C# en .Net

Autor:

Kevin Cárdenas.

2023

Índice

1. Introducción

C# (pronunciado "C Sharp") es un lenguaje de programación de propósito general, moderno y orientado a objetos, desarrollado por Microsoft como parte de su plataforma .NET. C# se utiliza para desarrollar aplicaciones de escritorio, aplicaciones web, aplicaciones móviles y videojuegos.

C# fue creado en 2000 por el programador danés Anders Hejlsberg, quien anteriormente había trabajado en el desarrollo del lenguaje de programación Turbo Pascal y del entorno de desarrollo integrado (IDE) Borland Delphi. La primera versión pública de C# fue lanzada en 2002 como parte de la plataforma .NET Framework.

C# es un lenguaje diseñado para ser fácil de leer, escribir y mantener. Está diseñado para ser seguro y robusto, y su sintaxis se asemeja a la de otros lenguajes de programación populares, como Java y C++. C# incluye muchas características avanzadas, como la recolección de basura, el control de excepciones, los delegados y los eventos, y los genéricos, que lo hacen muy potente y versátil.

C# se utiliza comúnmente junto con el IDE Visual Studio de Microsoft, que proporciona herramientas para la edición de código, la depuración, la compilación y el despliegue de aplicaciones. Además, C# es compatible con múltiples plataformas, incluyendo Windows, Linux y macOS, lo que lo hace ideal para el desarrollo de aplicaciones multiplataforma.

C# se inspiró en varios lenguajes de programación anteriores, incluidos C++, Java y Delphi, entre otros. El objetivo de C# era combinar lo mejor de estos lenguajes en un solo lenguaje que fuera fácil de aprender y usar.

C# se basa en gran medida en la sintaxis de C++, con algunas diferencias clave para hacer que el lenguaje sea más seguro y fácil de usar. C# utiliza palabras clave específicas para definir tipos de datos, como "int" para enteros y "string" para cadenas. También incluye soporte para la herencia y el polimorfismo, que son características clave de la programación orientada a objetos.

Además de C++, C# también se inspiró en Java. De hecho, la sintaxis de C# es muy similar a la de Java, lo que hace que sea fácil para los desarrolladores que ya conocen Java aprender C#. C# también utiliza el modelo de máquina virtual de Java, que permite que el código C# se ejecute en cualquier plataforma que tenga una implementación de la máquina virtual de .NET.

En C#, el tipado se refiere a la definición del tipo de datos que puede contener una variable. C# es un lenguaje de programación fuertemente tipado, lo que significa que todas las variables deben tener un tipo de datos definido antes de que se puedan utilizar. Esto es diferente a los lenguajes de programación débilmente tipados, como JavaScript, que permiten que las variables cambien de tipo de datos dinámicamente.

En C#, hay dos tipos de datos básicos: tipos de valor y tipos de referencia. Los tipos de valor contienen directamente el valor de los datos, como los números enteros, los números de punto flotante y los booleanos. Los tipos de referencia, por otro lado, contienen una referencia a un objeto en la memoria, como una cadena, una matriz o un objeto personalizado.

C# también admite el uso de tipos genéricos, que permiten definir una clase o un método que puede trabajar con diferentes tipos de datos. Por ejemplo, una lista genérica puede contener cualquier tipo de datos, como enteros, cadenas o objetos personalizados.

Además, C# permite la definición de tipos de datos personalizados a través de las estructuras y clases. Las estructuras son tipos de valor que se utilizan para almacenar datos relacionados, como un punto en el espacio tridimensional. Las clases, por otro lado, son tipos de referencia que se utilizan para definir objetos complejos con propiedades, métodos y eventos.

