3 TIPOS DE DATOS Y OPERACIONES BÁSICAS

Even the little things can change the course of the future.

J.R.R. Tolkien (The Lord of the Rings)

Todo lenguaje de programación, C# entre ellos, provee un conjunto de tipos de datos básicos, por lo general integrados sintácticamente en la notación del lenguaje (de ahí el término en inglés *built-in*). Estos tipos básicos forman parte del *framework* y sirven de base para definir, según los recursos existentes en el lenguaje de programación, otros tipos de datos más complejos y especializados de acuerdo con las necesidades de las aplicaciones.

En el código del Listado 3.1(ver Capítulo 2) se están usando cuatro tipos de datos: el tipo `Console`, que tiene las funcionalidades para leer y escribir a través de la ventana de consola; el tipo `Stopwatch`, que tiene las funcionalidades para medir el tiempo imitando a un cronómetro; el tipo `string`, que representa cadenas de caracteres con los textos que queremos visualizar, y el tipo numérico entero largo `long` (que es en este caso el tipo de valor devuelto por la propiedad `ElapsedMilliseconds` para dar una cantidad en milisegundos.

```
1   class Program
2   {
3     static void Main(string[] args)
4     {
5       Stopwatch crono = new Stopwatch();
6       Console.WriteLine("Hola, teclea tu nombre por favor");
7       crono.Start();
8       string nombre = Console.ReadLine();
9       crono.Stop();
10      Console.WriteLine("Hola " + nombre +
11                        ". Bienvenido a la Programación con C#");
12      Console.WriteLine("Te has demorado " + crono.ElapsedMilliseconds +
13                        " ms en teclear tu nombre");
14    }
15  }
16
```

Listado 3.1. Primeros tipos utilizados

Los tipos `Console` y `Stopwatch` son tipos definidos por clases (la definición de clases se ve en el Capítulo 6). Los tipos `string` y `long` son tipos básicos integrados sintácticamente en la notación del lenguaje y sus nombres son por tanto palabras reservadas en el lenguaje, con lo que se evitan confusiones al compilador impedir que a esos nombres se les dé un uso diferente.

3.1 Tipos numéricos

Los tipos numéricos se dividen en dos grupos: los que representan a números enteros (números sin parte fraccionaria) y los que representan a números racionales (con parte fraccionaria).

3.1.1 Tipos enteros int y long

Los tipos `int` y `long` se usan para representar a números enteros. La diferencia entre ambos radica en la cantidad de memoria que usan para su representación y por tanto en la magnitud de los números que pueden representar. El tipo `int` usa 32 bits de memoria y el tipo `long` utiliza 64 bits; es en la práctica el programador quien debe decidir cuál es el tipo de su conveniencia según el tamaño de las magnitudes numéricas con las que necesita trabajar. Por ejemplo, para poder expresar una gran cantidad de milisegundos los desarrolladores de la clase `Stopwatch` definieron que la propiedad `ElapsedMilliseconds` devolviese un valor de tipo `long`.

Los valores concretos de los tipos `int` y `long` se expresan con una notación similar a la utilizada comúnmente en las Matemáticas, es decir, mediante una secuencia de dígitos que puede estar precedida por un signo `+` (que sería redundante) para los positivos o por un signo `-` para los negativos.

Los tipos `int` y `long` tienen las propiedades `MaxValue` y `MinValue` que nos dan el valor entero mayor y el valor entero menor para cada representación

`int.MaxValue` es `2147483647`

`long.MaxValue` es `9223372036854775807`

`int.MinValue` es `-2147483648`

`long.MinValue` es `-9223372036854775808`

Puesto que el tipo `Stopwatch` utiliza un `long` para representar `ElapsedMilliseconds`, eso permitiría que una aplicación que estuviese usando un tal cronómetro pudiese estar ejecutando por unos 292471208 años y dar mediciones correctas

3.1.2 Operaciones y Expresiones

Sobre los valores aritméticos se pueden aplicar las **operaciones** aritméticas de suma, resta, multiplicación y división, denotadas respectivamente por los **operadores** `'+'`, `'-'`, `'*'` y `'/'`. En el caso de la operación de división, si se aplica a operandos enteros el resultado

es el **cociente entero** de la división; es decir, el resultado de `9/4` es `2`. Como complemento de esta operación de división, se tiene también la operación `'%'` que nos da el **resto** de la división; es decir, `9%4` es `1`.

La combinación de operadores y operandos permite formar **expresiones**. Cuando se combinan varios operandos y operadores en una expresión, primero se aplican los operadores de mayor prioridad. Los operadores `'*'`, `'/'` y `'%'` tienen la misma prioridad, que es mayor que la de los operadores `'+'` y `'-'`, que tienen igual prioridad. Cuando los operadores son de igual prioridad, entonces las operaciones se aplican de izquierda a derecha:

`4 * 3 + 2` es `14` porque se hace primero `4 * 3` y a eso se le suma `2`.

`4 + 3 * 2` es `10` porque se hace primero `3 * 2` y a eso se le suma `4`.

`2 + 3 * 4 - 1` es `13` porque primero se hace `3 * 4`, luego quedaría `2 + 12 – 1`, lo que da como resultado `13`.

Cuando se quiere aplicar otro orden de prioridad, se pueden utilizar paréntesis para agrupar las operaciones; en ese caso, primero se evaluarán las expresiones entre paréntesis:

`(2 + 3) * (4 - 1)` es `15`.

Supongamos que se quiere mejorar la funcionalidad del código del Listado 3.1. Además de dar el tiempo que se demora en teclear, se quiere dar también la velocidad de tecleo. Esto se pudiera calcular dividiendo la cantidad de caracteres del nombre entre el tiempo que se ha tardado. La cantidad de caracteres del nombre tecleado se puede obtener aplicando la propiedad `Length` al operando de tipo `string`.

El código del Listado 3.2 nos muestra tres variantes para hacer esto.

```
1    Console.WriteLine("Entra tu nombre");
2
3    Stopwatch crono = new Stopwatch();
4    crono.Start();
5
6    string nombre = Console.ReadLine();
7    crono.Stop();
8
9    Console.WriteLine("Hola {0} te has demorado {1}
10       ms en teclear tu nombre", nombre, crono.ElapsedMilliseconds);
11
12   Console.WriteLine("Tecleando a una velocidad de {0}
13       caracteres por     segundo",
14       nombre.Length / crono.ElapsedMilliseconds * 1000);
15   Console.WriteLine("Tecleando a una velocidad de {0}
16       caracteres por segundo",
         nombre.Length / (crono.ElapsedMilliseconds / 1000));
     Console.WriteLine("Tecleando a una velocidad de {0}
         caracteres por segundo",
         (double)nombre.Length / crono.ElapsedMilliseconds * 1000);
```

Listado 3.2. Midiendo la velocidad de tecleo

Si se ejecuta este código se obtiene el siguiente resultado

```
Entra tu nombre
maria
Hola maria te has demorado 2757 ms en teclear tu nombre
Tecleando a una velocidad de 0 caracteres por segundo
Tecleando a una velocidad de 2 caracteres por segundo
Tecleando a una velocidad de 1.81356546971346 caracteres por segundo
```

Note que se obtienen tres resultados diferentes.

En la *primera* variante se ha hecho:

```
nombre.Length / crono.ElapsedMilliseconds * 1000
```

Como la cantidad de milisegundos debe ser una magnitud mucho mayor que la longitud de la cadena tecleada y esta es una división entera, entonces el cociente entero de la división

`nombre.Length / crono.ElapsedMilliseconds` es `0`, lo que multiplicado por `1000` para llevarlo a cantidad de caracteres por segundo dará de todos modos `0`.

En la *segunda* variante se ha hecho:

```
nombre.Length / (crono.ElapsedMilliseconds / 1000)
```

Es decir, se han utilizado paréntesis para forzar que se haga primero la división entre `1000` y llevar el tiempo a segundos, pero recuerde que la división dará el cociente entero, lo que hace perder en exactitud. Luego la longitud de la cadena tecleada se divide entre esa cantidad de segundos y de esa división se da el cociente entero. De modo que el resultado de caracteres por segundo será dado como un número entero (`2` en este ejemplo), lo cual no es muy exacto.

El mayor problema en ambas variantes es que se ha utilizado la división entera, que da cociente entero, cuando realmente la parte fraccionaria tiene un significado importante en este ejemplo. La tercera variante se verá en la siguiente sección.

3.1.3 Los tipos float y double

Para representar valores numéricos que puedan tener parte fraccionaria se tienen los tipos `float` (que se representa en 32 bits) y `double` (que se representa en 64 bits).

Los números `float` y `double` se representan en lo que se conoce como aritmética de "punto flotante", en la que se dedica parte de los bits a representar los dígitos significativos (lo que se conoce como **mantisa**) y parte de los bits a representar un exponente en base 10 que indica el lugar del punto en el número.

Felizmente, hay un estándar establecido por la IEEE para esta representación y para las operaciones aritméticas sobre este tipo de valores, lo cual hoy día es implementado por la mayoría de los CPU y por tanto incluido en la mayoría de los lenguajes de programación actuales

Los valores de estos tipos se representan mediante una secuencia de dígitos para la parte entera, un punto y otra secuencia de dígitos para la parte fraccionaria; por ejemplo `23.45`.

```
<secuencia de dígitos parte entera>.<secuencia de dígitos parte fraccionaria>
```

Al igual que con los tipos `int` y `long`, los valores mínimos y máximos representables para float y double se obtienen mediante las propiedades `MinValue y MaxValue` aplicadas a esos tipos:

`float.MaxValue` es `3.402823E+38`

`double.MaxValue` es `1.79769313486232E+308`

`float.MinValue` es `-3.402823E+38`

`double.MinValue` es `-1.79769313486232E+308`

La notación utilizando el carácter `E` para denotar el exponente es también una forma válida de expresar los valores `float` y `double`, y es útil para expresar números muy grandes o muy pequeños sin tener que escribir demasiadas cifras significativas:

`3.402823E+38` es lo mismo que `3.402823 * Math.Pow(10, 38)`

La *tercera* variante del código del Listado 3.2 hace

```
(double)nombre.Length / crono.ElapsedMilliseconds * 1000
```

La operación `(double)nombre.Length` se denomina una operación de **coerción** (casting), que quiere decir en este caso que el valor entero resultado de `nombre.Length` se convierta a la representación de `double`. De esta manera, al tener uno de los operandos de tipo `double` entonces la operación de división / ya no se interpreta como la operación de división entre enteros, sino que será una división que dará un resultado con parte fraccionaria.

