Fundamentos de C#

Tipos de Datos y Variables

C# es un lenguaje de tipado estático, lo que significa que el tipo de una variable se conoce en tiempo de compilación. Esta característica ayuda a detectar errores temprano en el proceso de desarrollo. C# proporciona un conjunto rico de tipos de datos incorporados, que se pueden clasificar en tipos de valor y tipos de referencia.

Tipos de Valor

Los tipos de valor contienen datos directamente. Se almacenan en la pila e incluyen lo siguiente:

- **Tipos Integrales:** Estos incluyen `int`, `long`, `short`, `byte`, `sbyte`, `uint`, `ulong` y `ushort`. Por ejemplo:
  ```
  int edad = 30;
  ```
- **Tipos de Punto Flotante:** Estos incluyen `float` y `double`. Por ejemplo:
  ```
  double precio = 19.99;
  ```
- **Tipos Integrales:** Estos incluyen `int`, `long`, `short`, `byte`, `sbyte`, `uint`, `ulong` y `ushort`. Por ejemplo:
  ```
  int edad = 30;
  ```
- **Tipo Decimal:** El tipo `decimal` se utiliza para cálculos financieros donde la precisión es crítica:
  ```
  decimal salario = 50000.00m;
  ```
- **Tipo Booleano:** El tipo `bool` puede contener dos valores: `true` o `false`:
  ```
  bool esActivo = true;
  ```
- **Tipo Carácter:** El tipo `char` representa un único carácter Unicode de 16 bits:
  ```
  char inicial = 'A';
  ```

Tipos de Referencia

Los tipos de referencia almacenan referencias a sus datos (objetos) y se almacenan en el montón. Los tipos de referencia comunes incluyen:

- **Cadena:** Una secuencia de caracteres:
  ```
  string nombre = "John Doe";
  ```
- **Arreglos:** Una colección de elementos del mismo tipo:
  ```
  int[] numeros = { 1, 2, 3, 4, 5 };
  bool resultado = (5 > 3) && (3 < 10); // resultado es true
  ```
- **Clases:** Tipos definidos por el usuario que pueden contener miembros de datos y métodos:
  ```
  class Persona { public string Nombre; public int Edad; }
  ```
- **Interfaces:** Definen un contrato que las clases pueden implementar:
  ```
  interface IAnimal { void Hablar(); }
  ```

Conversión de Tipos

La conversión de tipos es cuando se asigna un valor de un tipo de datos a otro tipo.

Hay dos tipos de conversión:

- Conversión implícita: `(automática): conversión de un tipo más pequeño a un tamaño de tipo más grande`
  `char -> int -> long -> float -> double`
- Conversión explícita: `conversión de un tipo más grande a un tipo de tamaño más pequeño`
  `double -> float -> long -> int -> char`

```
int myInt = 9;
double myDouble = myInt; // Casting implícito

double myDouble = 9.78;
int myInt = (int) myDouble; // Casting explícito
```

Métodos de Conversión

```
int myInt = 10;
double myDouble = 5.25;
bool myBool = true;

Console.WriteLine(Convert.ToString(myInt)); // convert int to string
Console.WriteLine(Convert.ToDouble(myInt)); // convert int to double
Console.WriteLine(Convert.ToInt32(myDouble)); // convert double to int
Console.WriteLine(Convert.ToString(myBool)); // convert bool to string
```

Enumeradores

Un enum es una “clase” especial que representa un grupo de constantes (de sólo lectura)

```
enum Level
{
  Low,
  Medium,
  High
}
```

Arreglos

Los arreglos son indexados iniciando en cero (0)

Unidimensionales

```
Declarar: int[] valores;
Crear: valores = new int[100]; //crea un arreglo con 100 elementos
```

int[] valores1 = new int[10] {0,1,2,3,4,5,6,7,8,9};
valores1[1] = 4; //Cambia el valor del indice 1 a 4

Multidimensionales
```
int[,] valores1; //sin inicializar
int[,] valores2 = new int[3,7];
int[,] numeros = new int[3, 4] { {1, 2,3,4}, {9, 8,7,6}, {7, 6,2,5} };

int[,,] valores1; //sin inicializar
int[,,] valores2 = new int[3,7,4];
numeros[2,1] = 10; //Cambia el valor de indice 2,1 a 10
```

Arreglos de Arreglos
```
int[][] matriz = new int[3][];
for (int i = 0; i < matriz.Length; i++)
{
  matriz[i] = new int[4];
}
matriz[2][1] = 4;
```

Operadores y Expresiones

Los operadores en C# son símbolos especiales que realizan operaciones en variables y valores. Se pueden clasificar en varios tipos:

Operadores Aritméticos

Estos operadores se utilizan para realizar operaciones matemáticas básicas:

- `+` (Suma)
- `-` (Resta)
- `*` (Multiplicación)
- `/` (División)
- `%` (Módulo)

Ejemplo:

```
int suma = 5 + 10; // suma es 15
```

Operadores Relacionales

Estos operadores se utilizan para comparar dos valores:

