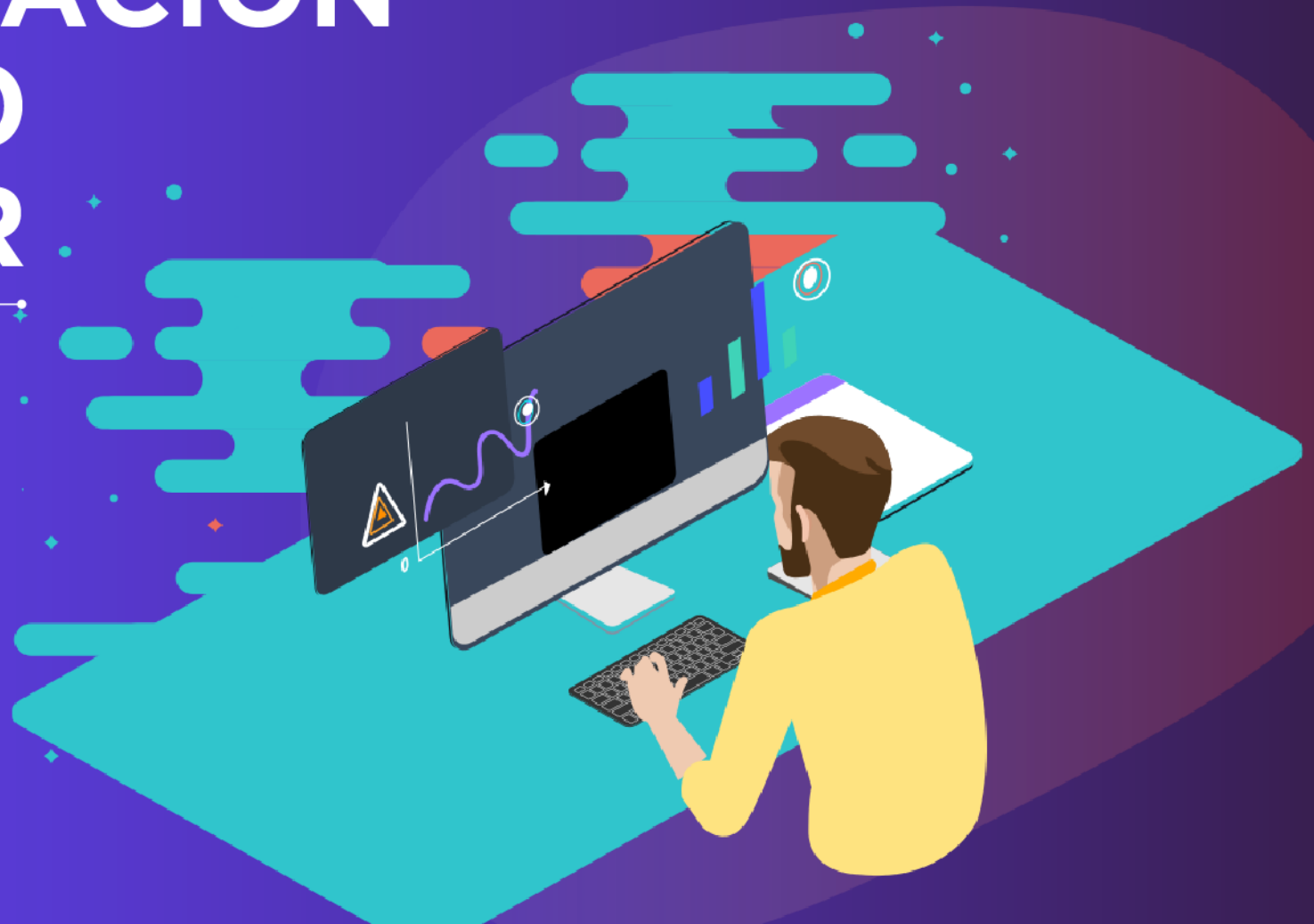
PROGRAMA

.NET ESPECIALIZACIÓN: NET

ESPECIALIZACIÓN ASP.NET 5.0 DEVELOPER



CURSO VIRTUAL

Fundamentos de Programación

Tipos de datos y Operadores

Instructor: Erick Aróstegui
earostegui@galaxy.edu.pe

GALAXY TRAINING

TEMAS

Tipos de datos y Operadores

01	Tipos de datos (num., cadenas, fechas y especiales).
02	Clases de conversión y principales métodos.
03	Operadores matemáticos y operadores OOP
04	Operadores lógicos.
05	Operadores relacionales y operador ternario.