Usted debe tener presente que cuando se trabaja con valores `float` o `double` los dígitos significativos pueden no ser exactos. Los problemas de precisión numérica, y lidiar con los errores de aproximación en los cálculos numéricos, es toda una disciplina dentro de la Matemática y la Ciencia de la Computación. Abordar este tema se sale del alcance de las pretensiones de este libro.

3.1.4 Desbordamiento (overflow)

El resultado de una operación aritmética entre enteros puede ser un valor no representable por las limitaciones de la cantidad de bits dedicada a representar cada número. Por ejemplo, no se puede representar como valor de tipo `int` el resultado de hacer

```
int k = 1; ...
Console.WriteLine(int.MaxValue + k);
```

Ni el compilador de C# ni el framework .NET pretenden detectar este tipo de situaciones.

Producto del desbordamiento en la representación en bits el resultado de una operación como la anterior es -2147483648 que sería la que corresponde al menor negativo que se puede representar en 32 bits. Esto es lo que se conoce como **overflow**.

Es responsabilidad del programador cuidarse de este tipo de situaciones para que no interprete como valores numéricos correctos cálculos que realmente son erróneos.

Si se prevé que las operaciones con `int` pudieran provocar un desbordamiento, entonces sería mejor usar el tipo `long`. Por ejemplo, si se hace

```
int k = 3; ...

Console.WriteLine((long)int.MaxValue + k);
```

se obtiene como resultado `2147483650.`

Claro si se hiciera `Console.WriteLine(long.MaxValue + k);` daría de igual modo error de debordamiento porque se estaría queriendo sumar 1 al mayor `long` representable.

Por cuestiones de eficiencia no es práctico que el hardware o el código generado por el compilador de C# se protejan de este tipo de errores de desbordamiento. El CPU tendría que estar comprobando preventivamente antes de hacer cada operación aritmética si el resultado no se va a desbordar, lo que aumentaría el tiempo de ejecución de los programas afectando el rendimiento. De modo que tenga cuidado, un error de desbordamiento podría no ser detectado y traer como consecuencia que se interpreten como correctos resultados erróneos; o ser detectado pero no en el lugar y momento en que exactamente se originó

3.1.5 El tipo decimal

El tipo de datos `decimal` es un tipo de dato numérico con precisión para representar hasta 28 cifras decimales significativas; fíjese que el valor mayor que se representa con el tipo `long` tiene solo 19 cifras decimales.

El valor de `long.MaxValue` es `9223372036854775807,`

mientras que el valor de `decimal.MaxValue` es `79228162514264337593543950335`

Este tipo `decimal` maneja el punto decimal, es decir permite representar números con parte fraccionaria, pero con lo que se conoce como semántica de punto fijo, a diferencia de los tipos `float` y `double` que manejan semántica de punto flotante y que permiten manejar números muy grandes (con exponentes positivos muy grandes) o muy pequeños (con exponentes negativos), pero al precio de perder en cifras significativas. El tipo `decimal` es apropiado para cálculos donde los errores de "redondeo" debido a la representación en punto flotante sean inaceptables; por ejemplo, en transacciones monetarias que hacen conversiones entre diferentes monedas o donde se calculan intereses e impuestos.

El lector podrá preguntarse ¿entonces para qué el `float` y el `double,` por qué no usar siempre decimal? Como por lo general no hay correspondencia directa en el hardware-CPU para este tipo decimal, las operaciones se emulan por software, lo que se traduce en menos velocidad del código ejecutable. De modo que utilice el tipo `decimal` cuando realmente sea necesario por la precisión que requieren sus resultados, pero cuídese de aplicarlo cuando requiera mucho cálculo intensivo

La asignación `long k = 1111111111111111111;` es correcta,

pero `long k = 11111111111111111111;` (con un dígito más) provoca error de compilación porque el valor numérico de la parte derecha no puede representarse en el espacio de un `long`.

Para que los valores literales se interpreten como de tipo `decimal,` deben terminar con el sufijo `m` o `M.` Las asignaciones

```
decimal d = 99999999999999999999999999m;
```
y

```
decimal pi = 3.14159265358979323846264338327m;
```
son correctas.

3.2 Asignación

Sin duda alguna, por sus conocimientos de Matemática, el lector estará familiarizado con especificaciones como $x = 2$ o $y = m * x + n$. Estas especificaciones indican que la entidad x (que se le suele llamar variable x) tiene valor 2 y que la entidad y tiene como valor el resultado del cálculo expresado por $m * x + n$.

Para expresar un efecto similar, en C# se tiene lo que se conoce como **instrucción de asignación**. La operación de asignación permite asociar un valor a una variable (o a una propiedad de un objeto, como se en el Capítulo 6). Para hacer una **asignación** se escribe:

```
<entidad parte_izquierda> = <expresión parte_derecha>
```

Esto indica que a la *<entidad parte_izquierda>* se le "asigna" (se asocia con) el valor resultado de evaluar la *<expresión parte_derecha>*. A nivel interno en el funcionamiento del código generado por el compilador, esto significa que la parte izquierda tendrá asociada una zona de la memoria donde se "guardará" el valor resultado de evaluar la expresión.

La parte izquierda de una asignación debe ser una **variable** o una propiedad que tenga definida una parte `set` (ver Capítulo 6) que esté declarada de un tipo "compatible" con el tipo del resultado de la parte derecha. Por ahora, asuma que tipo compatible quiere decir que sea el "mismo tipo" que el tipo del valor calculado en la parte derecha. Sobre esta "compatibilidad de tipos" se trata en los Capítulos 10 y 11, que estudian los conceptos de herencia y las jerarquías de tipos.

En el código de Listado 3.1 ya se han hecho asignaciones. Cuando en la línea 8 se hace:

```
string nombre = Console.ReadLine();
```

se le está asignando a la variable `nombre,` que ha sido declarada de tipo `string`, el valor de la expresión de la parte derecha, que en este caso es el valor de tipo `string` devuelto por el método `ReadLine` al leer los caracteres que se han tecleado hasta teclear "Enter".

Cuando en la línea 5 se hace:

```
Stopwatch crono = new Stopwatch();
```

se le está asignando a la variable `crono` el valor de la parte derecha, que en este caso es el objeto de tipo `Stopwatch` que se ha creado al hacer `new Stopwatch()`.

Una asignación tiene como objetivo conservar un valor computado en un momento y lugar del código para que pueda ser utilizado en otro momento y lugar a través del nombre de la entidad en donde se ha guardado.

De algún modo esto "emula" nuestra memoria: guardamos información para luego acceder a ella. Guardar y acceder a la información guardada forman parte de la naturaleza básica de la mayoría de las aplicaciones computacionales

El código del Listado 3.3 lee una cadena de texto que debe corresponder a la edad de la persona, luego la convierte a un valor de tipo numérico y escribe el resultado de sumarle `10`.

```
1    Console.WriteLine("Entra tu edad");
2
3    string s = Console.ReadLine();
4    int edad = Int32.Parse(s);
     Console.WriteLine("Tu edad dentro de 10 años es {0}", edad + 10);
```

Listado 3.3. Leer la edad y sumarle 10

La asignación:

```
string s = Console.ReadLine();
```

lee una cadena de texto y le asigna ese valor a la variable `s`, que ha sido declarada de tipo `string`. Luego la asignación:

```
int edad = int.Parse(s);
```

convierte el valor de la cadena `s` a un valor de tipo `int` y lo asigna a la variable `edad`.

Finalmente, la instrucción:

```
Console.WriteLine("Tu edad dentro de 10 años es {0}", edad + 10);
```

suma `10` al valor guardado en la variable `edad` y lo escribe (realmente el valor computado por la expresión `edad + 10` se le ha pasado como parámetro al método `WriteLine`).

Realmente en este caso el código podía haberse escrito:

```
Console.WriteLine("Tu edad dentro de 10 años es {0}",
                  int.Parse(Console.ReadLine()) + 10);
```

sin necesidad de usar las variables `s` y `edad` y las asignaciones respectivas. Sin embargo, puede que así el código le resulte menos legible.

El compilador de C# exige que toda variable tenga que aparecer en la parte izquierda de una asignación antes de poder usarse como operando dentro del código de una expresión. Un código como

```
int k;
int j = k*2;
```

es reportado error por el compilador porque la variable `k` aparece en la expresión `k*2` cuando realmente no hay ningún código antes que le asigne un valor.

Los primeros lenguajes de programación no hacían este tipo de verificaciones, lo que podía resultar en errores difíciles de detectar. En un caso como éste, al evaluar una expresión como `k*2`, donde a la variable `k` no se le ha asignado previamente ningún valor, se tomaba como valor entero de `k` la información que casualmente estuviera en los bits de la memoria correspondiente a `k`

3.3 Funciones numéricas

.NET ofrece mediante la clase `Math` una amplia variedad de las funciones matemáticas conocidas. El código del Listado 3.4 lee las coordenadas de dos puntos en el espacio y calcula la distancia entre los mismos aplicando el Teorema de Pitágoras.

```
1   Console.WriteLine("\nEntra coordenada x del primer punto");
2
3   double x1 = double.Parse(Console.ReadLine());
4   Console.WriteLine("\nEntra coordenada y del primer punto");
5
6   double y1 = double.Parse(Console.ReadLine());
7
8
9   Console.WriteLine("\nEntra coordenada x del segundo punto");
10
11  double x2 = double.Parse(Console.ReadLine());
12  Console.WriteLine("\nEntra coordenada y del segundo punto");
    double y2 = double.Parse(Console.ReadLine());


    Console.WriteLine("La distancia entre los dos puntos es {0}",
         Math.Sqrt( (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)));
```

Listado 3.4. Calculando la distancia entre dos puntos

Se ha utilizado aquí el método `Sqrt` de la clase `Math`, que calcula la raíz cuadrada de un valor numérico.

