return a3;
6    else
7      if (a2 >= a3) return a2;
8      else return a3;
9  }
```

Listado 4.4 Método que devuelve el mayor de 3 números

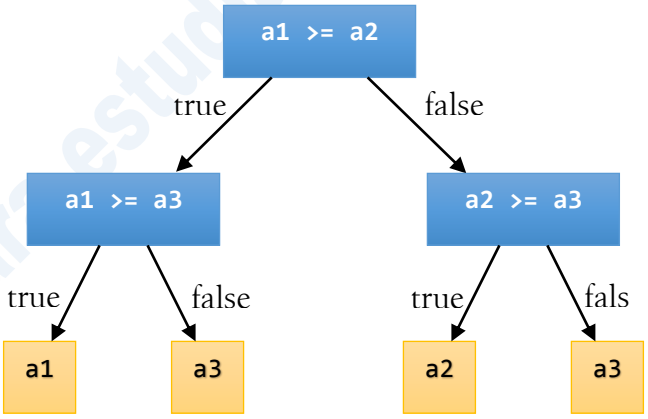

Figura 4-1 Flujos de ejecución para determinar el mayor de tres números

[2] Por ahora un anglicismo pues, el término no existe en español.

[3] Aunque por ello Usted podría reprobar la asignatura o ser despedido como programador por hacerse el gracioso.

4.1.3 Uso de los operadores booleanos para disminuir los anidamientos

. Para la mente humana un razonamiento con anidamientos resulta complejo de seguir. Usando adecuadamente las operaciones lógicas podemos simplificar los anidamientos y darle mayor claridad al código. El método anterior podría haberse implementado de modo más simple como se muestra en el Listado 4.5

```
1 public static int Maximo(int a1, int a2, int a3)
2 {
3   if ((a1 >= a2) && (a2 >= a3)) return a1;
4   else
5     if ((a2 >= a1) && (a2 >= a3)) return a2;
6     else return a3;
7 }
```

Listado 4.5 Operadores booleanos para simplificar los if anidados

Note en este ejemplo que la otra condicional `if` se encuentra anidada dentro de la *<parte else>*. Este es un patrón frecuente que corresponde a estar preguntando por distintas alternativas hasta dar con la adecuada. Es una práctica escribirla como se muestra en el Listado 4.6, disminuyendo las indentaciones y poniendo todas las preguntas al mismo nivel (este es el estilo adoptado en el resto de este libro).

```
1  public static int Maximo(int a1, int a2, int a3)
2  {
3    if ((a1 >= a2) && (a1 >= a3)) return a1;
4    else if ((a2 >= a1) && (a2 >= a3)) return a2;
5    else return a3;
6  }
```

Listado 4.6 Usando el else if

Escriba ahora de forma similar un método para hallar el mayor de 4 números. Usted podrá apreciar cómo se complica más el código si trata de expresar todas las alternativas posibles siguiendo el estilo de solución del Listado 4.6. Sin embargo, se podría escribir de manera más sencilla con un código como el del Listado 4.7.

```
1  public static int Maximo(int a1, int a2, int a3, int a4)
2  {
3    int mayor = a1;
4    if (a2 > mayor) mayor = a2;
5    if (a3 > mayor) mayor = a3;
6    if (a4 > mayor) mayor = a4;
   }
```

Listado 4.7 Hallar el mayor de varios números

Se toma como posible mayor uno de los valores y se deja en una variable (línea 3), y luego se va comparando el valor de esta variable que sirve de "acumulador" con cada uno de los demás valores, cambiando el valor de la variable cada vez que se encuentre un valor

que sea mayor que el que tiene. Este es un algoritmo que servirá de solución cuando se quiera hallar el mayor de una colección de valores.

4.1.4 Determinar la naturaleza de un triángulo

Se quiere escribir un método que reciba tres valores enteros que significan las longitudes de tres lados de un triángulo y que devuelva una cadena que indique si forman un triángulo equilátero, isósceles o escaleno, o que diga que no forman un triángulo.

Recuerde de sus conocimientos de geometría que para que tres valores puedan representar las longitudes de los lados de un triángulo debe cumplirse que cada uno sea menor que la suma de las otros dos (línea 3 del Listado 4.8); si forman triángulo y los tres lados son iguales el triángulo es equilátero, si solo dos son iguales es isósceles, y si los tres son desiguales es escaleno.