Tipos de datos y Operadores

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos de datos (numéricos, cadenas, fechas y especiales).

C# es un lenguaje fuertemente tipado. Todas las variables y constantes tienen un tipo, al igual que todas las expresiones que se evalúan como un valor. Cada declaración del método especifica un nombre, un número de parámetros, un tipo y una naturaleza (valor, referencia o salida) para cada parámetro de entrada y para el valor devuelto. La biblioteca de clases .NET define un conjunto de tipos numéricos integrados, así como tipos más complejos que representan una amplia variedad de construcciones lógicas, como el sistema de archivos, conexiones de red, colecciones y matrices de objetos, y fechas. Los programas de C# típicos usan tipos de la biblioteca de clases, así como tipos definidos por el usuario que modelan los conceptos que son específicos del dominio del problema del programa.

Tipos de datos (numéricos, cadenas, fechas y especiales).

El compilador usa información de tipo para garantizar que todas las operaciones que se realizan en el código cuentan con seguridad de tipos. Por ejemplo, si declara una variable de tipo **int**, el compilador le permite usar la variable en operaciones de suma y resta. Si intenta realizar esas mismas operaciones en una variable de tipo bool, el compilador genera un error, como se muestra en el siguiente ejemplo:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Especificar tipos en declaraciones de variable

Cuando declare una variable o constante en un programa, debe especificar su tipo o usar la palabra clave **var** para que el compilador infiera el tipo. En el ejemplo siguiente se muestran algunas declaraciones de variable que utilizan tanto tipos numéricos integrados como tipos complejos definidos por el usuario:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos integrados

C# proporciona un conjunto estándar de tipos integrados para representar números enteros, valores de punto flotante, expresiones booleanas, caracteres de texto, valores decimales y otros tipos de datos. También hay tipos string y object integrados. Estos tipos están disponibles para su uso en cualquier programa de C#. Para obtener una lista completa de los tipos integrados, vea Tipos integrados.

- Tipos de valor
- Tipos de referencia

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos de valor

Los tipos de valor y los tipos de referencia son las dos categorías principales de tipos de C#. Una variable de un tipo de valor contiene una instancia del tipo. Esto difiere de una variable de un tipo de referencia, que contiene una referencia a una instancia del tipo. De forma predeterminada, al asignar, pasar un argumento a un método o devolver el resultado de un método, se copian los valores de variable. En el caso de las variables de tipo de valor, se copian las instancias de tipo correspondientes. En el ejemplo siguiente se muestra ese comportamiento:

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

    public MutablePoint(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified: {p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}
// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos numéricos enteros

Los tipos numéricos integrales representan números enteros. Todos los tipos numéricos integrales son tipos de valor. También son tipos simples y se pueden inicializar con literales. Todos los tipos numéricos enteros admiten operadores aritméticos, lógicos bit a bit, de comparación y de igualdad.

Palabra clave/tipo de C#	Intervalo	Tamaño	Tipo de .NET
sbyte	De -128 a 127	Entero de 8 bits con signo	System.SByte
byte	De 0 a 255	Entero de 8 bits sin signo	System.Byte
short	De -32 768 a 32 767	Entero de 16 bits con signo	System.Int16
ushort	De 0 a 65.535	Entero de 16 bits sin signo	System.UInt16
int	De -2.147.483.648 a 2.147.483.647	Entero de 32 bits con signo	System.Int32
uint	De 0 a 4.294.967.295	Entero de 32 bits sin signo	System.UInt32
long	De -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	Entero de 64 bits con signo	System.Int64
ulong	De 0 a 18.446.744.073.709.551.615	Entero de 64 bits sin signo	System.UInt64

```
int a = 123;
System.Int32 b = 123;

var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos numéricos de punto flotante

Los tipos numéricos de punto flotante representan números reales. Todos los tipos numéricos de punto flotante son tipos de valor. También son tipos simples y se pueden inicializar con literales. Todos los tipos de punto flotante numéricos admiten operadores aritméticos, de comparación y de igualdad.

Palabra clave/tipo de C#	Intervalo aproximado	Precisión	Tamaño	Tipo de .NET
float	De ±1,5 x 10-45 a ±3,4 x 1038	De 6 a 9 dígitos aproximadamente	4 bytes	System.Single
double	De ±5,0 × 10−324 a ±1,7 × 10308	De 15 a 17 dígitos aproximadamente	8 bytes	System.Double
decimal	De ±1,0 x 10-28 to ±7,9228 x 1028	28-29 dígitos	16 bytes	System.Decimal