La expresión a la que se le calcula la raíz cuadrada podría haberse escrito utilizando el método `Pow`, que eleva un número a una potencia:

```
Console.WriteLine("La distancia entre los dos puntos es {0}",
                  Math.Sqrt(Math.Pow(x2 - x1, 2) + Math.Pow(y2 - y1, 2)));
```

Invitamos al lector a explorar los métodos numéricos que hay en la clase `Math` según los vaya necesitando, ya que sería muy aburrido presentarlos todos.

3.3.1 Otros tipos numéricos

Tabla 3.1 muestra todos los tipos numéricos disponibles en C#.

Tipo	Descripción	Ejemplo

`sbyte`[1]	Tipo entero de 8 bits con signo, rango de valores de -128 a 127	`sbyte val = 12;`
`Short`	Tipo entero de 16 bits con signo, rango de valores de -32768 a 32767	`short val = 12;`
`Int`	Tipo entero de 32 bits con signo, rango de valores de –2147483648 a 2147483647	`int val = 12;`
`long`	Tipo entero de 64 bits con signo, rango de valores de –9223372036854775808 a 9223372036854775807	`long val1 = 12;` `long val2 = 34L;`
`byte`	Tipo entero de 8 bits sin signo, rango de valores de 0 a 255	`byte val = 12;`
`ushort`	Tipo entero de 16 bits sin signo, rango de valores de 0 a 65535	`ushort val = 12;`
`uint`	Tipo entero de 32 bits sin signo, rango de valores de 0 a 4294967295	`uint val1 = 12;` `uint val2 = 34U;`

`ulong`	Tipo entero de 64 bits sin signo, rango de valores de 0 a 18446744073709551615	`ulong val1 = 12;`
		`ulong val2 = 34U;`
		`ulong val3 = 56L;`
		`ulong val4 = 78UL;`

`float`	Tipo de punto flotante de simple precisión (32 bits), rango aproximado de valores de 1.5×10^{-45} a 3.4×10^{38} con una precisión de 7 dígitos.	`float val = 1.23F;`
`double`	Tipo de punto flotante de doble precisión (64 bits), rango aproximado de valores de 5.0×10^{-324} a 1.7×10^{308} con una precisión de 15-16 dígitos	`double val1 = 1.23;` `double val2 = 4.56D;`
`decimal`	Tipo de datos de 128 bits, rango de valores de 1.0×10^{-28} a aproximadamente 7.9×10^{28} con 28-29 dígitos significativos.	`decimal val = 1.23M;`

Tabla 3.1 Tipos numéricos

3.4 Métodos

Uno de los recursos más importantes de abstracción en la programación es el concepto de **método**[2]. Un método agrupa un conjunto de instrucciones a las que se les da un **nombre** y cuya ejecución puede llevarse a cabo **invocando** (**llamando**) al método a través de ese nombre desde distintas partes del código. Para su ejecución, un método puede necesitar de parámetros que deben ser suministrados por quien llama al método. La ejecución de las instrucciones que conforman el método pueden dar (retornar) como resultado un valor; a estos métodos se les llama **métodos que devuelven un valor** y al tipo de ese valor se le llama **tipo del método**. Se dice que el método **retorna** el valor para que sea utilizado

[1] [1] La "s" (de `sbyte`) viene del inglés *signed* (con signo) y la "u" (de `ushort`, `uint`, `ulong`) viene del inglés *unsigned* (sin signo). Lamentablemente, para mayor uniformidad con los restantes tipos debió utilizarse el nombre `ubyte` para los enteros de 8 bits sin signo y `byte` para los enteros en 8 bits con signo, pero se hizo lo contrario ☹

[2] También llamado procedimiento (procedure) o función (function) en otros lenguajes de programación; o subrutina si nos remontamos más atrás en el tiempo.

por quien llamó al método. Un método puede realizar acciones pero que no desembocan explícitamente en un valor; en este caso se dice que este es un método que no retorna ningún valor o que retorna `void` (`void` es la palabra reservada utilizada por C# para tales fines).

En la mayoría de los lenguajes denominados **orientados a objeto**, en nuestro caso C#, la definición de un método tiene que estar ubicada dentro de una clase. El método puede ser invocado desde cualquier parte del código de la misma clase a través del nombre, o puede ser invocado desde otra clase precediendo el nombre del método con el nombre de la clase cuando el método es un **método de clase** (en el caso de C#, su definición estará precedida por la palabra `static`). Una clase también puede tener **métodos de instancia**, los que requieren ser invocados a través de una instancia de la clase. En los ejemplos que se han presentado hasta ahora podemos apreciar estas clases de métodos.

El primer método utilizado (`WriteLine`) para dar un saludo es un método de la clase `Console`. A este método hay que pasarle un parámetro de tipo `string` que es lo que el método escribirá por la ventana de consola. El método `WriteLine` no devuelve explícitamente ningún valor, sino que se limita a escribir el texto. La definición del método dentro de la clase `Console` sería entonces algo como:

```
public class Console {
  public static void WriteLine(string s)
  {
    //...instrucciones del método
 }
  public static string ReadLine()
  {
    //...instrucciones del método
 }
  //...resto de la definición de la clase Console
}
```

Para invocar a este método se debe escribir entonces

```
Console.WriteLine("Hola, Teclea tu nombre por favor");
```

Es decir, hay que pasarle un parámetro de tipo `string` porque es lo que requiere el método, y como el método es un método de clase (ya que tiene la especificación `static`) y ha sido invocado desde fuera de la clase, la llamada debe estar precedida por el nombre de la clase (en este caso `Console`).

El método `ReadLine` es también un método `static` de la clase `Console`, pero es un método que devuelve un valor de tipo `string`, por lo que debe ser invocado en un contexto en que se use un valor de tipo `string`, como ha sido en este ejemplo en que el valor leído se asigna a la variable `nombre`. Note que este método `ReadLine` no requiere de parámetros, por lo que la lista de parámetros entre los paréntesis en la definición es vacía. La instrucción de asignación:

```
string nombre = Console.ReadLine();
```

invoca al método sin pasarle ningún parámetro, el valor `string` devuelto por este es asignado a la variable `nombre`.

La clase `Stopwatch` (Listado 3.5) que se ha usado en estos primeros ejemplos, tiene, entre otros, los métodos de instancia `Start` y `Stop`:

```
public class Stopwatch {
  public void Start()
  {
     //...instrucciones del método
 }
  public void Stop()
  {
     //...instrucciones del método
 }
 //...resto de la definición de Stopwatch
}
```

Listado 3.5 Esbozo de la clase Stopwatch

En este caso, los métodos `Start` y `Stop` son métodos de instancia (note que su definición no está precedida por la palabra `static`), por lo que deben ser invocados a través de instancias (objetos) del tipo de la clase. Ese es el caso cuando se hizo:

```
Stopwatch crono = new Stopwatch();
Console.WriteLine("Hola, teclea tu nombre por favor");
crono.Start();
string nombre = Console.ReadLine();
crono.Stop();
```

Se creó una instancia de `Stopwatch` al hacer `new Stopwatch()`, esa instancia se le asignó a la variable `crono` y luego a través de esa instancia se han invocado los métodos `Start` y `Stop`. Note que estos métodos son `void`, es decir, que no devuelven ningún valor sino que provocan alguna acción en la instancia ("echar a andar" y "parar" el cronómetro, respectivamente).

3.4.1 Propiedades

Las **propiedades** son una suerte de métodos que devuelven un valor y que no requieren de parámetros para ser invocados. Ese es el caso de la propiedad `ElapsedMilliseconds` de la clase `Stopwatch` y que ha sido utilizada en estos ejemplos.

Esta propiedad se define en la forma:

```
public class Stopwatch {
```

```
  public long ElapsedMilliseconds
  {
    get {
     //...
  }
  }
  //...
}
```

Cuando se ejecuta:

```
Console.WriteLine("Te has demorado " + crono.ElapsedMilliseconds +
                  " ms en teclear tu nombre");
```

se está llamando al método `WriteLine` y se le está pasado como parámetro el resultado de evaluar la expresión

```
"Te has demorado " + crono.ElapsedMilliseconds + " ms en teclear tu nombre"
```

Aquí el operador `+` es la operación de concatenación de cadenas. Note que se está usando la propiedad `ElapsedMilliseconds` a través de la instancia `crono`. Esta propiedad devuelve un valor de tipo `long`, pero como se usado en la concatenación con un `string` este valor `long` es convertido implícitamente a `string`.

Las propiedades se estudian con más detalle en el Capítulo 6.

3.4.2 Definiendo un método numérico

Retomemos el ejemplo del Listado 3.4. Vamos a definir un método `Distancia` para calcular la distancia entre dos puntos. Un tal método requiere de 4 parámetros de tipo `double`, que representarán las coordenadas de los dos puntos, y devuelve un valor de tipo `double`, que es la distancia entre los dos puntos.