Existen convenciones para nombrar variables y otros elementos del código que ayudan a que el código sea más legible y fácil de entender. Algunas de las convenciones de nomenclatura más comunes son

- CamelCase: es la convención más comúnmente utilizada. En este caso, la primera letra de la primera palabra se escribe en minúscula y la primera letra de cada palabra subsiguiente se escribe en mayúscula. Por ejemplo, "nombreDeVariable" o "miVariableNumeroUno". Esta convención se utiliza comúnmente para nombrar **variables y métodos.**

- PascalCase: en este caso, la primera letra de cada palabra se escribe en mayúscula, sin espacios ni guiones bajos. Por ejemplo, "NombreDeVariable" "MiVariableNumeroUno". Esta convención se utiliza a menudo para nombrar **clases**.

- Upper_Case: en este caso, todas las letras se escriben en mayúscula. Esta convención se utiliza generalmente para nombrar constantes. Por ejemplo, "VALOR_MAXIMO" o "PI". Eta convención se utiliza comúnmente para nombrar **constantes**.

- snake_case: en este caso, se utiliza una separación con guión bajo entre las palabras, todas en minúsculas. Por ejemplo, "nombre_de_variable" o "mi_variable_numero_uno". Esta convención se utiliza a menudo en otros lenguajes de programación, como Python, y algunos desarrolladores la usan para nombrar **variables y métodos.**

C# es un lenguaje de programación muy completo que está diseñado para trabajar con programación orientada a objetos, pero también permite la programación funcional. Es importante conocer los principios de la orientación a objetos para poder programar eficientemente en C#, y también es posible utilizar conceptos de la programación funcional como inmutabilidad, funciones de orden superior y lambdas.

1.1. Gramatica y Tipos

Para iniciar un programa en C# es necesario crear un archivo con extensión `.cs` que contenga el código fuente del programa. El archivo debe contener una clase con un método llamado `Main`, que es el punto de entrada del programa. La estructura básica de un programa en C# se ve así:

```
1 using System;
```

```

class Program{
    static void Main(string[] args){
        // code
    }
}
```

En la primera línea se encuentra la directiva `using System`, que indica que se utilizarán clases del espacio de nombres `System`. Luego se define la clase `Program`, que contiene el método `Main` con un parámetro `args` de tipo `string[]`, que representa los argumentos pasados al programa desde la línea de comandos.

Dentro del método `Main` se escribe el código que se quiere ejecutar al iniciar el programa. Por ejemplo, si se quiere imprimir un mensaje por consola, se puede utilizar el método `Console.WriteLine` de la clase `System`:

```
using System;

class Program{
    static void Main(string[] args){
        Console.WriteLine("Hola, mundo!");
    }
}
```

Para ejecutar el programa, se debe compilar el archivo `.cs` utilizando un compilador de C# como el IDE Visual Studio, y luego, si es el caso, ejecutar el archivo resultante con extensión `.exe`.

En C#, las variables deben declararse antes de usarse y se pueden inicializar con un valor. La sintaxis básica de una declaración de variable en C# es la siguiente:

```
tipo nombre_variable;
```

Donde `tipo` es el tipo de variable y `nombre_variable` es el nombre que se le da a la variable.

Para inicializar una variable con un valor, se puede usar la siguiente sintaxis:

```
tipo nombre_variable = valor_inicial;
```

Donde `valor_inicial` es el valor que se le asigna a la variable al momento de su inicialización.

En C#, también es posible declarar varias variables del mismo tipo en una sola línea, separando cada nombre de variable con una coma. Por ejemplo:

```
int a, b, c;
```

También es posible inicializar varias variables en una sola línea, separando cada par de nombre de variable y valor inicial con una coma. Por ejemplo:

```
int a = 1, b = 2, c = 3;
```

Aquí te presento una lista más precisa de los tipos de variables que existen en C#:

- Tipos numéricos: `int`, `long`, `short`, `byte`, `float`, `double`, `decimal`.

- Tipo de caracteres: `char`.
- Tipo booleano: `bool`.
- Tipos de referencia: `string`, `object` y otros tipos definidos por el usuario.
- Tipos de colecciones: `array`, `List<T>`, `Dictionary<TKey, TValue>` y otros tipos de colecciones definidos por el usuario.
- Tipo implícito: `var`.