```
double a = 12.3;
System.Double b = 12.3;
```

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

bool

La palabra clave de tipo bool es un alias para el tipo de estructura de .NET System.Boolean que representa un valor booleano que puede ser true o false.

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not checked
```

char

La palabra clave de tipo char es un alias para el tipo de estructura de .NET System.Char que representa un carácter Unicode UTF-16.

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars));  // output: j j j j
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos de enumeración

Un tipo de enumeración es un tipo de valor definido por un conjunto de constantes con nombre del tipo numérico integral subyacente. Para definir un tipo de enumeración, use la palabra clave enum y especifique los nombres de miembros de enumeración

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos de estructura

Un tipo de estructura (o tipo struct) es un tipo de valor que puede encapsular datos y funcionalidad relacionada. Para definir un tipo de estructura se usa la palabra clave struct.

Al diseñar un tipo de estructura, tiene las mismas funciones que con un tipo de clase, con las siguientes excepciones

No se puede declarar un constructor sin parámetros.

No se puede inicializar una propiedad o un campo de instancia en su declaración.

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos de tupla

Disponible en C# 7.0 y versiones posteriores, la característica tuplas proporciona una sintaxis concisa para agrupar varios elementos de datos en una estructura de datos ligera. En el siguiente ejemplo se muestra cómo se puede declarar una variable de tupla, inicializarla y acceder a sus miembros de datos:

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos de valor que admiten valores NULL

Un tipo de valor que admite un valor NULL T? representa todos los valores de su tipo de valor subyacente T y un valor NULL adicional. Por ejemplo, puede asignar cualquiera de los tres valores siguientes a una variable bool?: true, false o null. Un tipo de valor subyacente T no puede ser un tipo de valor que acepte valores NULL por sí mismo.

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7
```

```
int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

var

A partir de C# 3, las variables que se declaran en el ámbito de método pueden tener un "tipo" var implícito. Una variable local con tipo implícito es fuertemente tipada exactamente igual que si hubiera declarado el tipo, solo que en este caso es el compilador el que lo determina. Las dos declaraciones siguientes de i tienen una función equivalente.

```
var i = 10; // Implicitly typed.
int i = 10; // Explicitly typed.

List<int> xs = new();
List<int>? ys = new();
```

```
// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
                where word[0] == 'g'
                select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
                where cust.City == "Phoenix"
                select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos de registro

En C# 9.0 se presentan los tipos de registro, un tipo de referencia que ofrece métodos sintetizados para proporcionar semántica de valores para la igualdad. Los registros son inmutables de forma predeterminada.

```
public record Pet(string Name)
{
    public void ShredTheFurniture() =>
        Console.WriteLine("Shredding furniture");
}

public record Dog(string Name) : Pet(Name)
{
    public void WagTail() =>
        Console.WriteLine("It's tail wagging time");

    public override string ToString()
    {
        StringBuilder s = new();
        base.PrintMembers(s);
        return $"{s.ToString()} is a dog";
    }
}
```

Tipos de datos (numéricos, cadenas, fechas y especiales).

Tipos de registro

En C# 9.0 se presentan los tipos de registro, un tipo de referencia que ofrece métodos sintetizados para proporcionar semántica de valores para la igualdad. Los registros son inmutables de forma predeterminada.