El código del Listado 3.6 define un tal método `Distancia` como método estático de la propia clase `Program`; por tanto, su invocación desde el método `Main` no requiere que esté precedida por el nombre de la clase.

```
1   using System;
2
3   namespace WEBOO.Programacion
4   {
5
6     class Program
7     {
8
9       static double Distancia(double x1, double y1, double x2, double y2)
10      {
11
12        return Math.Sqrt(Math.Pow(x2 - x1, 2) + Math.Pow(y2 - y1, 2)) );
13      }
14
15      static void Main(string[] args)
16
```

```
17        {
18          Console.WriteLine("\nEntra coordenadas x del primer punto");
19          double p1_x = double.Parse(Console.ReadLine());
20          Console.WriteLine("\nEntra coordenadas y del primer punto");
21          double p1_y = double.Parse(Console.ReadLine());
22          Console.WriteLine("\nEntra coordenadas x del segundo punto");
23          double p2_x = double.Parse(Console.ReadLine());
24          Console.WriteLine("\nEntra coordenadas y del segundo punto");
            double p2_y = double.Parse(Console.ReadLine());
            Console.WriteLine("La distancia entre los dos puntos es {0}",
                              Distancia(p1_x, p1_y, p2_x, p2_y));
        }
      }
    }
```

Listado 3.6. Método Distancia

Note la instrucción `return` que se ha utilizado dentro de la definición del método `Distancia`. Si un método devuelve un valor de un determinado tipo, como en este caso `Distancia`, que devuelve un valor de tipo `double`, entonces la ejecución del método debe terminar utilizando una instrucción `return` que retorne un valor de dicho tipo, lo que en este ejemplo es el valor calculado por la expresión

```
Math.Sqrt(Math.Pow(x2 - x1, 2) + Math.Pow(y2 - y1, 2))
```

3.5 El tipo string y el tipo char

Las **cadenas de caracteres**, expresadas en C# mediante el tipo de datos `string`, son secuencias de símbolos (conocidos por caracteres y representados por el tipo simple `char`) que se emplean para representar la información simbólica (no numérica) involucrada en una aplicación. Los primeros ejemplos del uso de cadenas se ven en el Capítulo 2, donde se usan las cadenas de caracteres para dar la salida en la ventana de consola a los resultados en forma de un texto. Este fue el caso del Listado 3.7.

```
1   namespace WEBOO.Programacion
2   {
3     class PrimerPrograma
4     {
5       static void Main(string[] args)
6       {
7      //Escribir mensaje de saludo
8      System.Console.WriteLine(
9        "Hola, bienvenido a la Programación con C#");
```

```
    }
     }
}
```

Listado 3.7. Ejemplo del primer programa

La secuencia de símbolos `"Hola, bienvenido a la Programación con C#"` que debe escribirse entre doble comillas, es una cadena (es decir, un objeto de tipo `string`).

La versión mejorada de este programa ya en del Listado 3.1 incluye además una instrucción para leer una cadena que se escriba en la ventana de consola. En este caso, el nombre `ReadLine` significa que al teclear la cadena ésta debe terminar en un cambio de línea (haber pulsado la tecla *Enter*)

Ambos ejemplos ilustran de manera elemental la amplia utilidad del tipo `string` para la entrada y salida de información en forma de texto (y por tanto "legible" al ojo humano). La información textual que manejan la mayoría de las aplicaciones puede ser representada de manera natural usando el tipo `string`.

Usando el tipo `string` se pueden declarar variables para almacenar cadenas de caracteres, a las que además se le pueden aplicar todos los métodos y propiedades definidas en la clase `String`, muchos de los cuales se verán a continuación en este epígrafe. Hay métodos y propiedades para saber la longitud de la cadena, examinar cada uno de los caracteres que la conforman, comparar la cadena con otra según el orden alfabético, buscar y extraer subcadenas, crear copias de una cadena con todos sus caracteres en minúsculas o mayúsculas, entre otros. A su vez, muchos métodos y propiedades de otras clases de la biblioteca BCL trabajan con parámetros de tipo `string` y/o devuelven resultados de tipo `string`. Ejemplo de ello son precisamente los dos métodos estáticos `WriteLine` y `ReadLine` de la clase `Console`.

3.5.1 String como objeto

Como ya se ha dicho, los datos de tipo `string` son también objetos, y como tales, responden a las llamadas a los métodos y propiedades definidas por su clase. Sin embargo, por su amplia utilización, los `string` cuentan dentro del lenguaje C# con ciertas "bondades" sintácticas adicionales que hacen más fácil su uso. Una de estas facilidades es la posibilidad de denotar directamente (al igual que en los valores numéricos) valores de tipo `string`.

3.5.1.1 Literales

Las cadenas literales son secuencias de caracteres enmarcadas entre comillas (`"..."`) que se escriben en el código para denotar el valor concreto de un `string`.

Al ser considerados como objetos, se les pueden aplicar directamente métodos y propiedades al igual que a una variable común. Por ejemplo, es válido en C# hacer:

```
string str = "String como objeto".ToUpper();
```

Y obtener en la variable `str` un objeto `string` con la cadena `"STRING COMO OBJETO"`. Esta posibilidad de utilizar estas cadenas literales como objetos con todas las de la ley resulta

ser bastante cómoda cuando se necesitan aplicar otros métodos directamente sobre estos valores.

3.5.1.2 Operadores de cadena

Otra de las bondades de las cadenas de caracteres en C# consiste en los operadores que son aplicables a los objetos de tipo `string` (y por tanto también a los literales de cadena). La línea:

```
Console.WriteLine("Te has demorado " + crono.ElapsedMilliseconds +
                  " ms en teclear tu nombre");
```

del Listado 3.2 nos ilustra el uso de operadores. En este ejemplo el operador `+`, conocido habitualmente para los tipos numéricos como operador de suma, sirve aquí para indicar la **concatenación** de cadenas:

```
"Te has demorado " + crono.ElapsedMilliseconds + " ms en teclear tu nombre"
```

Se está concatenando aquí la cadena literal `"Te has demorado "` con el valor numérico devuelto por la propiedad `crono.ElapsedMilliseconds` y con la cadena literal `" ms en teclear tu nombre"`.

Este mismo efecto puede obtenerse si se usa el método `Concat` de la clase `String`, que construye una nueva cadena concatenando los parámetros que recibe:

```
String.Concat("Te has demorado ", crono.ElapsedMilliseconds,
              " ms en teclear tu nombre");
```

También se pueden aplicar los operadores `==` y `!=` para comparar las secuencias de símbolos que forman el `string`. Estos son los únicos operadores de comparación aplicables al tipo `string`, ejemplificado en la Figura 3.1.

```
string str = "Gandalf";
...
bool iguales = str == "Gandalf";
```

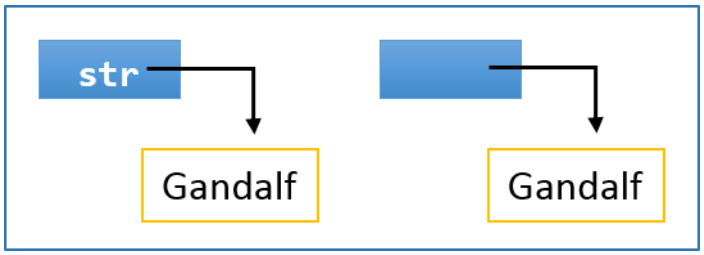

Figura 3.1. Espacios de memorias con el mismo valor para dos tipos string

No importa que sean objetos `string` cuyas cadenas de caracteres estén en lugares diferentes de memoria, el operador `==` en este caso compara las secuencias de ambas cadenas carácter a carácter y evaluará `true` si ambos `string` tienen el mismo valor

Si alguna de las variables tiene valor `null`, es decir no refiere a ningún objeto, el resultado de la comparación será `false` si la otra con la que se compara no es `null` y `true` si ambas son `null`. El lector no debe confundir una variable `string` que tenga como valor `null` (no tiene ningún objeto cadena asociado) con la cadena vacía `""`, ya que esta sí es un objeto de tipo `String` pero con cero caracteres.

3.5.1.3 Inmutabilidad de las cadenas

Los objetos `string` tienen la peculiaridad de ser objetos inmutables, o sea que ninguna de las operaciones o métodos que se le apliquen a un objeto `string` puede cambiar la secuencia de caracteres que lo conforman, sino en todo caso dar un nuevo objeto `string`. Para el Listado 3.8:

```
1  class PruebaDeInmutabilidad
2
3  {
4    private static void Main()
5
6
7    {
8
9      string str1 = "Yo soy inmutable";
10 string str2 = str1.ToUpper();
   Console.WriteLine(str1);
   Console.WriteLine(str2);
     }
   }
```

Listado 3.8 Inmutabilidad de las cadenas

Se obtendrá como resultado:

```
Yo soy inmutable
YO SOY INMUTABLE
```

Lo que indica que la secuencia de caracteres referida a través de la variable `str1` no se modificó por habérsele aplicado el método `ToUpper`, sino que se creó una nueva cadena que fue la que se le asignó a `str2` y que tiene una secuencia con los correspondientes caracteres de `str1` en mayúsculas.

Cuando se hace `string str1 = "Yo soy inmutable",` a la variable `str1` se le asocia el objeto correspondiente con la secuencia `"Yo soy inmutable",` como se muestra en la Figura 3.2 a).

Luego, cuando a través de `str1` se invoca al método `ToUpper`, lo que ocurre es que se crea un nuevo `string` con la secuencia de caracteres en mayúsculas, como se muestra en la Figura 3.2 b).

La nueva cadena se devuelve como resultado de la aplicación de `ToUpper` y se asigna a la variable `str2,` como se muestra en la Figura 3.2 c).