En cuanto a la gramática de las variables en C#, hay algunas reglas que se deben seguir:

- Los nombres de las variables deben comenzar con una letra o un guión bajo (_), seguidos de cero o más letras, dígitos o guiones bajos.
- Los nombres de las variables no pueden ser iguales a palabras clave de C#.
- Los nombres de las variables son sensibles a mayúsculas y minúsculas.

Es importante tener en cuenta que los nombres de las variables deben ser descriptivos y representar el propósito de la variable en el código.

En cuanto a los tipos de variables en C#, hay varios tipos numéricos (enteros y decimales), tipos de caracteres, tipos booleanos y tipos de referencia. Cada tipo de variable tiene un rango de valores que puede almacenar y una precisión determinada.

1.2. Operadores

En C#, los operadores se utilizan para realizar operaciones matemáticas, lógicas, de comparación y de asignación. A continuación, se describen algunos de los operadores más comunes en C#:

Operadores aritméticos

Los operadores aritméticos se utilizan para realizar operaciones matemáticas básicas como la suma, la resta, la multiplicación y la división. A continuación, se muestran los operadores aritméticos en C#:

- +: Suma dos valores. Ejemplo: `int resultado = 5 + 3;` // resultado es igual a 8.
- -: Resta dos valores. Ejemplo: `int resultado = 5 - 3;` // resultado es igual a 2.
- *: Multiplica dos valores. Ejemplo: `int resultado = 5 * 3;` // resultado es igual a 15.
- /: Divide dos valores. Ejemplo: `int resultado = 15 / 3;` // resultado es igual a 5.
- %: Devuelve el resto de una división. Ejemplo: `int resultado = 15% 4;` // resultado es igual a 3.

Operadores de asignación

Los operadores de asignación se utilizan para asignar un valor a una variable. A continuación, se muestran los operadores de asignación en C#:

- =: Asigna un valor a una variable. Ejemplo: `int edad = 25;`
- +=: Incrementa el valor de una variable. Ejemplo: `int contador = 0; contador += 1;` // contador es igual a 1.
- -=: Decrementa el valor de una variable. Ejemplo: `int contador = 1; contador -= 1;` // contador es igual a 0.
- *=: Multiplica el valor de una variable. Ejemplo: `int resultado = 5; resultado *= 3;` // resultado es igual a 15.
- /=: Divide el valor de una variable. Ejemplo: `int resultado = 15; resultado /= 3;` // resultado es igual a 5.
- %=: Asigna el resto de una división a una variable. Ejemplo: `int resultado = 15; resultado %= 4;` // resultado es igual a 3.

Operadores de comparación

Los operadores de comparación en C# se utilizan para comparar dos valores y evaluar si se cumple o no una condición. Los operadores de comparación devuelven un valor booleano (verdadero o falso) como resultado.

Algunos de los operadores de comparación en C# son:

- ==: este operador se utiliza para comprobar si dos valores son iguales.
- !=: este operador se utiliza para comprobar si dos valores son diferentes.
- >: este operador se utiliza para comprobar si un valor es mayor que otro.
- <: este operador se utiliza para comprobar si un valor es menor que otro.
- >=: este operador se utiliza para comprobar si un valor es mayor o igual que otro.
- <=: este operador se utiliza para comprobar si un valor es menor o igual que otro.

Veamos algunos ejemplos de cómo se utilizan estos operadores en C#:

```
int edad = 25;
bool esMayorDeEdad = edad >= 18; // Devuelve true
bool esMenorDeEdad = edad < 18; // Devuelve false

string nombre = "Juan";
string apellido = "Perez";
bool tienenElMismoNombre = nombre == apellido; // Devuelve false
bool tienenDistintoNombre = nombre != apellido; // Devuelve true
```

En el primer ejemplo, se utiliza el operador $>=$ para comprobar si la variable *edad* es mayor o igual a 18. Como el valor de *edad* es 25, el resultado de la operación es *true*, lo que significa que la variable *esMayorDeEdad* toma el valor de *true*.