```
public record Pet(string Name)
{
    public void ShredTheFurniture() =>
        Console.WriteLine("Shredding furniture");
}

public record Dog(string Name) : Pet(Name)
{
    public void WagTail() =>
        Console.WriteLine("It's tail wagging time");

    public override string ToString()
    {
        StringBuilder s = new();
        base.PrintMembers(s);
        return $"{s.ToString()} is a dog";
    }
}
```

Conversiones de tipos

Conversiones de tipos

Dado que C# tiene tipos estáticos en tiempo de compilación, después de declarar una variable, no se puede volver a declarar ni se le puede asignar un valor de otro tipo a menos que ese tipo sea convertible de forma implícita al tipo de la variable. Por ejemplo, string no se puede convertir de forma implícita a int. Por tanto, después de declarar i como un valor int, no se le puede asignar la cadena "Hello", como se muestra en el código siguiente:

```
int i;

// error CS0029: Cannot implicitly convert type 'string' to 'int'
i = "Hello";
```

Conversiones de tipos

Conversiones implícitas

No se requiere ninguna sintaxis especial porque la conversión siempre es correcta y no se perderá ningún dato. Los ejemplos incluyen conversiones de tipos enteros más pequeños a más grandes, y conversiones de clases derivadas a clases base.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

Conversiones de tipos

Conversiones explícitas

Las conversiones explícitas requieren una expresión Cast. La conversión es necesaria si es posible que se pierda información en la conversión, o cuando es posible que la conversión no sea correcta por otros motivos. Entre los ejemplos típicos están la conversión numérica a un tipo que tiene menos precisión o un intervalo más pequeño, y la conversión de una instancia de clase base a una clase derivada.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe)a;
```

Conversiones de tipos

Conversiones definidas por el usuario

Las conversiones definidas por el usuario se realizan por medio de métodos especiales que se pueden definir para habilitar las conversiones explícitas e implícitas entre tipos personalizados que no tienen una relación de clase base-clase derivada. Para obtener más información, vea Operadores de conversión definidos por el usuario.

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number);  // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit);  // output: 7
    }
}
```

Conversiones de tipos

Conversiones con clases del asistente de conversión

Para realizar conversiones entre tipos no compatibles, como enteros y objetos System.DateTime, o cadenas hexadecimales y matrices de bytes puede usar la clase System.BitConverter, la clase System.Convert y los métodos Parse de los tipos numéricos integrados, como Int32.Parse. Para obtener más información, consulte Procedimiento Convertir una matriz de bytes en un valor int, Procedimiento Convertir una cadena en un número y Procedimiento Convertir cadenas hexadecimales en tipos numéricos.

```
double dNumber = 23.15;

try {
    // Returns 23
    int    iNumber = System.Convert.ToInt32(dNumber);
}
catch (System.OverflowException) {
    System.Console.WriteLine(
                "Overflow in double to int conversion.");
}
// Returns True
```

```
Convert("12,000", NumberStyles.Float | NumberStyles.AllowThousands,
        new CultureInfo("en-GB"));
Convert("12,000", NumberStyles.Float | NumberStyles.AllowThousands,
        new CultureInfo("fr-FR"));
Convert("12,000", NumberStyles.Float, new CultureInfo("en-US"));
```

```
int number = Int32.Parse(value, style, provider);
Console.WriteLine("Converted '{0}' to {1}.", value, number);
```

Conversiones de tipos

Conversiones con clases del asistente de conversión

Para realizar conversiones entre tipos no compatibles, como enteros y objetos System.DateTime, o cadenas hexadecimales y matrices de bytes puede usar la clase System.BitConverter, la clase System.Convert y los métodos Parse de los tipos numéricos integrados, como Int32.Parse. Para obtener más información, consulte Procedimiento Convertir una matriz de bytes en un valor int, Procedimiento Convertir una cadena en un número y Procedimiento Convertir cadenas hexadecimales en tipos numéricos.