- `==` (Igual a)
- `!=` (No igual a)
- `>` (Mayor que)
- `<` (Menor que)
- `>=` (Mayor o igual a)
- `<=` (Menor o igual a)

Ejemplo:

```
bool esIgual = (5 == 5); // esIgual es true
```

Operadores Lógicos

Los operadores lógicos se utilizan para combinar múltiples expresiones booleanas:

- `&&` (Y lógico)
- `||` (O lógico)
- `!` (No lógico)

Operadores de Asignación

Estos operadores se utilizan para asignar valores a variables:

- `=` (Asignación simple)
- `+=` (Sumar y asignar)
- `-=` (Restar y asignar)
- `*=` (Multiplicar y asignar)
- `/=` (Dividir y asignar)
- `%=` (Módulo y asignar)

Ejemplo:

```
int x = 5; x += 3; // x ahora es 8
```

Operador Condicional

El operador condicional (también conocido como operador ternario) es una forma abreviada de la declaración `if-else`:

```
int max = (a > b) ? a : b; // max será el mayor de a o b
```

Sentencias de Control de Flujo

Las sentencias de control de flujo te permiten dictar el orden en que se ejecutan las sentencias en tu programa. C# proporciona varios tipos de sentencias de control de flujo:

Sentencias Condicionales

Las sentencias condicionales ejecutan diferentes bloques de código según ciertas condiciones:

Sentencia If

La sentencia `if` ejecuta un bloque de código si una condición especificada es verdadera:

```
if (edad >= 18) { Console.WriteLine("Adulto"); }
```

Sentencia If-Else

La sentencia `if-else` te permite ejecutar un bloque de código si la condición es verdadera y otro bloque si es falsa:

```
if (edad >= 18) { Console.WriteLine("Adulto"); } else {
Console.WriteLine("Menor"); }
```

Sentencia Switch

La sentencia `switch` es una forma más limpia de manejar múltiples condiciones:

```
switch (dia) { case 1: Console.WriteLine("Lunes"); break; case 2:
Console.WriteLine("Martes"); break; default:
Console.WriteLine("Otro"); }
```

Sentencias de Bucle

Las sentencias de bucle te permiten ejecutar un bloque de código múltiples veces:

Bucle For

El bucle `for` se utiliza cuando se conoce el número de iteraciones:

```
for (int i = 0; i < 10; i++) { Console.WriteLine(i); }
```

Bucle While

El bucle `while` continúa ejecutándose mientras una condición especificada sea verdadera:

```
int i = 0; while (i < 10) { Console.WriteLine(i); i++; }
```

Bucle Do-While

El bucle `do-while` es similar al bucle `while`, pero garantiza que el bloque de código se ejecute al menos una vez:

```
int i = 0; do { Console.WriteLine(i); i++; } while (i < 10);
```

Bucle Foreach

El bucle `foreach` se utiliza para iterar sobre colecciones, como arreglos o listas:

```
foreach (var numero in numeros) { Console.WriteLine(numero); }
```

Entender estos conceptos fundamentales de la sintaxis y los fundamentos de C# es crucial para cualquier desarrollador que busque sobresalir en C#. La maestría de los

tipos de datos, operadores y sentencias de control de flujo no solo ayudará a escribir código eficiente, sino también a prepararse para entrevistas técnicas donde estos temas se discuten con frecuencia.

Modificadores de acceso

Se utilizan para establecer el nivel de acceso/visibilidad para clases, campos, métodos y propiedades.

Modificador	Descripción
`Public`	El código es accesible desde todas las clases
`private`	El código es sólo accesible dentro de la misma clase
`protected`	El código es accesible dentro de la misma clase o desde una clase que hereda de esa clase
`internal`	El código es sólo accesible dentro del propio ensamblado, pero no desde otro ensamblado

Programación Orientada a Objetos en C#

La Programación Orientada a Objetos (OOP) es un paradigma de programación que utiliza "objetos" para representar datos y métodos para manipular esos datos. C# es un lenguaje que apoya completamente los principios de OOP, lo que lo convierte en una herramienta poderosa para los desarrolladores. Exploraremos los conceptos fundamentales de OOP en C#, incluyendo clases y objetos, herencia y polimorfismo, así como encapsulación y abstracción.

Clases y Objetos

Una **clase** en C# es un plano para crear objetos. Define propiedades (atributos) y métodos (funciones) que tendrán los objetos creados. Un **objeto** es una instancia de una clase. Cuando creas un objeto, estás instanciando una clase.

```
public class Car
{
    // Propiedades
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }

    // Método
    public void DisplayInfo()
    {
        Console.WriteLine($"Coche: {Year} {Make} {Model}");
    }
}

// Creando un objeto
Car myCar = new Car();
myCar.Make = "Toyota";
myCar.Model = "Corolla";
myCar.Year = 2020;
myCar.DisplayInfo(); // Salida: Coche: 2020 Toyota Corolla
```

En el ejemplo anterior, definimos una clase `Car` con tres propiedades: `Make`, `Model` y `Year`. El método `DisplayInfo` muestra los detalles del coche. Luego creamos una instancia de la clase `Car` y establecemos sus propiedades antes de llamar al método para mostrar la información.