En el segundo ejemplo, se utiliza el operador $<$ para comprobar si la variable *edad* es menor que 18. Como el valor de *edad* es 25, el resultado de la operación es *false*, lo que significa que la variable *esMenorDeEdad* toma el valor de *false*.

En el tercer ejemplo, se comparan dos variables de tipo *string*, *nombre* y *apellido*, utilizando el operador $==$. Como los valores de las variables son distintos, el resultado de la operación es *false*, lo que significa que la variable *tienenElMismoNombre* toma el valor de *false*.

En el cuarto ejemplo, se utiliza el operador $!=$ para comprobar si las variables *nombre* y *apellido* son distintas. Como los valores de las variables son distintos, el resultado de la operación es *true*, lo que significa que la variable *tienenDistintoNombre* toma el valor de *true*.

1.3. Bucles, Sentencias if y Switch

En C#, existen varios tipos de bucles que permiten repetir un bloque de código un determinado número de veces o mientras se cumpla una condición. A continuación, se describen los principales tipos de bucles en C#:

Bucle `for`

El bucle `for` permite repetir un bloque de código un número determinado de veces. Su sintaxis es la siguiente:

```
for (inicializacion; condicionn; expresion de actualizacion) {
// Codigo a repetir
}
```

La `inicialización` se utiliza para declarar y asignar valores a una variable de control que se utiliza en el bucle. La `condición` se evalúa al comienzo de cada iteración y si es verdadera, se ejecuta el código dentro del bucle. La `expresión de actualización` se utiliza para modificar el valor de la variable de control después de cada iteración.

Por ejemplo, el siguiente bucle `for` imprime los números del 1 al 10:

```
for (int i = 1; i <= 10; i++) {
Console.WriteLine(i);
}
```

Bucle foreach

El bucle foreach en C# es utilizado para iterar sobre los elementos de una colección o arreglo sin la necesidad de conocer el número de elementos previamente. Su sintaxis es la siguiente:

```
foreach (tipoDeElemento variable in coleccion) {
// Code
}
```

Donde `tipoDeElemento` es el tipo de datos de cada elemento de la colección, `variable` es una variable que se utiliza para representar cada elemento de la colección y `coleccion` es la colección o arreglo sobre el que se va a iterar.

El bucle foreach es una forma más sencilla y legible de iterar sobre los elementos de una colección o arreglo en comparación con el bucle for tradicional. Además, también evita errores comunes como desbordamientos de índice o intentos de acceder a elementos que no existen en la colección.

Te doy un ejemplo:

```
List<string> nombres = new List<string>() { "Juan", "Pedro", "Maria", "Lucas" };

foreach (string nombre in nombres){
    Console.WriteLine(nombre);
}
```

Bucle `while`

El bucle `while` permite repetir un bloque de código mientras se cumpla una condición. Su sintaxis es la siguiente:

```
while (condicion) {
// Codigo a repetir
}
```

La `condición` se evalúa al comienzo de cada iteración y si es verdadera, se ejecuta el código dentro del bucle. Si la condición es falsa, el bucle se detiene.

Por ejemplo, el siguiente bucle `while` imprime los números del 1 al 10:

```
int i = 1;
while (i <= 10) {
Console.WriteLine(i);
i++;
}
```

Bucle `do-while`

El bucle `do-while` permite repetir un bloque de código al menos una vez y mientras se cumpla una condición. Su sintaxis es la siguiente:

```
do {
// Codigo a repetir
} while (condicion);
```

El código dentro del bucle se ejecuta al menos una vez, y luego la `condición` se evalúa. Si es verdadera, se vuelve a ejecutar el código dentro del bucle. Si la condición es falsa, el bucle se detiene.