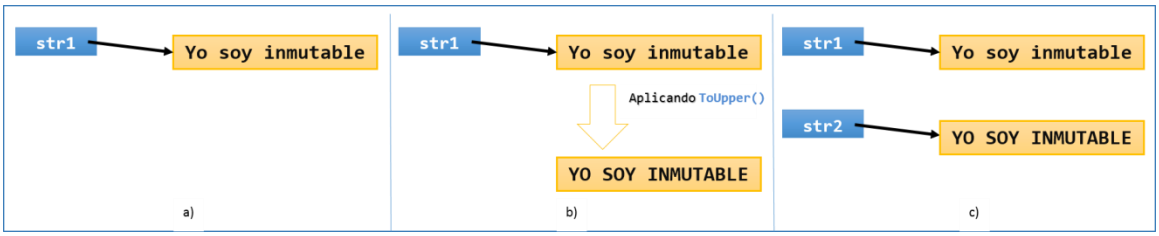

Figura 3.2. Inmutabilidad del tipo string

De forma que `str1` y `str2` hacen referencia a dos objetos independientes.

Esta es la semántica predeterminada, que aplica C#, para el trabajo con cadenas y es la deseada en la mayor parte de las aplicaciones porque evita efectos indeseados de afectar una cadena original sin proponérselo. Por accidente nunca podrá ocurrir entonces que *donde dije "diego" quise decir "digo"*

3.5.2 La clase System.String

Toda la operatoria básica asociada a las cadenas de caracteres está definida en la clase `String` del .NET Framework, la cual reúne un conjunto de métodos y propiedades útiles; a continuación se describen algunas de las más importantes.

3.5.2.1 Longitud de una cadena

La clase `String` define la propiedad `Length` que devuelve un `int` que indica la longitud de la cadena, es decir la cantidad de caracteres que forman la secuencia de la cadena. Por ejemplo:

```
"Gandalf".Length
```

devuelve como resultado el valor entero `7`, que es la longitud o cantidad de caracteres de la cadena.

Es un estilo en las bibliotecas de .NET usar el nombre `Length` para hablar de la longitud o de la cantidad de elementos de una cierta estructura que no cambia de tamaño. Tal es el caso también de los arrays (Capítulo 7). Cuando se habla de una estructura cuya cantidad de elementos puede cambiar durante la ejecución, se suele usar el nombre `Count`.

3.5.2.2 Búsqueda dentro de una cadena

Los métodos `IndexOf` y `LastIndexOf` permiten buscar subcadenas dentro de una cadena; una subcadena o *substring* es cualquier secuencia consecutiva de caracteres contenida en un `string`. Si se tiene:

```
"debieras amar la programación".IndexOf("ama")
```

esto devuelve el valor `9` por ser la primera ocurrencia de la subcadena `"ama"` en la cadena `"debieras amar la programación"`. Las posiciones de los caracteres dentro de la cadena

comienzan a contarse a partir `0`. Si se preguntara, sin embargo, por la cadena `"de"`, el resultado del método será `0`, por ser la posición de la primera ocurrencia de la cadena `"de"`.

Para preguntar por la siguiente aparición de la subcadena, habría que usar una sobrecarga del método `IndexOf`, que permite conocer la posición de la subcadena a partir de una posición que se le da como parámetro, por ejemplo:

```
int i = "debieras amar la programación".IndexOf("ama");
i = "debieras amar la programación".IndexOf("ama", i + 1);
```

En este caso `i` terminará con valor 22, porque en la segunda llamada al método buscará la ocurrencia de la subcadena `"ama"` a partir de la posición que se le indica en el segundo parámetro (`9` en este caso), aunque el resultado se sigue dando con respecto al inicio del `string` original.

Si no hay ninguna ocurrencia de la subcadena, entonces `IndexOf` devuelve `-1`. El método `LastIndexOf` es similar a `IndexOf`, pero lo que devuelve es la posición de la última ocurrencia de la subcadena.

3.5.3 El tipo char

El tipo `char` es un tipo integrado predefinido para expresar un solo carácter, que el lector puede interpretar como el caso más simple de un `string`. Sin embargo, no es lo mismo un `char` que un `string` de longitud `1`. Los valores literales de tipo `char` se representan entre apóstrofes o comillas simples (`'_'`) y no entre comillas dobles (`"_"`) como los `string`. Por ejemplo:

```
char ch = 'S';
```

El tipo `char` es un tipo tratado por valor y corresponde al `struct System.Char`

3.5.3.1 Métodos del tipo char

Los principales métodos aplicables a objetos de tipo `char` tienen que ver con la tabla de codificación estándar de caracteres Unicode[3]. Ejemplo de ello podemos encontrar los métodos estáticos `IsNumber`, `IsLetter`, `IsLetterOrDigit`, `IsUpper`, `IsSeparator`, `IsPunctuation`, etc. Por ejemplo:

```
bool minúscula = char.IsUpper('u');
```

Asigna el valor `false` a la variable `minúscula`, pues el caracter `'u'` es una letra minúscula.

Si se necesitara usar un carácter como caso especial de `string` y aplicarle los métodos de la clase `String`, todas las clases cuentan con un método `ToString()` (hecho que se explicará con mayor profundidad en el Capítulo 10 con la herencia)que devuelve un `string` correspondiente a la forma textual de ver un objeto. En el caso del tipo `char`, la aplicación de:

```
string str = 'U'.ToString();
```

Lo que hace es asignar a la variable `str` la cadena `"U"`.

[3] El Estándar Unicode es un estándar de codificación de caracteres diseñado para facilitar el tratamiento informático, transmisión y visualización de textos de múltiples lenguajes y disciplinas técnicas además de textos clásicos de lenguas muertas. El término Unicode proviene de los tres objetivos perseguidos: universalidad, uniformidad y unicidad.

3.5.3.2 Literales especiales

Una de las características de mayor importancia en cuanto al uso de caracteres en C# es la posibilidad de poder emplear valores literales especiales cuando no es posible teclear directamente ciertos caracteres particulares mediante los editores de texto.

Considere que se desea almacenar un párrafo completo en una variable de tipo `string` para su futuro procesamiento y queremos indicar cuándo se termina cada línea del párrafo. Es necesario poder concatenar cada secuencia que corresponde a una línea del párrafo con un separador que indique el cambio de línea en el párrafo correspondiente. Para ello existe el carácter especial `'\n'` que indica el cambio de línea; para el ejemplo:

```
string parrafo = "Si mucho coco comiera \nmucho coco
                  comprara\npero como poco coco como\npoco coco compro";
Console.WriteLine(parrafo);
```

se obtiene la salida:

```
Si mucho coco comiera
mucho coco comprara
Pero como poco coco como
Poco coco compro
```

En general cuando dentro del valor literal de una cadena se encuentra la combinación del caracter \ (en inglés *back slash*) con otro carácter detrás, la combinación de ambos caracteres se sustituye por el carácter especial correspondiente

Si no se pusiera el carácter especial y se ejecuta:

```
string parrafo = "Si mucho coco comiera mucho coco comprara"
                 +
                "pero como poco coco como poco coco compro";
Console.WriteLine(parrafo);
```

lo que se obtendría es

```
Si mucho coco comiera mucho coco comprara
pero como poco coco como poco coco compro
```

Además de este uso del \ para expresar el cambio de línea existen otros usos del \ para poder incluir en las cadenas los propios caracteres " , ' y \, y esto se hace con `'\"'`, `'\''` y `'\\'`. Si por ejemplo se quisiera representar en un `string` la ruta completa del archivo `c:\Windows\Explorer.exe`, habría que escribir:

```
string nombreDeArchivo = "c:\\Windows\\Explorer.exe";
```

O su contraparte utilizando *verbatim* string:

```
string nombreDeArchivo = @"c:\Windows\Explorer.exe";
```

Los literales de *cadena verbatim* son aquellos para los cuales al poner el caracter "@" delante de la cadena se hace posible obviar la interpretación del carácter \ como un carácter especial

3.5.4 Cadenas mutables

A diferencia de los métodos de `string`, los métodos de la clase `StringBuilder` sí trabajan sobre la propia secuencia de caracteres del objeto al que se le aplica. El tipo `StringBuilder` se encuentra en el espacio de nombres `System.Text`. Algunos de los métodos de esta clase son:

- `StringBuilder Append(string value)`
- `StringBuilder Insert(int index, string value)`
- `StringBuilder Remove(int startIndex, int length)`
- `StringBuilder Replace(string oldValue, string s)`

De este modo, un método como `StringBuilder Append(string value)` hace crecer el objeto `StringBuilder` al que se le aplica, logrando el equivalente de insertar la cadena valor de `s` al final del objeto al que se le aplica el método. En el ejemplo:

```
StringBuilder mutable = new StringBuilder("yo soy");
string inmutable = " programador";
mutable.Append(inmutable);
```

Se adiciona `" programador"` a la secuencia contenida en la variable `mutable`, como se muestra en la Figura 3.3.

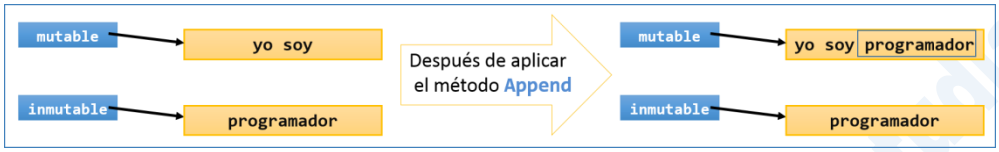

Figura 3.3. Ejemplo de aplicación del método Append a un StringBuilder

`StringBuilder Insert(int index, string value)` inserta la cadena `value` en la posición indicada por el parámetro `index` de la secuencia del objeto al que se le aplica el método. El ejemplo:

```
StringBuilder mutable = new StringBuilder("yosoyprogramador");
mutable.Insert(2, " ");
mutable.Insert(6, " ");
```

Convierte la secuencia contenida en el `StringBuilder mutable` en la secuencia `"yo soy programador"`, como se muestra en la Figura 3.4.