```
1   public static string TipoDelTriangulo(int a, int b, int c)
2   {
3     if ((a < b + c) && (b < c + a) && (c < a + b))
4     {
5       if (a == b)
6       {
7         if (b == c) return "Equilátero";
8         else return "Isósceles";
9       }
10      else
11      {
12        if (b == c) return "Isósceles";
13        else
14        {
15          if (a == c) return "Isósceles";
16          else return "Escaleno";
17        }
18      }
19    }
20    else return "No forman un triángulo";
21  }
```

Listado 4.8 Método para clasificar un triángulo según sus lados

La Figura 4-2 muestra el esquema de las alternativas de ejecución del método del Listado 4.8 según los datos de entrada.

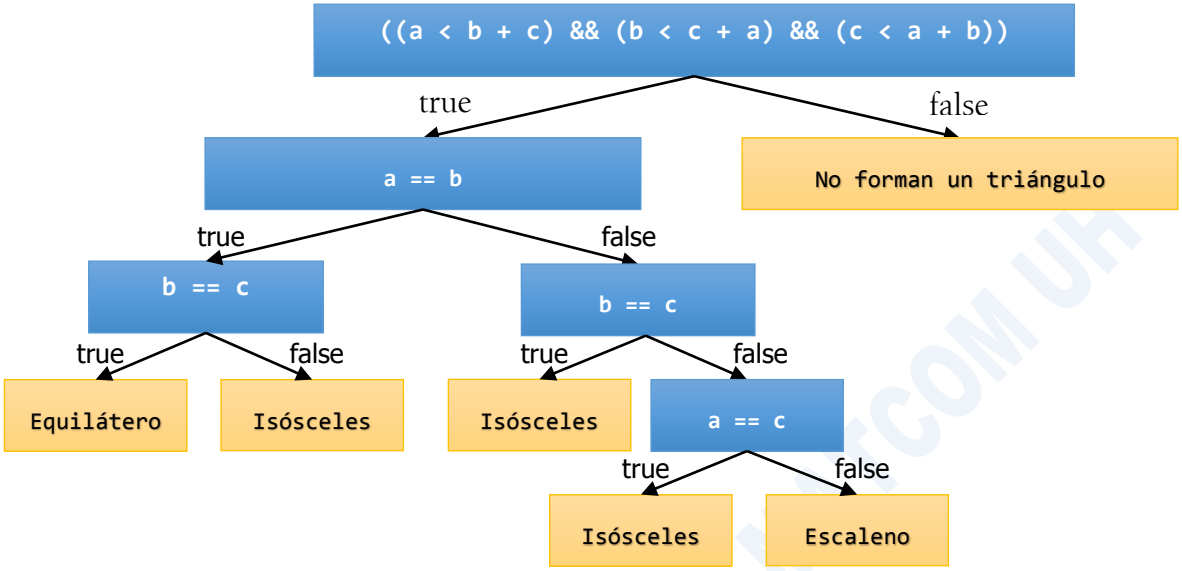

Figura 4-2 Esquema para saber la naturaleza de un triángulo según sus lados

Aunque en este caso no son necesarias, se han utilizado llaves para dar más claridad al agrupar cada *<parte if>*. Sin embargo, si se usan los operadores booleanos podrían evitarse anidamientos y el código del listado podría simplificarse y hacerse mucho más legible, como se muestra en el Listado 4.9.

```
1   public static string TipoDelTriangulo(int a, int b, int c)
2   {
3     if ((a < b + c) && (b < c + a) && (c < a + b))
4     {
5       if ((a == b) && (b == c)) return "Equilátero";
6       else if ((a == b) || (b == c) || (a == c)) return "Isósceles";
7       else return "Escaleno";
8     }
9     else return "No forman triángulo";
10  }
```

Listado 4.9 Uso de los operadores lógicos && y || para simplificar una instrucción condicional

4.1.5 Validación de una fecha correcta

Se quiere escribir un método que determine si tres valores enteros que pueden ser el día, el mes y el año de una fecha, forman realmente una fecha correcta.

Como se explica en el Capítulo 6, este tipo de validación es característico de los métodos "constructores" de una clase. Un método constructor recibe, de quién desea crear una instancia de un objeto, los parámetros para inicializar las componentes necesarias del objeto y debe "validar" si con estos valores se puede realmente tener un objeto según la lógica del mismo. Por ejemplo, una fecha está formada por tres valores enteros, pero no tres enteros cualesquiera forman una fecha.

El Listado 4.10 implementa para ello un método booleano `FormanFecha`. Note que no todos los meses tienen la misma cantidad de días, y que en el caso del mes de febrero hay que tener en cuenta además si el año es bisiesto. El método determina cuántos días tiene el mes (si es que el valor que se le ha dado para el mes está entre 1 y 12, porque de lo contrario devolvería `false`, línea 18 del código) para después poder determinar si el valor que se ha dado al parámetro día cae en el rango de ese mes.