Por ejemplo, el siguiente bucle `do-while` imprime los números del 1 al 10:

```
int i = 1;
do {
Console.WriteLine(i);
i++;
} while (i <= 10);
```

Sentencia if - else if - else

La sentencia if-elif-else es una estructura condicional en programación que permite ejecutar diferentes bloques de código dependiendo del valor de una expresión booleana. La sintaxis en C# de la sentencia if-elif-else es la siguiente:

```
if (expresion_booleana1){
    // code
}
else if (expresion_booleana2){
    // code
}
else{
    // code
```

En esta estructura, se evalúa primero la expresión booleana1. Si esta expresión es verdadera, se ejecuta el bloque de código correspondiente. Si es falsa, se evalúa la expresión booleana2. Si esta es verdadera, se ejecuta el bloque de código correspondiente. Si es falsa, se ejecuta el bloque de código en la cláusula else.

Es importante destacar que la sentencia if-elif-else puede tener tantas cláusulas elif como sea necesario, y que las expresiones booleanas pueden ser complejas, incluyendo operadores lógicos y aritméticos, así como llamadas a funciones que devuelvan un valor booleano.

Un ejemplo de uso de la sentencia if-elif-else podría ser el siguiente:

```
int edad = 25;

if (edad < 18){
```

```
    Console.WriteLine("Eres menor de edad");
}
else if (edad >= 18 && edad < 65){
    Console.WriteLine("Eres adulto");
}
else{
    Console.WriteLine("Eres mayor de edad");
}
```

En la primera línea se declara la variable "edad" y se le asigna un valor de 25. Luego, se utiliza la sentencia if para evaluar si la edad es menor a 18, en cuyo caso se imprime por pantalla el mensaje "Eres menor de edad". Si la edad no es menor a 18, se evalúa si la edad se encuentra en el rango de 18 a 64 años utilizando la sentencia else if. Si la edad está dentro de ese rango, se imprime por pantalla el mensaje "Eres adulto". Si la edad no es menor a 18 y no se encuentra en el rango de 18 a 64 años, se asume que es mayor de 65 años y se imprime el mensaje "Eres mayor de edad" utilizando la sentencia else.

Sentencia switch

La sentencia switch es otra estructura condicional en C# que permite ejecutar diferentes bloques de código dependiendo del valor de una expresión. La sintaxis de la sentencia switch es la siguiente:

```
switch (expresion){
    case valor1:
        // code
        break;
    case valor2:
        // code
        break;
    // otros casos
    default:
        // code
}
```

En esta estructura, se evalúa la expresión y se compara su valor con los distintos valores especificados en los casos. Si la expresión coincide con alguno de los valores, se ejecuta el bloque de código correspondiente. Si no coincide con ninguno de los valores, se ejecuta el bloque de código en la cláusula default.

Es importante destacar que la sentencia switch solo puede utilizarse para comparar expresiones que se puedan evaluar como enteros, caracteres, cadenas de texto o enumeraciones.

Un ejemplo de uso de la sentencia switch podría ser el siguiente:

```
int diaDeLaSemana = 3;

switch (diaDeLaSemana){
    case 1:
```

```
        Console.WriteLine("Lunes");
        break;
    case 2:
        Console.WriteLine("Martes");
        break;
    case 3:
        Console.WriteLine("Miercoles");
        break;
    case 4:
        Console.WriteLine("Jueves");
        break;
    case 5:
        Console.WriteLine("Viernes");
        break;
    case 6:
        Console.WriteLine("Sabado");
        break;
    case 7:
        Console.WriteLine("Domingo");
        break;
    default:
        Console.WriteLine("Valor invalido");
        break;
}
```

