Tema 2.2

Descargar estos apuntes en pdf o html

Índice

Operadores y Expresiones en C#

Aunque han aparecido muchos operadores nuevos en C# 8 (2019), nosotros vamos a ver los más comunes a la hora de formar expresiones en la mayoría de lenguajes de programación. Muchos otros iremos hablando de ellos a lo largo del curso conforme veamos conceptos que nos permitan aplicarlos.

- **Tipos atendiendo al número de operadores**

Tipo	Ejemplo
Unarios	`(exp.)++`
Binarios	`(exp.1) + (exp.2)`
Ternarios	`(exp.1) ? (exp.2) : (exp.3)`

- **Tipos atendiendo a los valores con los que operan**

Tipo	Ejemplo
Operan con escalares	`(exp. eval. a escalar) * (exp. eval. a escalar)`
Operan con booleanos	`(exp. eval. a bool) && (exp. eval. a bool)`
Operan con bits	`(bits mem. de exp1) & (bits mem. de exp2)`

- **Tipos atendiendo al resultado**

Tipo	Ejemplo
Resultado escalar	`(exp.1) / (exp.2) → Se evalúa a escalar`
Resultado booleano	`(exp.1) >= (exp.2) → Se evalúa a bool`
	**`(exp.1)

Operadores Unarios

- **Casting o Conversión Explícita**
  Sintaxis: `(tipo)operando`

```
float a = 2.3F;
double b = a;
long c;

// Casteamos el resultado de la expresión a / b
c = (long)(a / b);
```

  - Forzamos un cambio de un tipo a otro.
  - Normalmente se utiliza para poder realizar algún tipo de operación o forzar el resultado en algún tipo determinado.
  - Si utilizamos conversión **implícita** con el operador '=' el compilador nos avisará, pero con la **explícita** no.
    > **Nota:** Cuidado con las conversiones de mayor a menor rango.

- **Unarios de Pre/Post incremento y decremento en 1**

```
i = ++j; // Equivale a hacer j=j+1; i=j;
i = j--; // Equivale a hacer i=j; j=j-1;
```

  - **Post-incremento**: a++; → Se evalúa a y se hace a = a + 1
  - **Post-decremento**: a--; → Se evalúa a y se hace a = a - 1
    Los de Pre-incremento/decremento se evalúan después que los de Post al tener menos prioridad.
  - **Pre-incremento**: ++a; → Se hace a = a + 1 y se evalúa a
  - **Pre-decremento**: --a; → Se hace a = a - 1 y se evalúa a

- **typeof(*<nombre tipo de dato>*)**
  Devuelve un objeto del tipo System.Type que guarda información sobre el tipo de datos sobre lo que lo apliquemos.

```
Type t1 = typeof(double);
Type t2 = typeof(int);
Console.WriteLine(t1);
Console.WriteLine(t2);
```

- **nameof(*<identificador>*)** C# 6
  Devuelve un literal de cadena con el identificador de una variable, ...

```
int total = 4;
Console.WriteLine(total); // Muestra -> 4
Console.WriteLine(nameof(total)); // Muestra -> total
```

- **!(*<expresión booleana>*)**
  Negación lógica, devuelve el valor lógico inverso de una expresión booleana.

```
bool r = !(5 > 7) // r se evaluará a true
```

- **-(*<expresión escalar>*)**
  Aplica el signo negativo al resultado escalar de la derecha.

```
int r = -(5 - 7) // r se evaluará a 2
```

- **+(<*expresión escalar*>)** (No muy común)
  Aplica el signo positivo al resultado escalar de la derecha.
  ```
  int r = +(5 - 7) // r se evaluará a -2
  ```

- **~(<*expresión*>)** (No muy común)
  Negación de bit. Invierte los bits en memoria de lo que estemos evaluando en la expresión.
  Se obtiene con la combinación (Alt + 126) o (Alt Gr + 4 seguido de espacio)
  ```
  byte r = ~0b10011011; // r se evaluará a 0b01100100;
  ```

- **checked(<*expresión escalar*>)** (No muy común)
  Detecta condiciones de desbordamiento en un expresión, generando un error durante la ejecución.
  ```
  short d1 = 20000, d2 = 20000;
  short miShort = checked((short)(d1 + d2));
  Console.WriteLine(miShort);
  ```