```
double dNumber = 23.15;

try {
    // Returns 23
    int    iNumber = System.Convert.ToInt32(dNumber);
}
catch (System.OverflowException) {
    System.Console.WriteLine(
                "Overflow in double to int conversion.");
}
// Returns True
```

```
Convert("12,000", NumberStyles.Float | NumberStyles.AllowThousands,
        new CultureInfo("en-GB"));
Convert("12,000", NumberStyles.Float | NumberStyles.AllowThousands,
        new CultureInfo("fr-FR"));
Convert("12,000", NumberStyles.Float, new CultureInfo("en-US"));
```

```
int number = Int32.Parse(value, style, provider);
Console.WriteLine("Converted '{0}' to {1}.", value, number);
```

Operadores aritméticos

Operadores aritméticos

C# proporciona una serie de operadores. Muchos de ellos son compatibles con los tipos integrados y permiten realizar operaciones básicas con valores de esos tipos. Entre estos operadores se incluyen los siguientes grupos:

Operadores aritméticos, que realizan operaciones aritméticas con operandos numéricos.

Operadores de comparación, que comparan operandos numéricos.

Operadores lógicos booleanos, que realizan operaciones lógicas con operandos bool.

Operadores bit a bit y de desplazamiento, que realizan operaciones bit a bit o de desplazamiento con operandos de tipos enteros.

Operadores de igualdad, que comprueban si sus operandos son iguales o no.

Operadores aritméticos

Operadores aritméticos

Los operadores siguientes realizan operaciones aritméticas con operandos de tipos numéricos:

- **Operadores unarios ++ (incremento), -- (decremento), + (más) y - (menos).**
- **Operadores binarios * (multiplicación), / (división), % (resto), + (suma) y - (resta).**

Estos operadores se admiten en todos los tipos numéricos enteros y de punto flotante.

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4
```

```
Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

```
Console.WriteLine(+4);     // output: 4

Console.WriteLine(-4);     // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);            // output: -5
Console.WriteLine(b.GetType());  // output: System.Int64

Console.WriteLine(-double.NaN);  // output: NaN
```

```
Console.WriteLine(.41f % .2f); // output: 0.00999999

double a = 0.1;
double b = 3 * a;
Console.WriteLine(b == 0.3);   // output: False
Console.WriteLine(b - 0.3);    // output: 5.55111512312578E-17

decimal c = 1 / 3.0m;
decimal d = 3 * c;
Console.WriteLine(d == 1.0m);  // output: False
Console.WriteLine(d);          // output: 0.9999999999999999999999999999
```

Tipos de datos y Operadores

Operadores lógicos.

Operadores lógicos

Los operandos siguientes realizan operaciones lógicas con los operandos bool:

- **Operador unario !** (negación lógica).
- **Operadores binarios & (AND lógico), | (OR lógico) y ^ (OR exclusivo lógico)**. Esos operadores siempre evalúan ambos operandos.
- **Operadores binarios && (AND lógico condicional) y || (OR lógico condicional)**. Esos operadores evalúan el operando derecho solo si es necesario.

```
bool passed = false;
Console.WriteLine(!passed);  // output: True
Console.WriteLine(!true);    // output: False
```

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

Operadores relacionales y operador ternario

Operadores relacionales y operador ternario

Operador ?

El operador condicional ?:, también conocido como operador condicional ternario, evalúa una expresión booleana y devuelve el resultado de una de las dos expresiones, en función de que la expresión booleana se evalúe como true o false.

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;

IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2, 3 };
```

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

Operadores relacionales y operador ternario

?? Operadores ?? y ??

El operador de uso combinado de NULL ?? devuelve el valor del operando izquierdo si no es null; en caso contrario, evalúa el operando derecho y devuelve su resultado. El operador ?? no evalúa su operando derecho si el operando izquierdo se evalúa como no NULL.

Disponible en C# 8.0 y versiones posteriores, el operador de asignación de uso combinado de NULL ??= asigna el valor de su operando derecho al operando izquierdo solo si el operando izquierdo se evalúa como null. El operador ??= no evalúa su operando derecho si el operando izquierdo se evalúa como no NULL.

Operadores relacionales y operador ternario

?? Operadores ?? y ??