El código que utiliza la sentencia switch se podría expresar en términos de if-else de la siguiente manera:

```
int diaDeLaSemana = 3;

if (diaDeLaSemana == 1){
    Console.WriteLine("Lunes");
}
else if (diaDeLaSemana == 2){
    Console.WriteLine("Martes");
}
else if (diaDeLaSemana == 3){
    Console.WriteLine("Miercoles");
}
else if (diaDeLaSemana == 4){
    Console.WriteLine("Jueves");
}
else if (diaDeLaSemana == 5){
    Console.WriteLine("Viernes");
}
else if (diaDeLaSemana == 6){
```

```
    Console.WriteLine("Sabado");
}
else if (diaDeLaSemana == 7){
    Console.WriteLine("Domingo");
}
else{
    Console.WriteLine("Valor invalido");
}
```

En ese sentido, podemos comparar cual es más legible o más útil, en el fondo hacen lo mismo, la diferencia es la legibilidad y la facilidad de mantenimiento del código.

En general, el uso de la sentencia switch es más adecuado cuando se tienen múltiples casos que deben ser evaluados. El código es más compacto y fácil de leer, ya que se evita tener una gran cantidad de sentencias if-else anidadas. Además, el uso de la sentencia switch permite una ejecución más rápida del código, ya que el compilador puede optimizar la estructura de la sentencia para hacerla más eficiente.

Por otro lado, el uso de una serie de sentencias if-else puede ser más adecuado cuando se tienen condiciones complejas y varias opciones que deben ser evaluadas. El código es más detallado y es más fácil de entender cómo se están evaluando las diferentes opciones. Sin embargo, el uso excesivo de sentencias if-else puede hacer que el código sea más difícil de leer y mantener, especialmente si hay muchas condiciones anidadas.

1.4. Metodos y funciones

En C#, un método es una acción que se puede realizar sobre un objeto o una instancia de una clase. Por otro lado, una función es una operación que se realiza y devuelve un valor.

Existen diferentes tipos de funciones y métodos en C#, a continuación, se describen algunos de ellos:

Funciones y métodos estáticos

Los métodos estáticos se definen en una clase y se pueden llamar sin necesidad de instanciar un objeto de dicha clase. Por lo tanto, se pueden usar en cualquier parte del código sin tener que crear un objeto primero. Por otro lado, una función estática es una función que no tiene acceso a las propiedades de un objeto y solo utiliza los parámetros que se le pasan.

A continuación, se muestra un ejemplo de cómo se define un método estático en C#:

```
public static int Sum(int a, int b){
    return a + b;
}
```

En este ejemplo, el método `Sum` se define como `static`, lo que significa que se puede llamar sin tener que crear una instancia de la clase. El método toma dos argumentos de tipo `int` y devuelve la suma

de estos dos valores.

Métodos de instancia

Los métodos de instancia son métodos que solo se pueden llamar en una instancia de una clase. Es decir, se tiene que crear un objeto primero antes de poder llamar al método.

A continuación, se muestra un ejemplo de cómo se define un método de instancia en C#:

```
public class Calculator{
    public int Sum(int a, int b){
        return a + b;
        }
    }
```

En este ejemplo, se define una clase llamada `Calculator` que contiene un método llamado `Sum`. Este método toma dos argumentos de tipo `int` y devuelve la suma de estos dos valores.

Métodos y funciones con parámetros opcionales

En C#, se pueden definir métodos y funciones que tengan parámetros opcionales. Esto significa que se pueden llamar sin especificar todos los argumentos, y los argumentos que no se especifican utilizarán valores predeterminados.

A continuación, se muestra un ejemplo de cómo se define un método con un parámetro opcional en C#:

```
public static void PrintName(string firstName, string lastName = ""){
    Console.WriteLine($"Nombre completo: {firstName} {lastName}");
    Console.WriteLine($"Hola soy {firstName} {lastName}");
}
```

En este ejemplo, el parámetro `lastName` se define como un parámetro opcional, lo que significa que se puede llamar al método sin especificar un valor para este parámetro. Si no se especifica un valor para `lastName`, se utilizará una cadena vacía como valor predeterminado.

Métodos y funciones con valor de retorno

En C#, tanto los métodos como las funciones pueden tener un valor de retorno. La principal diferencia entre un método y una función es que un método está asociado a un objeto o clase y puede modificar su estado, mientras que una función no tiene asociación con un objeto o clase y no puede modificar su estado.