- **unchecked(<expresión escalar>)** (No muy común)
  Ignora condiciones de desbordamiento en un expresión, continuando con la ejecución.
  ```
  short d1 = 20000, d2 = 20000;
  short miShort = unchecked((short)(d1 + d2));
  Console.WriteLine(miShort);
  ```

Operadores Binarios

Aritméticos

Operador	Propósito
*	Multiplicación
/	División
%	Módulo
+	Suma
-	Resta

- El `%` requiere que el segundo operador no sea nulo.
  Al contrario que en otros lenguajes pueden ser reales.
- El de división `/` podemos tener varios casos:
  i. Si `int / int = Parte entera resultado`.
  ii. Si `float / float = float`
  iii. Si `float / int o int / float = float`

Comparación

- Dispondremos de los triviales operadores de comparación `x < y, x > y, x <= y, x >= y` que se evalúan a un valor booleano.
- Con un poco menos de prioridad los de igualdad `x == y, x != y` que también se evalúan a un valor booleano.
  > ✋ **Importante:** Un **error típico de principiante**, es confundir la asignación `=` con la comparación `==`
- En este apartado también podremos incluir el operador **is**, que me ayudará a preguntar a un identificador si es de un determinado tipo y lo usaremos más adelante al ver POO.
  También se le conoce como operador de reflexión y tendrá la siguiente sintaxis: `<id> is <tipo>`
  Devolverá un booleano indicándome si el operador es del tipo o no.

```
int i = 0;
bool test = i is int;

// test = true;
```

Lógicos

- **AND**: (<expresión booleana>) **&&** (<expresión booleana>)
  Tabla de verdad...

expresión	evaluación
`true  && true`	true
`true  && false`	false
`false && true`	false
`false && false`	false

- **OR**: (<expresión booleana>) **||** (<expresión booleana>)
  Tiene menos prioridad que el AND
  Tabla de verdad...

expresión	evaluación
**`true	
**`true	
**`false	
**`false	

Operador de uso combinado Null

- Disponible desde **C#7**, también se le conoce como **null coalescing operator**.
  Tradicionalmente no se ha usado, pero recientemente se está incrementando su uso. Podemos encontrarlo con similar funcionamiento en Swift, ES6 o PHP7 y con diferente sintaxis en otros lenguajes.
  Se evalúa de derecha izquierda y tiene la siguiente sintaxis:
  `(tipo o expresión anulable) ?? (tipo o expresión anulable)`

  ```
  int? a = null, b = 5;

  Console.WriteLine(a ?? b ?? 3);
  ```

  En el código de ejemplo hará:
  i. `b ?? 3` pero como **b** no es null se evaluará a su valor **5**.
  ii. `a ?? 5` al ser **a null** se evaluará a **5**.

  Si `int? b = null` entonces toda la expresión se evaluaría a **3**;
- Dispondremos de la versión del operador con asignación a partir de **C#8**
  Se evalúa de derecha izquierda y tiene la siguiente sintaxis:
  `(tipo o expresión anulable) ??= (tipo o expresión anulable)`

  ```
  int? a = null, b = 5;

  a = a ?? b;
  //Equivale a...
  a ??= b;
  ```

Operadores condicionales Null ?. y ?[]

Lo veremos más adelante en el tema 5.1 cuando veamos programación orientada a objetos básica.

Condicional Ternario

- **Sintaxis**: `Condición ? Consecuencia : Alternativa`
- **Condición**: Es una expresión que se evalúa a un booleano.
  - **Consecuencia**: A lo que se evalúa toda la expresión si `Condición` se evalúa a `true`.
  - **Alternativa**: A lo que se evalúa toda la expresión si `Condición` se evalúa a `false`.

  > **Importante**: Las expresiones `Consecuencia` y `Alternativa` se deben evaluar al mismo tipo de dato.

  ```
  (a > b) ? a : b; // Si a mayor que b entonces a sino b.
  ```

- Trataremos de **evitar usarla** o abusar del mismo, si obtenemos **expresiones ofuscadas** o podemos simplificar usando otros operadores.

```
int? a = 3, b = 11, c = null;

int? d = a > 0 && b <= 10 ? ++a * b : c != null ? c : 5;

// Podríamos reescribirla como...
int? d = (a > 2 && b <= 10)
         ? (++a * b)
         : (c ?? 5);
```