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers));  // output: 5

numbers.Add(a ??= 0);
Console.WriteLine(string.Join(" ", numbers));  // output: 5 0
Console.WriteLine(a);  // output: 0
```

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum = SumNumbers(null, 0);
Console.WriteLine(sum);  // output: NaN
```

```
int? a = null;
int b = a ?? -1;
Console.WriteLine(b);  // output: -1
```

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), "Name cannot be null");
}
```

Operadores relacionales y operador ternario

Operador =>

El token => se admite de dos formas: como el operador lambda y como un separador de un nombre de miembro y la implementación del miembro en una definición de cuerpo de expresión.

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
  .Where(w => w.StartsWith("a"))
  .Min(w => w.Length);
Console.WriteLine(minimalLength);   // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);   // output: 280
```

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product);   // output: 280
```

```
public override string ToString() => $"{fname} {lname}".Trim();
```

```
public override string ToString()
{
   return $"{fname} {lname}".Trim();
}
```

Covarianza y contravarianza

Covarianza y contravarianza

Covarianza y contravarianza son términos que hacen referencia a la capacidad de usar un tipo más derivado (más específico) o menos derivado (menos específico) que el indicado originalmente. Los parámetros de tipo genérico admiten la covarianza y contravarianza para proporcionar mayor flexibilidad a la hora de asignar y usar tipos genéricos.

Cuando se hace referencia a un sistema de tipos, la covarianza, contravarianza e invarianza tienen las definiciones siguientes. En el ejemplo se presupone una clase base denominada Base y una clase derivada denominada Derived.

Covarianza y contravarianza

Covariance

Permite usar un tipo más derivado que el especificado originalmente.

Puede asignar una instancia **de IEnumerable<Derived>** a una variable de tipo **IEnumerable<Base>.**

Contravariance

Permite usar un tipo más genérico (menos derivado) que el especificado originalmente.

Puede asignar una instancia de **Action<Base>** a una variable de tipo **Action<Derived>.**

Invariance

Significa que solo se puede usar el tipo especificado originalmente. Un parámetro de tipo genérico invariable no es covariante ni contravariante.

No se puede asignar una instancia de **List<Base>** a una variable de tipo **List<Derived>** o viceversa.

Covarianza y contravarianza

```
IEnumerable<Derived> d = new List<Derived>();
IEnumerable<Base> b = d;
```

```
Action<Base> b = (target) => { Console.WriteLine(target.GetType().Name); };
Action<Derived> d = b;
d(new Derived());
```

```
using System;
using System.Collections.Generic;

class Base
{
    public static void PrintBases(IEnumerable<Base> bases)
    {
        foreach(Base b in bases)
        {
            Console.WriteLine(b);
        }
    }
}

class Derived : Base
{
    public static void Main()
    {
        List<Derived> dlist = new List<Derived>();

        Derived.PrintBases(dlist);
        IEnumerable<Base> bIEnum = dlist;
    }
}
```

Funciones locales

Funciones locales

A partir de C# 7.0, C# admite funciones locales . Las funciones locales son métodos privados de un tipo que están anidados en otro miembro. Solo se pueden llamar desde su miembro contenedor. Las funciones locales se pueden declarar en y llamar desde:

- Métodos, especialmente los métodos de iterador y asincrónicos
- Constructores
- Descriptores de acceso de propiedad
- Descriptores de acceso de un evento
- Métodos anónimos
- Expresiones lambda
- Finalizadores
- Otras funciones locales

En cambio, las funciones locales no se pueden declarar dentro de un miembro con forma de expresión.

Funciones locales

```
private static string GetText(string path, string filename)
{
     var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
     var text = reader.ReadToEnd();
     return text;

     string AppendPathSeparator(string filepath)
     {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
     }
}
```

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

```
using System;
using System.Collections.Generic;

public class IteratorWithoutLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)  // line 11
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the output like this:
//
//    Retrieved enumerator...
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100. (Parameter 'en
//    at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32 end)+MoveNext() in IteratorWithoutLocal.cs:line
//    at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line 11
```

Clases y métodos parciales

Clases y métodos parciales

Clases parciales

Es posible dividir la definición de una clase, un struct, una interfaz o un método en dos o más archivos de código fuente. Cada archivo de código fuente contiene una sección de la definición de tipo o método, y todos los elementos se combinan cuando se compila la aplicación.

Es recomendable dividir una definición de clase en varias situaciones:

- Cuando se trabaja con proyectos grandes, el hecho de repartir una clase entre archivos independientes permite que varios programadores trabajen en ella al mismo tiempo.
- Cuando se trabaja con código fuente generado automáticamente, se puede agregar código a la clase sin tener que volver a crear el archivo de código fuente. Visual Studio usa este enfoque al crear formularios Windows Forms, código de contenedor de servicio Web, etc. Puede crear código que use estas clases sin necesidad de modificar el archivo creado por Visual Studio.
- Para dividir una definición de clase, use el modificador de palabra clave partial

Clases y métodos parciales

Clases parciales

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

```
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

Clases y métodos parciales

Métodos parciales

Una clase o struct parcial puede contener un método parcial. Un elemento de la clase contiene la firma del método. Se puede definir una implementación opcional en el mismo elemento o en otro. Si no se proporciona la implementación, el método y todas las llamadas al método se quitan en tiempo de compilación.