Para declarar una función con valor de retorno, se utiliza la palabra clave **return** seguida del valor que se desea devolver. Por ejemplo, la siguiente función recibe dos números enteros y devuelve la suma de ellos:

```
int Sumar(int a, int b)
{
int resultado = a + b;
return resultado;
}
```

Para llamar a la función anterior y obtener su resultado, se puede hacer lo siguiente:

```
int resultado = Sumar(3, 4);
Console.WriteLine(resultado); // Imprime 7
```

También se pueden declarar métodos con valor de retorno de la misma manera. Por ejemplo, el siguiente método estático de una clase llamada **Calculadora** recibe dos números enteros y devuelve su suma:

```
public static int Sumar(int a, int b)
{
int resultado = a + b;
return resultado;
}
```

Para llamar a este método desde otro lugar del código, se utiliza el nombre de la clase seguido del operador punto y el nombre del método:

```
int resultado = Calculadora.Sumar(3, 4);
Console.WriteLine(resultado); // Imprime 7
```

Es importante destacar que el valor de retorno de una función o método debe ser del mismo tipo que el indicado en la declaración. Si se intenta devolver un valor de un tipo distinto, el compilador generará un error.

Existen otros tipos de funciones en C# como las funciones void, funciones anónimas y funciones lambda.

Las funciones void son aquellas que no retornan ningún valor, es decir, no se espera que devuelvan un resultado. En su lugar, estas funciones suelen realizar una tarea específica, como imprimir información en la consola o modificar un valor dentro de la aplicación.

Las funciones anónimas son aquellas que no tienen un nombre específico y se utilizan principalmente como argumentos de otras funciones. A menudo se utilizan en conjunción con delegados o eventos.

Las funciones lambda son un tipo especial de función anónima que se utilizan para crear expresiones que se pueden utilizar en lugar de métodos o delegados. Estas funciones son útiles para simplificar el código y mejorar la legibilidad.

En cuanto a los tipos de métodos, además de los métodos con valor de retorno, también existen los métodos void, que no retornan ningún valor, los métodos estáticos, que pertenecen a la clase y no a una instancia específica de la misma, y los métodos de instancia, que se ejecutan en una instancia específica de una clase.

En cuanto al nivel de acceso, los métodos y las variables pueden ser públicos, privados, protegidos o internos, lo que determina si se pueden acceder desde otras clases o dentro de la misma clase.

Por último, en función de los argumentos y la salida, podemos encontrar métodos y funciones con parámetros de entrada y sin ellos, así como con un número variable de argumentos. También pueden ser funciones y métodos genéricos, que aceptan tipos de datos específicos como argumentos y retornan valores del mismo tipo.

1.5. Cadenas

Las cadenas de texto son muy comunes en la programación, y en C# se cuenta con una gran cantidad de métodos para trabajar con ellas. Algunos de los métodos más utilizados son:

Concatenación de cadenas

El método más básico para trabajar con cadenas es la concatenación, que consiste en unir dos o más cadenas en una sola. En C#, se puede hacer con el operador + o con el método Concat:

```
string saludo = "Hola";
string nombre = "Juan";
string mensaje1 = saludo + " " + nombre + "!";
string mensaje2 = string.Concat(saludo, " ", nombre, "!");
```

Ambos métodos producirán la misma salida: "Hola Juan!".

Métodos de formato

C# también ofrece una serie de métodos para formatear cadenas de manera más sofisticada. Uno de ellos es el método Format, que permite crear una cadena con un formato predefinido, en el que se pueden incluir parámetros variables:

```
string saludo = "Hola";
string nombre = "Juan";
string mensaje1 = saludo + " " + nombre + "!";
string mensaje2 = string.Concat(saludo, " ", nombre, "!");
```

El método Format toma una cadena de formato, que contiene marcadores de posición (en este caso, los números entre llaves), y una serie de argumentos, que se sustituyen en los marcadores de posición. La salida será: “Hola Juan, tienes 25 años.”.