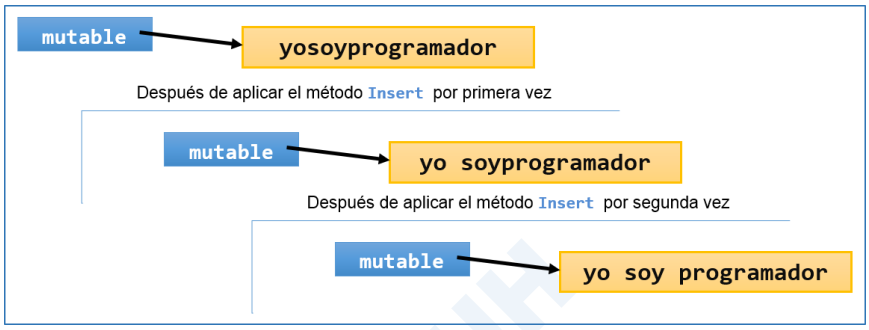

Figura 3.4. Ejemplo de aplicación del método Insert a un StringBuilder

`StringBuilder Remove(int startIndex, int length)` elimina de la secuencia de caracteres del objeto al que se le aplica el método una cantidad `length` de caracteres a partir de la posición indicada por `startIndex`.

`StringBuilder Replace(string oldValue, string newValue)` reemplaza en la cadena actual todas las ocurrencias de secuencias de caracteres coincidentes con `oldValue` por la secuencia de caracteres de la cadena `newValue`. Por ejemplo:

```
StringBuilder mutable = new StringBuilder("loco loco y loquito");
mutable.Replace("l", "c");
```

Cambia la secuencia original `"loco loco y loquito"` por la secuencia `"coco coco y coquito"`, como se muestra en la Figura 3.5.

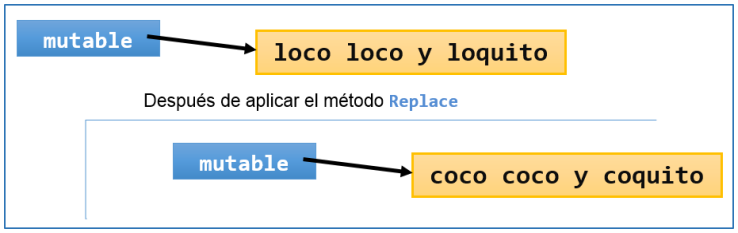

Figura 3.5. Ejemplo de aplicación del método Replace a un StringBuilder

3.6 El tipo bool

El tipo `bool` representa dos valores lógicos que se denotan con las palabras reservadas `true` (verdadero) y `false` (falso).

Los valores de tipo `bool` (en lo adelante le denominaremos booleanos) son muy útiles para expresar el cumplimiento o no de una condición y son utilizados en las instrucciones condicionales para determinar alternativas de ejecución o en las instrucciones de ciclos para determinar si se repite o no la ejecución de una determinada secuencia de instrucciones. Las instrucciones condicionales y de ciclos se estudian en los Capítulos 4 y 5.

3.6.1 Operaciones sobre operandos numéricos que dan resultados booleanos

La operación más básica que produce un resultado booleano es la operación de comparación por igualdad `==` y la comparación por desigualdad `!=`.

Ejemplos (suponga que las variables `a` y `b` son de tipo `int`):

- `a==10` da el valor `true` si el valor de la variable `a` es `10` y `false` en caso contrario
- `b!=0` da como resultado `true` si el valor de la variable `b` es diferente de cero y `false` en caso contrario
- `a==b` da como valor `true` si ambas variables tienen el mismo valor

Sobre operandos de algún tipo numérico se pueden aplicar también las operaciones de orden usuales de la matemática, `<` (menor), `<=` (menor o igual), `>` (mayor) y `>=` (mayor o igual).

Ejemplos:

- `a>0` da como valor `true` si el valor de la variable `a` es mayor que `0`
- `b<=100` da como valor `true` si el valor de la variable `b` es menor o igual que `100` y `false` en caso contrario.

3.6.2 Operaciones sobre operandos string que dan resultados booleanos

Las operaciones de comparación `==` y `!=` son aplicables también a operandos de tipo `string`. Dos cadenas son iguales si tienen la misma longitud y la misma secuencia de caracteres.

Ejemplos

```
string s1 = "locoloco"
string s2 = "loquito";
```

`s1 == "locoloco"` es `true`, pero `s1 == "LoCoLoCo"` es `false`, ya que la comparación entre cadenas es lo que se denomina como *case sensitive* (es decir "sensible" a la diferencia entre mayúsculas y minúsculas).

`s1 == "loco" + "loco"` es `true`

Si `s3 = "lo"`, entonces las siguientes comparaciones dan como resultado `true`:

```
s1 == s3 + "coloco"
s2 == s3 + "quito"
```

3.6.3 Operación de orden entre strings. El método CompareTo

Los operadores de orden `<`, `<=`, `>` y `>=` no son aplicables a operandos de tipo `string`. Aunque parezca tener sentido (aplicando el orden alfabético), las siguientes comparaciones de orden son reportadas como error por el compilador de C#

`s1 < s3` es error de compilación

`s2 >= "lo"` es error de compilación

Sin embargo, el tipo `string` ofrece el método `int CompareTo(string otro)` que compara alfabéticamente una cadena con otra devolviendo un valor de tipo `int` que debe interpretarse de la manera siguiente:

Si `p es "coco", q` es `"loco"` y `r` es `"poco"` entonces

`p.CompareTo(q)` es `-1,` ya que alfabéticamente `"coco"` es anterior a `"loco"`

`p.CompareTo("coco")` es `0,` ya que las cadenas son iguales

`r.CompareTo(q)` es 1, ya que alfabéticamente `"poco"` es posterior a `"loco"`

Esta idea de disponer de un método `CompareTo`, para con una notación uniforme poder hacer una comparación de orden entre dos objetos (siempre que para estos exista un tal criterio de comparación), se estudia en detalle en el Capítulo 10 con el tipo `IComparable`.

3.6.4 Métodos y Propiedades que devuelven un valor de tipo bool

Una de las mayores utilidades del tipo `bool` es que la de poder escribir métodos y propiedades que devuelven un resultado `bool`. De este modo, llamar a través de un objeto a un método o a una propiedad que devuelva un `bool` se puede percibir como si el objeto pudo hacer una determinada acción o cumple con una característica o propiedad.

Por ejemplo, el tipo `Stopwatch`, que se ha venido usando en algunos ejemplos, tiene una propiedad `IsRunning` que devuelve `true` para indicar que el cronómetro "está andando" (porque se hizo previamente un `Start`) y devuelve `false` si el cronómetro está detenido (porque se hizo un `Stop` o el cronómetro fue creado, pero nunca se le hizo un `Start`).

El código del Listado 3.9 escribe `true` al comenzar porque el cronómetro está andando, luego escribe `true` o `false` según si los nombres tecleados sean a no iguales, y finalmente `false` para indicar que el cronómetro se paró. La ejecución de este código da como salida:

```
Teclea tu nombre
miguel
Vuélvelo a teclear
IsRunning es True
miguel
Que los nombres sean iguales es True
IsRunning es False
```

```
1    Stopwatch crono = new Stopwatch();
2
3    Console.WriteLine("Teclea tu nombre");
4    string nombre1 = Console.ReadLine();
5
6    Console.WriteLine("Vuélvelo a teclear");
7    crono.Start();
8
9
```

```
10    Console.WriteLine("IsRunning es {0}", crono.IsRunning);
11
      string nombre2 = Console.ReadLine();
      crono.Stop();
      Console.WriteLine("Que los nombres sean iguales es {0}",
          nombre1 == nombre2);
      Console.WriteLine("IsRunning es {0}", crono.IsRunning);
```

Listado 3.9. Usando el tipo bool

3.6.5 Operadores booleanos

Es conveniente poder combinar en una misma expresión varias operaciones de comparación. Por ejemplo, si se tienen tres variables `a`, `b` y `c` de tipo numérico y se quiere determinar si `a` tiene el mayor valor de las tres, se podría expresar determinando si `a` es mayor o igual que `b` y también mayor o igual que `c`. Esto se puede expresar usando el operador booleano `&&` (operación que se debe leer como **y** en español y como **and** en inglés)

De modo que si la expresión `a >= b && a >= c` evalúa `true`, esto quiere decir que `a` es mayor o igual que `b` y que también es mayor o igual que `c`, y por tanto es la mayor de las tres

Note que no es necesario usar paréntesis porque el operador `&&` tiene menos prioridad que los operadores de comparación `>=`.

Si tiene dudas o quiere dar más legibilidad al código, puede escribir `(a >= b) && (a >= c)`, porque recuerde que los paréntesis ociosos no indigestan ni hacen daño

Si para tomar una decisión basta con que se cumpla una de dos condiciones, entonces puede usar el operador booleano `||` (se lee como **o** en español y como **or** en inglés). Si, por ejemplo, queremos saber si uno de los dos valores `a` o `b` es cero, se puede escribir la expresión:

```
a == 0 || b == 0
```

La Tabla 3.2 muestra los tres operadores booleanos `&&`, `||` y `!` (negación)

Variable	operando 1	operando 2	resultado
&& (y lógico)	false	False	false
	false	True	false
	true	False	false
	true	True	true
|| (o lógico)	false	False	false
	false	True	true
	true	False	true

	true	True	true
! (negación)	true		false
	false		true

Tabla 3.2 Operadores Booleanos

3.6.6 Evaluación en corto circuito

Si se observa la Tabla 3.2, se puede apreciar que el resultado de la operación `&&` siempre es `false` si alguno de los operandos es `false`, y el resultado de la operación `||` siempre es `true` si al menos alguno de los operandos es `true`. C# permite aplicar por tanto lo que se denomina evaluación en **corto circuito**. En una operación `&&` se evalúa el operando de la izquierda y si este es `false` ya no es necesario evaluar el operando de la derecha, porque el resultado final será `false`. Similarmente, si al aplicar la operación `||` el operando de la izquierda es `true` ya no se evalúa el operando de la derecha porque el resultado será `true`.