```
1  public bool FormanFecha(int d, int m, int a)
2  {
3    if (a > 0)
4    {
5      int diasEnElMes;
6      if ((m == 4) || (m == 6) || (m == 9) || (m == 11))
7        diasEnElMes = 30;
8      else if ((m == 1) || (m == 3) || (m == 5) || (m == 7) ||
9               (m == 8) || (m == 10) || (m == 12))
10       diasEnElMes = 31;
11     else if (m == 2)
12     {
13       if ((a % 4 == 0) && (a % 100 != 0) || (a % 400 == 0)
14         diasEnElMes = 29;
15       else
16         diasEnElMes = 28;
17     }
18     else return false; //Mes no es correcto
19
20     if ((d >= 1) || (d <= diasEnElMes)) return true;
21     else return false; //Día no es correcto
22   }
23   else return false; //Año no es correcto
24 }
```

Listado 4.10 Verificar si tres enteros forman una fecha

Note que la evaluación en corto circuito hace más eficiente la evaluación de este código porque no hay que evaluar todas las comparaciones.

Observe la expresión booleana utilizada en la línea 13 para determinar si un año es bisiesto. Se considera que un año es bisiesto si es múltiplo de 4 pero no de 100, aunque los múltiplos de 400 también son bisiestos.

Originalmente el calendario que se utilizaba en Roma (que dio origen al actual), no se regía por el movimiento del Sol, y consistía en 300 días divididos en 10 meses de 30 días cada mes. Este calendario no era adecuado para administrar la agricultura, pues no facilitaba interpretar los ciclos de siembra y cosecha que acompañan a las estaciones y al sol en sus movimientos. Es por ello que después de mucho tiempo, cambios y ajustes, se estableció en la época de Julio Cesar el calendario juliano, que constaba de 365.25 días, o sea, cada año era de 365 + 0.25 días, por lo que se establecieron los años de 365 días más un día adicional cada cuatro años para recuperar la acumulación de la fracción de 0.25 días. Sin embargo, luego se determinó que una medición más exacta resulta del año solar era la de 365.2425 días, de modo que cada 4 años no se acumula exactamente un día sino * 4 = 0.97 días y no uno exacto. Es por ello que el 4 de octubre de 1582 se

> desplazó el año oficial en 10 días con respecto al año solar, y el papa Gregorio XIII decretó que el día siguiente sería el 15 de octubre de 1582. Además, para corregir el error se implantó entonces el calendario gregoriano, que dice lo siguiente:
>
> Los años serán de 365 días, y cada cuatro años se agregará un día al mes de febrero, tal como en el calendario juliano (año bisiesto). Pero para corregir el error, cada 25 veces de tales años bisiestos de 366 días (o sea cada 100 años) el año será de 365 días, a menos que este año sea a la vez un múltiplo de 400.
>
> Si se hace el cálculo: 1/4 - 1/100 + 1/400 = 0.2425 da exactamente el error que se acumula por cada año de 365 días. Luego, para verificar que un año es bisiesto (es decir con 29 días en febrero, en lugar de 28), hay que verificar que sea divisible por 4, pero no puede ser divisible por 100, a menos que lo sea por 400.

Dada la anterior aclaración, se puede apreciar que la expresión booleana de la línea 13 en el Listado 4.10 no tiene en cuenta el año a partir del cual comienza a aplicarse el calendario gregoriano. De aquí que la lógica para conocer si un año es bisiesto incluye otras validaciones que pueden ser implementadas en un nuevo método. Esta separación ayuda a poder reutilizar esta lógica en cualquier lugar donde se requiera como el método `FormanFecha`, haciendo énfasis así en una de las premisas del paradigma de programación orientada a objetos, que es la reutilización.

La implementación del método `EsBisiesto` tiene en cuenta entonces si estamos en un año juliano o en un año gregoriano para hacer el cálculo en consecuencia. O sea, que si el año es juliano será bisiesto si es divisible por 4, pero si es un año gregoriano será bisiesto si es divisible por 400, o si no si lo es por 4 pero no por 100. La implementación se muestra en el Listado 4.11.