Cuadro Resumen con Precedencia y Asociatividad

Orden	Nombre	Asociatividad	Operador
0	Principales	izq. a der.	`x.y, f(x), a[i], x?.y, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, stackalloc`
1	**Unarios**	izq. a der.	`+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false`
2	**Intervalo** C#8	izq. a der.	`x..y`
3	`switch` **como expresión** C#8	izq. a der.	`x switch { v1 => expr, v2 => expr, _ => expr }`
4	Multiplicación, división, módulo división	izq. a der.	`x * y, x / y, x % y`
5	Suma y concatenación de cadenas, resta	izq. a der.	`x + y, x - y`
6	Desplazamiento de **bits**	izq. a der.	`x << y, x >> y`
7	Comparaciones, is, as	izq. a der.	`x < y, x > y, x <= y, x >= y, is, as`
8	Igualdad, desigualdad	izq. a der.	`x == y, x != y`
9	AND de **bits**	izq. a der.	`x & y`
10	XOR de **bits**	izq. a der.	`x ^ y`
11	OR de **bits**	izq. a der.	**`x
12	AND **lógico**	izq. a der.	`x && y`
13	OR **lógico**	izq. a der.	**`x
14	Operador de uso combinado de Null	der. a izq.	`x ?? y`
15	Condicional **ternario**	der. a izq.	`c ? t : f`
17	Asignación y Asignación compuesta.	der. a izq.	`x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y,` **`x
18	Evaluación Múltiple	izq. a der.	`x, y, ..., z`

Ejemplos de Precedencia con Operadores Básicos

Ejemplo 1:

Escribe la expresión algorítmica en C# para la siguiente expresión aritmética usando el menor número de paréntesis:

$$r = \frac{a^2}{b-c} + \frac{d-e}{f - \frac{g \cdot h}{j}}$$

```
int r = (a * a / (b - c)) + (d - e) / (f - g * h / j);
```

Ejemplo 2:

Evalúa la expresión `int a = -(4 * 4 / 2 - (4 * (8 % 2) + 12)) + 8 / 2 % 2;`
paso a paso teniendo en cuenta la precedencia de los operadores aritméticos.

```
int a = -(4 * 4 / 2 - (4 * (8 % 2) + 12)) + 8 / 2 % 2;
int a = -(4 * 4 / 2 - (4 * 0 + 12)) + 8 / 2 % 2;
int a = -(4 * 4 / 2 - (0 + 12)) + 8 / 2 % 2;
int a = -(4 * 4 / 2 - 12) + 8 / 2 % 2;
int a = -(16 / 2 - 12) + 8 / 2 % 2;
int a = -(8 - 12) + 8 / 2 % 2;
int a = -(-4) + 8 / 2 % 2;
int a = 4 + 8 / 2 % 2;
int a = 4 + 4 % 2;
int a = 4 + 0;
int a = 4;
```

Ejemplo 3:

Evalúa la expresión `bool a = 35 > 47 && 9 == 9 || 35 != 3 + 2 && 3 >= 3;`
paso a paso teniendo en cuenta la precedencia de los operadores aritméticos.

```
bool a = 35 > 47 && 9 == 9 || 35 != 3 + 2 && 3 >= 3;
bool a = 35 > 47 && 9 == 9 || 35 != 5 && 3 >= 3;
bool a = false && 9 == 9 || 35 != 5 && 3 >= 3;
bool a = false && 9 == 9 || 35 != 5 && true;
bool a = false && true || 35 != 5 && true;
bool a = false && true || true && true;
bool a = false || true && true;
bool a = false || true;
bool a = true;
```

Ejemplo 4:

- Sean **x, y, z, u, v, t, w** variables que contienen respectivamente los valores **2, 3, 4, 5, 6 y 7**

```
double x=2d, y=3d, z=4d, u=5d, v=6d, t=7d, w;
```

- ¿Qué almacenarán después de ejecutar cada una de las siguientes sentencias?
- Realiza una traza creando un tabla donde cada una de las filas sea la expresión que estoy evaluando en ese momento y las columnas el valor de las variable.

Nota: Puedes ver el resultado de evaluar estas expresiones mediante la instrucción:

```
Console.WriteLine($"x={x} - y={y} - z={z} - u={u} - v={v} - t={t} - w={w}");
```

	x	y	z	u	v	t	w
Valor Inicial	2	3	4	5	6	7	-
`x++;`	3	3	4	5	6	7	-
`y = ++z;`	3	5	5	5	6	7	-
`t = v--;`	3	5	5	5	5	6	-
`v = x + (y*=3) / 2;`	3	15	5	5	10,5	6	-
`w = x + y / 2;`	3	15	5	5	10,5	6	10,5