Esta forma de evaluación no solo es más eficiente porque evita evaluar innecesariamente el operando de la derecha, sino que además ayuda a formular expresiones que si tuviesen que evaluar previamente los dos operandos antes de aplicar la operación lógica podrían dar excepción. Observe la expresión a continuación (ponemos paréntesis para mayor legibilidad):

`(a != 0) &&` (b/a > 3)

Si se evalúa en corto circuito y `a` es `0`, entonces el operando `(a != 0)` daría `false` y no se evaluaría el operando `(b/a > 3)`, dando `false` como resultado. Sin embargo, si no se aplicara el corto circuito, entonces al evaluar el operando `(b/a > 3)` se estaría haciendo una división por `0`, lo que provocaría una excepción.

C# aplica de manera predeterminada la evaluación en corto circuito al aplicar los operadores `&&` y `||`

3.7 Asignación múltiple

Se pueden hacer varias asignaciones en una misma instrucción. Esto significa que se puede hacer:

```
int j = 2;
int k, m;
k = m = j + 3;
```

Esta instrucción se aplica con asociatividad a la derecha, es decir, primero se asigna el valor `j + 3` a la variable `m` y luego ese mismo valor se asigna a la variable `k`.

Es decir, que una instrucción de asignación puede usarse a su vez como un operando dentro de la expresión que calcula un valor. La evaluación del código a continuación:

```
int k, j, m;
Console.WriteLine(k = (j = (m = 1) + 1) + j + m);
```

escribe como resultado

```
5
```

Primero se hace la asignación `m = 1,` que asigna `1` a `m` y da como resultado 1; a esto se le suma `1` y se le asigna a `j` cuando se hace `j = (m = 1) + 1,` lo que deja en `j` el valor `2;` por lo tanto, si a esto se le suma `j` y `m` entonces lo que se le asigna a `k` es `5.`

Puede resultar confuso escribir código como el anterior. Pero esta interpretación de la asignación puede ser de utilidad cuando se pasa el valor de una expresión como un parámetro a un método y a la vez se quiere asignar a una variable el resultado de dicha evaluación, como se hace por ejemplo en el código siguiente

```
Math.Sqrt(c = a + b);
```

3.8 Asignación compuesta

La asignación compuesta es una posibilidad que brindan algunos lenguajes, entre ellos C#, de combinar un operador binario (+, -, *, etc.) con el operador de asignación = con una sintaxis de la forma

<entidad parte_izquierda> <op binario> = *<expresión parte_derecha>*

donde *<op binario>* puede ser uno de los siguientes operadores: `+, -, *, /, %, &, |, ^, <<, >>`.

En general, `x` *<op>*`= y`, es equivalente a `x = x` *<op>* `y`, por lo que este recurso del lenguaje permite escribir expresiones de forma más abreviada. Una instrucción como `x += y;` se interpreta como `x = x + y;`

El origen de este tipo de operación se remonta al lenguaje C, cuando con esta sintaxis se pretendía facilitarle al compilador generar un código más eficiente. Al escribir x+=y se le estaba diciendo al compilador que el resultado de la suma de x con y lo dejara en el propio x, para lo cual el repertorio de muchos CPU tienen un código más directo. Con las técnicas actuales de compilación y generación de código no sería necesario hacerle tales "sugerencias" al compilador, pero los lenguajes de la familia C (C++, Java, C#) mantuvieron esta notación.

Cuando se estudien en el Capítulo 6 las propiedades de un objeto, se verá que esta notación de expresiones compuestas es aplicable también cuando la entidad de la parte izquierda es el nombre de una propiedad. Para que esto sea posible, debe ser una propiedad que tenga tanto una parte `get` como parte `set`.

Considere un tipo `Cuenta` que tiene una propiedad `Saldo` de tipo `double`. Si `c` es un objeto de tipo `Cuenta` se pudiera escribir `c.Saldo += 100`. La parte `get` se necesita para que pueda ser aplicable la operación de tomar el valor de la propiedad `c.Saldo` y la parte `set` para que se pueda hacer la asignación `c.Saldo = c.Saldo + 100`.

3.9 Precedencia de los operadores

Como se ha visto en algunos de los ejemplos anteriores cuando una expresión contiene múltiples operadores, la **precedencia** de éstos controla el orden en que se evaluarán con relación a los otros. Por ejemplo al evaluar la expresión `x+y*z` ya se dijo que se evalúa primero `y*z` y luego se suma con `x`. Esto es debido a que el operador `*` tiene mayor precedencia que el operador `+`.

En el caso que un operando se encuentre entre dos operadores de igual precedencia, la asociatividad de estos controlará el orden de las operaciones a realizar. Exceptuando al operador de asignación `=`, todos los operadores binarios son asociativos por la izquierda; esto significa que las operaciones se realizan de izquierda a derecha. Por ejemplo, en `x*y/z` se evalúa primero `x*y` y luego eso se divide entre `z`, del mismo modo que si se hubiese escrito `(x*y)/z`.

La Tabla 3.3 resume la precedencia de los operadores (de mayor a menor, viendo la tabla de arriba hacia abajo). No se preocupe si aún no conoce alguno de estos operadores, los que hagan falta se irán presentando cuando se necesiten.

Categoría	Operadores
Primarios	`x.y f(x) a[x] x++ x-- new typeof checked unchecked`
Unarios	`+ - ! ~ ++x --x (T)x`
Multiplicativos	`* / %`
Aditivos	`+ -`
Shift o desplazamiento	`<< >>`
Relacionales y de chequeo de tipo	`< > <= >= is as`
Igualdad	`== !=`
AND lógico	`&`
XOR lógico	`^`
OR lógico	`|`
AND Condicional	`&&`
OR Condicional	`||`
Condicional	`?:`
Asignación	`= *= /= %= += -= <<= >>= &= ^= |=`

Tabla 3.3 Tabla de precedencia de operadores

3.10 Tipos enumerativos

Existe una forma para definir tipos de datos simples, que se conocen como **tipos enumerativos**, y que permite dar nombres simbólicos a los valores de un tal tipo enumerativo. Por ejemplo, se pudiera definir un tipo `TMes` para referirnos a los meses del año por su nombre y no por un valor numérico:

```
enum TMes{
    Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio, Agosto,
    Septiembre, Octubre, Noviembre, Diciembre };
```

Se les llama enumerativos porque precisamente la forma de definir el tipo es "enumerando" explícitamente todos sus valores.

Para evitar ambigüedades con el hecho de que un mismo nombre pueda usarse como valor de dos tipos enumerativos diferentes, para referirse a algún valor de un tipo enumerativo hay que hacerlo precediendo el nombre del valor con el nombre del tipo. Por ejemplo:

```
TMes miMesPreferido = TMes.Diciembre
```

Al trabajar con valores que se representan con nombres simbólicos, se facilita la lectura del código por parte de los humanos y a su vez se facilita a los compiladores la detección de errores. Por ejemplo, si se utiliza una variable `m` de tipo `int` para indicar un mes del año, la asignación `m=3;` es correcta aun cuando pueda haber un error de tecleo porque la intención era escribir `m=4;` (note que las teclas del 3 y el 4 están una al lado de la otra). Sin embargo, si `m` es del tipo `TMes` es muy poco probable que un error de tecleo nos lleve a escribir `m=TMes.Marzo;` cuando realmente lo que se quería escribir era `m=TMes.Abril;`

Los tipos enumerativos son útiles para usar en escenarios donde se maneja un número fijo de opciones conocidas y que para escribir el código es más simple y legible dar un nombre a cada una de estas opciones (imagine trabajar con colores usando un número para indicar cada color).

El lector no debe confundir el uso del tipo enumerativo con el uso del tipo `string`, que se ha venido utilizando para expresar secuencias o cadenas de caracteres. Por ejemplo, para expresar los nombres de personas no es viable usar un tipo enumerativo porque obligaría a enumerar explícitamente todos los nombres posibles[4], aunque claro esto introduciría el riesgo de que el compilador no nos puede detectar que no se teclee la cadena deseada.

De modo que si la intención era haber escrito

```
string nombre = "Gandalf";
```

el compilador no daría error si se escribe

```
string nombre = "Gandal";
```

Pero cuando las opciones son bien conocidas, y forman un conjunto finito razonable, es práctico, simple y seguro, usar un tipo enumerativo:

```
enum TSabores { Vainilla, Fresa, Chocolate, Almendra, Coco, Pistacho };
```

[4] Aunque a decir verdad, esto tal vez pudiera ser útil dada la cantidad creciente de nombres "raros" que proliferan en los últimos tiempos ☺.

```
TSabores miHeladoPreferido = TSabores.Chocolate;
```

En tal caso, el compilador sí podría detectar el error cuando detecte que se ha escrito:

```
TSabores miHeladoPreferido = TSabores.Chocolate;
```

Al ejecutar un código como

```
Console.WriteLine("Mi helado preferido es el de " + miHeladoPreferido);
```

C# reconoce que el operador de concatenación + espera un valor de tipo `string` y por tanto convierte el valor enumerativo de `miHeladoPreferido` a `string` y escribe

```
Mi helado preferido es el Chocolate
```

Realmente lo que sucede tras el telón es que se aplica invisiblemente el método `ToString`, que en este caso lo que hace es devolver un valor de tipo `string` con el nombre que se le dio a dicho valor enumerativo. El concepto que explica esta aplicación del método `ToString` se verá en el Capítulo 10 con la herencia.