Herencia y Polimorfismo

Herencia es un mecanismo en C# que permite que una clase herede las propiedades y métodos de otra clase. Esto promueve la reutilización del código y establece una relación jerárquica entre las clases. La clase de la que se hereda se llama **clase base**, mientras que la clase que hereda se llama **clase derivada**.

```
public class Vehicle
{
    public string Make { get; set; }
    public string Model { get; set; }

    public void DisplayInfo()
    {
        Console.WriteLine($"Vehículo: {Make} {Model}");
    }
}

public class Car : Vehicle
{
    public int Year { get; set; }

    public new void DisplayInfo()
    {
        Console.WriteLine($"Coche: {Year} {Make} {Model}");
    }
}

// Creando un objeto de la clase derivada
Car myCar = new Car();
myCar.Make = "Honda";
myCar.Model = "Civic";
myCar.Year = 2021;
Console.WriteLine(myCar.DisplayInfo()); // Salida: Coche: 2021 Honda
Civic
```

En este ejemplo, tenemos una clase base `Vehicle` con propiedades y un método. La clase `Car` hereda de `Vehicle` y añade una nueva propiedad, `Year`. También sobrescribe el método `DisplayInfo` para proporcionar una salida específica para coches. Esto demuestra cómo la herencia nos permite extender la funcionalidad de una clase base.

Polimorfismo es otro concepto clave en OOP que permite que los métodos hagan cosas diferentes según el objeto sobre el que actúan. En C#, el polimorfismo se puede lograr a través de la sobrescritura de métodos y interfaces.

```
public class Truck : Vehicle
{
    public int LoadCapacity { get; set; }

    public override void DisplayInfo()
    {
        Console.WriteLine($"Camión: {Make} {Model}, Capacidad de
Carga: {LoadCapacity} toneladas");
    }
}

// Usando polimorfismo
```

```
Vehicle myTruck = new Truck();
myTruck.Make = "Ford";
myTruck.Model = "F-150";
((Truck)myTruck).LoadCapacity = 3;
myTruck.DisplayInfo(); // Salida: Camión: Ford F-150, Capacidad de
Carga: 3 toneladas
```

En este ejemplo, creamos una clase `Truck` que también hereda de `Vehicle`. El método `DisplayInfo` se sobrescribe para proporcionar información específica sobre camiones. Cuando creamos una referencia de `Vehicle` a un objeto `Truck`, aún podemos llamar al método sobrescrito, demostrando el polimorfismo.

Encapsulación y Abstracción

Encapsulación es la agrupación de datos (atributos) y métodos (funciones) que operan sobre los datos en una sola unidad, o clase. Restringe el acceso directo a algunos de los componentes del objeto, lo que es un medio para prevenir interferencias no intencionadas y el uso indebido de los métodos y datos. En C#, la encapsulación se logra utilizando modificadores de acceso.

```
public class BankAccount
{
    private decimal balance;

    public void Deposit(decimal amount)
    {
        if (amount > 0)
        {
            balance += amount;
        }
    }

    public void Withdraw(decimal amount)
    {
        if (amount > 0 && amount <= balance)
        {
            balance -= amount;
        }
    }

    public decimal GetBalance()
    {
        return balance;
    }
}

// Usando la clase BankAccount
BankAccount account = new BankAccount();
account.Deposit(100);
account.Withdraw(50);
Console.WriteLine(account.GetBalance()); // Salida: 50
```

En este ejemplo, la clase `BankAccount` encapsula el campo `balance`, haciéndolo privado. La única forma de modificar el balance es a través de los métodos `Deposit` y `Withdraw`, lo que asegura que el balance no pueda establecerse en un estado inválido.

Abstracción es el concepto de ocultar la realidad compleja mientras se exponen solo las partes necesarias. Ayuda a reducir la complejidad de la programación y aumenta la eficiencia. En C#, la abstracción se puede lograr utilizando clases abstractas e interfaces.

```
public abstract class Shape
{
    public abstract double Area();
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public override double Area()
    {
        return Math.PI * Radius * Radius;
    }
}

// Usando la clase Circle
Circle circle = new Circle { Radius = 5 };
Console.WriteLine($"Área del Círculo: {circle.Area()}"); // Salida:
Área del Círculo: 78.53981633974483
```

En este ejemplo, definimos una clase abstracta `Shape` con un método abstracto `Area`. La clase `Circle` hereda de `Shape` y proporciona una implementación concreta del método `Area`. Esto nos permite trabajar con diferentes formas mientras solo necesitamos conocer el método `Area`, demostrando la abstracción.

Entender los principios de la Programación Orientada a Objetos en C#—incluyendo clases y objetos, herencia y polimorfismo, encapsulación y abstracción—es crucial para cualquier desarrollador. Estos conceptos no solo ayudan a organizar el código, sino también a crear aplicaciones escalables y mantenibles.