```
1   private bool EsBisiesto(int a)
2   {
3     if (a < 1582)
4     {
5       if (a % 4 == 0) return true;
6       else return false;
7     }
8     else
9     {
10      if ((a % 400 == 0) || ((a % 4 == 0) && !(a % 100 == 0)))
11        return true;
12      else return false;
13    }
14  }
```

Listado 4.11. Método para determinar si un año es bisiesto

Note la condicional anidada entre las líneas 5 y 6 y la condicional anidada entre las líneas 10 y 12, ambas instrucciones `if` son de la forma:

```
if (<condición>) return true;
else return false;
```

En lugar de preguntar por el resultado de una condición (expresión `bool`) para devolver `true` o `false`, es más simple devolver directamente el valor de dicha expresión.

```
return <condición>;
```

La implementación de `EsBisiesto` puede quedar entonces mucho más simple, como se muestra en el Listado 4.12.

```
1  private bool EsBisiesto(int a)
2  {
3    if (a < 1582) return (a % 4 == 0);
4    else return ((a % 400 == 0) ||
5                 ((a % 4 == 0) && !(a % 100 == 0)));
6  }
```

Listado 4.12. Método EsBisiesto modificado

4.2 Operador Condicional

Existe un patrón para usar una condicional dentro de una expresión, mediante el operador ternario representado por el par de caracteres (`?:`).

Supongamos que se quiere dejar en una variable `a` el mayor de dos valores `b` y `c`. Esto puede hacerse con el código:

```
if (b > c) a = b;
 else a = c;
```

Usando el operador `?:` condicional, esto se podría expresar:

```
a = (b > c) ? b : c;
```

La sintaxis del operador condicional es:

(<condición>) ? *<expresión1>* : *<expresión2>*

lo que significa "si *<condición>* es verdadera, el resultado de aplicar el operador es el valor de *<expresión1>*, si no es el valor de *<expresión2>*".

Para que la sintaxis del operador sea correcta, es necesario que *<expresión1>* y *<expresión2>* sean del mismo tipo (como en el ejemplo anterior, en el que ambas son de tipo entero) y que ambas sean expresiones que devuelvan un valor.

El siguiente ejemplo es incorrecto, porque `Console.WriteLine(b)` es una instrucción y no una expresión que computa un valor

```
(b > c) ? Console.WriteLine(b) : Console.WriteLine(c);
```

Este operador se puede utilizar de forma anidada. La instrucción a continuación deja en la variable `a` al mayor de tres valores `b, c` y `d`:

```
a = (b >= c) ? (b >= d) ? b : d : (c >= d) ? c : d;
```

Recuerde que siempre pueden usarse paréntesis para mejorar la legibilidad:

```
a = (b >= c) ? ((b >= d) ? b : d) : ((c >= d) ? c : d);
```

El `if` anidado del Listado 4.2 puede ahora expresarse en con una sola instrucción `WriteLine` (Listado 4.13).

```
1  Console.WriteLine(
2     "\nHola {0}, tu tiempo menor ha sido de {1} ms",
3
```

```
4       (crono1.ElapsedMilliseconds < crono2.ElapsedMilliseconds) ?
5         nombre1 : nombre2,
        crono1.ElapsedMilliseconds);
```

Listado 4.13 Uso del operador condicional para disminuir if anidados

4.3 La Instrucción Condicional SWITCH

Retomemos el código del Listado 4.10 para determinar si tres enteros pueden formar una fecha; observe la secuencia de `if else`:

```
if ((m == 4) || (m == 6) || (m == 9) || (m == 11))
  //...
else if ((m == 1) || (m == 3) || (m == 5) || (m == 7) ||
         (m == 8) || (m == 10) || (m == 12))
  //...
else if (m == 2)
  //...
else
```

¿Qué tienen en común todas las expresiones `bool` de este código? En todos los casos expresan diferentes alternativas o valores de una misma expresión (en este ejemplo, la expresión formada por la variable `m`). C# tiene una instrucción especial que permite expresar de manera más simple este tipo de situaciones de condicionales (a la vez que evita que la expresión se tenga que evaluar tantas veces). Esta es la **instrucción switch-case**.

Una implementación equivalente a la del Listado 4.10 puede expresarse entonces de manera más simple como se muestra en el Listado 4.14.