Una declaración de método parcial consta de dos elementos: la definición y la implementación. Pueden encontrarse en elementos independientes de una clase parcial o en el mismo elemento. Si no hay ninguna declaración de implementación, el compilador optimiza tanto la declaración de definición como todas las llamadas al método.

Clases y métodos parciales

Métodos parciales

- Las declaraciones de método parcial deben comenzar con la palabra clave contextual partial y el método debe devolver void.
- Los métodos parciales pueden tener parámetros in o ref, pero no parámetros out.
- Los métodos parciales son implícitamente private y, por tanto, no pueden ser virtual.
- Los métodos parciales no pueden ser extern, ya que la presencia del cuerpo determina si son de definición o de implementación.
- Los métodos parciales pueden tener modificadores static y unsafe.
- Los métodos parciales pueden ser genéricos. Las restricciones se colocan en la declaración de método parcial de definición y opcionalmente pueden repetirse en el de implementación. Los nombres del parámetro y del parámetro de tipo no tienen que ser iguales en la declaración de implementación y en la declaración de definición.
- Puede crear un delegado para un método parcial que se ha definido e implementado, pero no para un método parcial que solo se ha definido.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
  // method body
}
```

Establecedores de solo inicialización

Establecedores de solo inicialización

Los *establecedores de solo inicialización proporcionan una sintaxis coherente para inicializar miembros de un objeto. Los inicializadores de propiedades indican con claridad qué valor establece cada propiedad. El inconveniente es que esas propiedades se deben establecer. A partir de C# 9.0, puede crear descriptores de acceso init en lugar de descriptores de acceso set para propiedades e indizadores. Los autores de la llamada pueden usar la sintaxis de inicializador de propiedad para establecer estos valores en expresiones de creación, pero esas propiedades son de solo lectura una vez que se ha completado la construcción.

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure";
}
```

```
var now = new WeatherObservation
{
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

Instrucciones de nivel superior

Instrucciones de nivel superior

Las instrucciones de nivel superior quitan complejidad innecesaria de muchas aplicaciones. Considere el famoso programa "Hola mundo":

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

```
using System;

Console.WriteLine("Hello World!");
```

```
System.Console.WriteLine("Hello World!");
```

Instrucciones de nivel superior

Solo un archivo de la aplicación puede usar instrucciones de nivel superior. Si el compilador encuentra instrucciones de nivel superior en varios archivos de código fuente, se trata de un error. También es un error si combina instrucciones de nivel superior con un método de punto de entrada de programa declarado, normalmente Main. En cierto sentido, puede pensar que un archivo contiene las instrucciones que normalmente se encontrarían en el método Main de una clase Program.

Uno de los usos más comunes de esta característica es la creación de materiales educativos. Los desarrolladores principiantes de C# pueden escribir el programa "Hola mundo" en una o dos líneas de código. No se necesitan pasos adicionales. Pero los desarrolladores veteranos también encontrarán muchas aplicaciones a esta característica. Las instrucciones de nivel superior permiten una experiencia de experimentación de tipo script similar a la que proporcionan los cuadernos de Jupyter Notebook. Las instrucciones de nivel superior son excelentes para programas y utilidades de consola pequeños. Azure Functions es un caso de uso ideal para las instrucciones de nivel superior.

PREPÁRATE
PARA SER EL
MEJOR

+ ENTREMIENTO
EXPERIENCIA

BIENVENIDOS.

GRACIAS
POR TU PARTICIPACIÓN

Por favor, bríndanos tus comentarios
y sugerencias para mejorar nuestros servicios.

GALAXY
TRAINING


GALAXY
TRAINING