Para representar internamente los valores enumerativos se utilizan valores numéricos enteros. Cada tipo enumerativo tiene por detrás un tipo simple entero asociado a él, los cuales pueden ser: `sbyte`, `short`, `ushort`, `int`, `uint`, `long` o `ulong`. C# asume de modo predeterminado que es `int`, pero se pude indicar explícitamente cuál es el que se quiere utilizar:

```
enum TDiasSemana : sbyte {
  Lunes, Martes, Miércoles, Jueves,
  Viernes, Sábado, Domingo
}
```

Si no se especifica lo contrario, esta enumeración empieza en cero, de modo que en el tipo `TDiasSemana` `Lunes` es `0`, `Martes` es `1` y así sucesivamente. Esta enumeración puede cambiarse si se indican explícitamente los valores asociados, como se muestra en el Listado 3.10.

```
1   enum Temperaturas
2   {
3     CongelaciónDelAgua = 0,
4     AmbienteAgradable = 22,
5     CuerpoHumano = 36,
6     AguaTibia = 50,
7     EvaporaciónDelAgua = 100
8   }
```

Listado 3.10. Modificando la numeración por defecto

El valor numérico asociado a un tipo enumerativo se puede obtener indicando explícitamente una "conversión". La instrucción a continuación asigna el valor `36` a la variable `k`:

```
int k = (int)Temperaturas.CuerpoHumano;
```

3.11 Tipos por valor y por referencia

En la asignación existen dos posibles semánticas: **por valor** y **por referencia**. Durante la asignación se utiliza una u otra en dependencia del tipo de la entidad de la parte izquierda.

Si la parte izquierda es de un tipo por valor (como el caso de los tipos primitivos antes estudiados), un tipo definido con `struct` (que se verá más adelante) o un tipo definido por `enum` (que a fin de cuentas es un entero), se utilizará la semántica por valor en la asignación. Si la entidad de la parte izquierda ha sido declarada de un tipo definido por una clase, entonces la semántica aplicada en la asignación es la de referencia.

¿En qué se diferencian estas dos semánticas de asignación? Cuando se utiliza la semántica por valor sobre la memoria asociada a la parte izquierda se copia el valor resultante de la evaluación de la parte derecha. Por ejemplo, si se tiene:

```
int k = 2;
int j = 4;
```

se tendría una representación en memoria como la que se muestra en ....

Si ahora se hiciese `j = k,` la memoria quedaría como se muestra en ....

Es decir el valor `2` que había en la memoria de `k` se ha copiado sobre la memoria de `j.`

Si ahora se hiciese `k++,` la memoria quedaría como se muestra en ....

Note que el valor en `j` no ha cambiado.

Considere la existencia de un tipo `Fecha` que tuviese tres variables para representar el día, el mes y el año de la fecha. Luego de hacer

```
Fecha bill = new Fecha(10, 4, 2015);
Fecha steve = new Fecha(15, 4, 2015);
```

La memoria quedaría como se muestra en ....

Pero al hacer

`bill = steve;` la memoria quedaría como se muestra en ...

Es decir, ambas variables están conectadas con un (se refieren al) mismo objeto tipo `Fecha`

Se dice que en este caso el tipo `Fecha` es un **tipo por referencia** y la asignación lo que hace es copiar la referencia valor de la variable `steve` como nuevo valor de la variable `bill.`

Esta es la semántica que se aplica de manera predeterminada cuando el tipo del objeto ha sido definido por una clase, como ha sido el caso del tipo `Fecha`. Cómo definir nuevos tipos mediante la especificación `class` se estudia en el Capítulo 6.

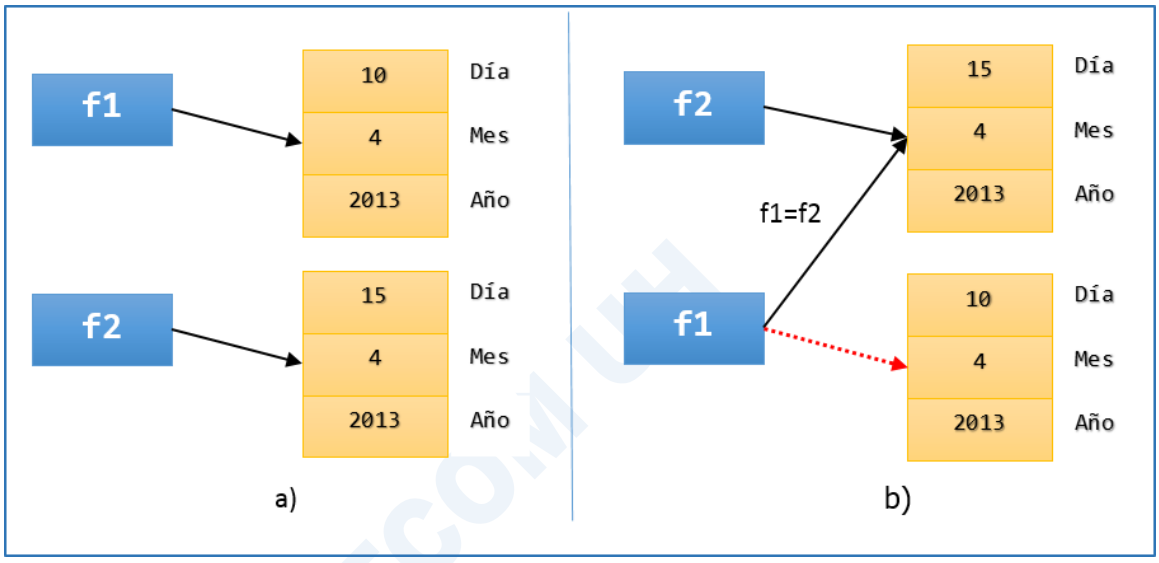

Figura 3.6. Ejemplo de asignación para tipo Fecha por Referenci

Note que luego de la asignación puede quedar la memoria de un objeto al que nadie refiere. A estos objetos que ocupan un lugar en memoria pero a los que ya ninguna entidad refiere se les denomina "basura" (*garbage*). Una característica importante del CLR de .NET es que hace **recolección automática de basura**. Esto significa que el CLR recupera y reutiliza la memoria de los objetos que se quedan sin ninguna referencia apuntando a ellos sin que tenga que mediar participación alguna del programador. Esto es causa de muchos errores de programación en otros lenguajes, porque o bien el programa podía quedar bloqueado debido a la ausencia de memoria libre, o el programador "liberaba" memoria cuando aún quedaba alguien refiriendo a ella, corrompiéndose por tanto la información que allí se tuviese.

Prácticamente cualquier lenguaje y plataforma de desarrollo moderna debe realizar recolección automática de basura. Los algoritmos y mecanismos utilizados por los recolectores de basura quedan fuera del alcance de este libro, aunque es importante comprender su importancia. La recolección automática de basura libera al programador de tener que controlar y administrar la utilización de la memoria, dando como resultado un código mucho más "limpio" y libre de errores, que por lo general son difíciles de detectar y solucionar.

3.11.1 Estructuras (struct)

Sin embargo, si el tipo `Fecha` se definiese utilizando `struct` en lugar de `class`:

```
struct Fecha{
    //Implementación de Fecha
}
```

entonces la representación en memoria luego de hacer

```
Fecha alan  = new Fecha(10, 4, 2013);
Fecha edgar = new Fecha(15, 4, 2013);
```

se muestra en ....

si ahora se hiciese `alan = edgar;` la representación en memoria quedaría como se muestra en ...

Es decir, todos los valores de las componentes de la fecha `edgar` se han copiado en las respectivas componentes de la fecha `alan`. Si se aplicase un método que hace avanzar una fecha una cantidad de días y se hiciese ahora `alan.AvanzaDias(2)`, se cambiaría la fecha valor de `alan` pero no se estaría cambiando el valor de `edgar` Figura 3.7 b).

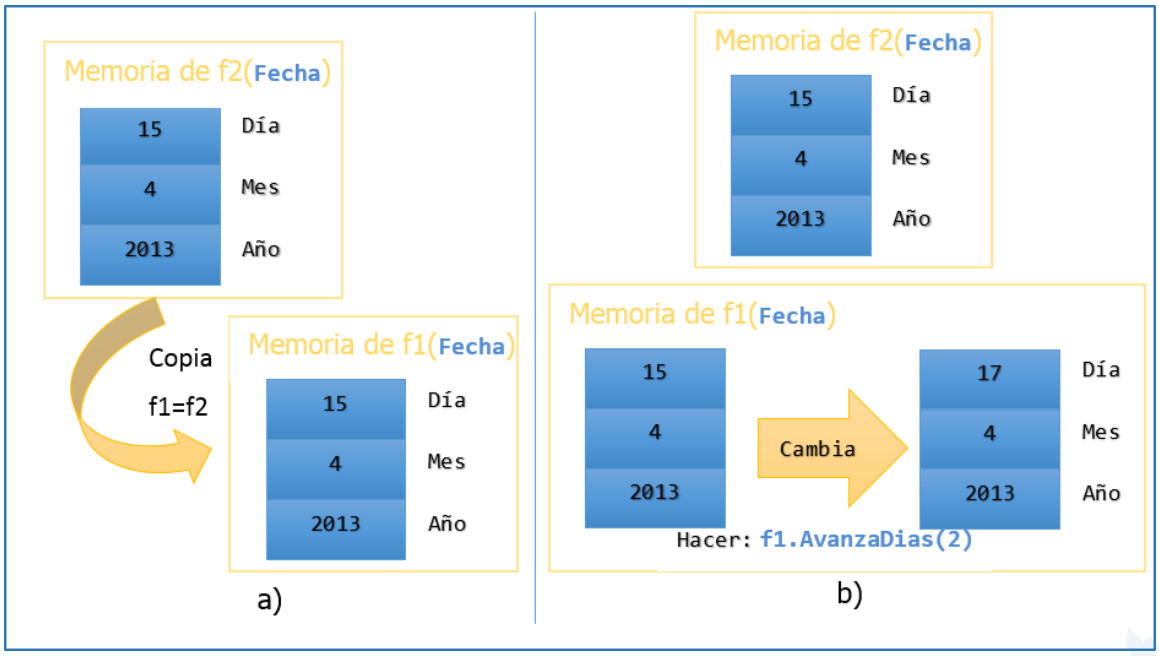

Figura 3.7. Ejemplo de asignación para tipo Fecha por Valor

3.11.2 Comparación por igualdad ==

TO DO