```
1  public bool FormanFecha(int d, int m, int a)
2  {
3    if (a > 0)
4    {
5      int diasEnElMes;
6      switch (m)
7      {
8        case 4: case 6: case 9: case 11:
9           diasEnElMes = 30;
10          break;
11       case 1: case 3: case 5: case 7: case 8: case 10: case 12:
12         diasEnElMes = 31;
13         break;
14       case 2:
15          diasEnElMes = (EsBisiesto(a))? 29 : 28;
16          break;
17       default:
18         return false;
19     }
20     return ((d >= 1) || (d <= diasEnElMes)) ? true : false;
21   }
22   else return false;
23 }
```

Listado 4.14 Verificar si tres enteros forman una fecha usando switch-case

La instrucción `switch (m)` indica que a continuación se van a enunciar diferentes casos (de ahí el uso de la palabra reservada `case`) según valores posibles de la variable `m`. La instrucción

```
case 4: case 6: case 9: case 11:
```

indica que si la variable `m` tiene alguno de los valores `4`, `6`, `9` u `11`, entonces se deben ejecutar las instrucciones que siguen hasta el próximo `case`, es decir:

```
diasEnElMes = 30;
break;
```

Cada sección `case` debe terminar con una instrucción de ruptura. El `break` en este caso significa continuar la ejecución en la instrucción que sigue al `switch`; en lugar de `break` también se puede terminar con un `return` (salir del método dentro del que está el código) o `throw` (disparar una excepción).

La instrucción `switch` puede terminar con una instrucción `default`, lo que indica lo que se quiere hacer si ninguna de las alternativas `case` se ha cumplido (en este ejemplo, cuando el valor de la variable `m` no está entre `1` y `12`, es decir, no es un mes posible).

En general, la sintaxis de la instrucción `switch` es como se muestra en el Listado 4.15.

```
switch (<expresión>){
    case <valor1-1>: case <valor1-2>: case <valor1-k>:
        <instrucción1>; <ruptura>;
    case <valor2-1>: case <valor2-2>: case <valor2-l>:
        <instrucción2>; <ruptura>;
    case <valorn-1>: case <valorn-2>: case <valorn-m>:
        <instrucciónn>; <ruptura>;
    [ default: <instrucción>; <ruptura>; ]
}
```

Listado 4.15 Sintaxis de la instrucción switch

Los valores con los que se compara la *<expresión>* están representados por su notación literal, o sea valores explícitos. Por lo tanto, el tipo de la expresión debe ser uno de los tipos simples, o el tipo `string`.

Esta restricción permite realizar algunas mejoras en la expresividad del lenguaje con respecto al chequeo de las condiciones, y algunas otras mejoras en la eficiencia, en cuanto a la ejecución del chequeo.

Será reportado como error de compilación el siguiente fragmento de código:

```
switch (opcion){
    case 1: System.Console.WriteLine("Opción uno");
    case 2: System.Console.WriteLine("Opción dos");
}
```

porque las alternativas `case 1` y `case 2` no terminan con una instrucción de ruptura.

No sería error, por ejemplo:

```
switch (opcion){
    case 1:
```

```
        System.Console.WriteLine("Opción uno");
        break;
      case 2:
        System.Console.WriteLine("Opción dos");
        break;
}
```

En una instrucción `switch` no es obligatorio el uso de la cláusula `default`, al igual que en una condicional `if` no es obligatorio el `else`.

4.3.1 Ejemplo Calculadora

Suponga que se necesita un método (por ejemplo para utilizar en una clase `Calculadora`), que reciba como parámetros dos `int`, representando a los dos operandos, y un `char` representando la operación que se debe realizar. Si se utiliza la instrucción `if`, la implementación de este método queda como se muestra en el Listado 4.16.

```
1  int Calcula(int op1, int op2, char operador)
2  {
3    if (operador == '+') return op1 + op2;
4    else if (operador == '-') return op1 - op2;
5    else if (operador == '*') return op1 * op2;
6    else if (operador == '/') return op1 / op2;
7    else throw new Exception("Operador incorrecto");
8  }
```

Listado 4.16 Calculadora usando if-else

Note como está presente aquí un patrón similar al anterior, ya que todas las condiciones son comparaciones con la variable `operador`. Si se utiliza la instrucción `switch`, la implementación se transforma en la mostrada en el Listado 4.17.

```
1   int Calcula(int op1, int op2, char operador)
2   {
3     switch (operador)
4     {
5       case '+':
6         return op1 + op2;
7       case '-':
8         return op1 - op2;
9       case '*':
10        return op1 * op2;
11      case '/':
12        return op1 / op2;
13      default:
14        throw new Exception("Operador incorrecto");
15    }
16  }
```

Listado 4.17 Calculadora usando switch-case

Se puede observar que de esta manera se expresa de forma más clara la intención de discernir el valor de la expresión en el `switch` (o sea, `operador`) de entre varios valores (`'+', '-', '*', '/'`).