También se puede utilizar el operador $ para crear una cadena interpolada, que es una forma más compacta de utilizar la sintaxis de formato:

```
string saludo = "Hola";
string nombre = "Juan";
int edad = 25;
string mensaje = $"{saludo} {nombre}, tienes {edad} anos."; \\$
```

La salida será la misma que en el ejemplo anterior.

Métodos de búsqueda

C# también proporciona métodos para buscar y reemplazar cadenas dentro de otras. Por ejemplo, el método IndexOf busca la primera aparición de una subcadena dentro de otra cadena:

```
string texto = "La casa de Juan esta en la calle Mayor";
int posicion = texto.IndexOf("Juan");
```

El valor de posición será 12, que es la posición en la que empieza la palabra "Juan"dentro de la cadena.

También se puede utilizar el método Replace para reemplazar una subcadena por otra:

```
string texto = "La casa de Juan esta en la calle Mayor";
string nuevoTexto = texto.Replace("Juan", "Pedro");
```

La variable nuevoTexto contendrá la cadena "La casa de Pedro está en la calle Mayor".

Métodos de conversión

Por último, C# proporciona métodos para convertir cadenas en otros tipos de datos y viceversa. Uno de los métodos más útiles es el método Parse, que convierte una cadena en un valor numérico:

```
string numeroComoTexto = "123";
int numero = int.Parse(numeroComoTexto);
```

El valor de la variable número será 123, que es el valor numérico representado por la cadena.

También se puede utilizar el método ToString para convertir un valor numérico en una cadena:

```
int numero = 123;
string numeroComoTexto = numero.ToString();
```

StringBuilder

StringBuilder es una clase en C# que se utiliza para manipular cadenas de caracteres de manera más eficiente que con la clase String. La clase StringBuilder tiene métodos que permiten agregar, insertar, eliminar y reemplazar caracteres en una cadena. Además, permite trabajar con cadenas de caracteres mutables (modificables), a diferencia de la clase String, que trabaja con cadenas inmutables (no modificables).

En C#, para utilizar la clase StringBuilder es necesario importar la librería System.Text. Por lo tanto, al comienzo del archivo de código se debe incluir la siguiente línea:

```
using System.Text;
```

La clase StringBuilder es especialmente útil cuando se trabaja con cadenas grandes y complejas, ya que se pueden realizar múltiples operaciones en la misma cadena sin tener que crear una nueva cadena cada vez, lo que ahorra memoria y tiempo de ejecución.

Aquí hay un ejemplo de cómo se utiliza StringBuilder:

```
StringBuilder sb = new StringBuilder("Hola ");
sb.Append("mundo!");
Console.WriteLine(sb.ToString()); // Salida: "Hola mundo!"
```

En este ejemplo, se crea un nuevo objeto StringBuilder con la cadena "Hola ". Luego, se utiliza el método Append para agregar la cadena "mundo!" al final de la cadena existente. Finalmente, se utiliza el método ToString para obtener la cadena resultante y mostrarla por consola.

El método ToString

El método ToString es uno de los métodos más utilizados en C#, ya que nos permite convertir cualquier objeto a su representación en formato de cadena. Este método se utiliza principalmente para imprimir valores en la pantalla o para concatenar cadenas.

La sintaxis básica del método ToString es la siguiente:

```
string texto = objeto.ToString();
```

Donde *objeto* es el objeto que se desea convertir a una cadena y *texto* es la cadena resultante.

Es importante tener en cuenta que el método ToString se puede sobrescribir en una clase personalizada para proporcionar una representación de cadena personalizada del objeto. De esta manera, podemos controlar cómo se mostrará el objeto al llamar al método ToString en lugar de la representación predeterminada.

Veamos un ejemplo sencillo de cómo podemos utilizar el método ToString en C#:

```
int num = 123;
string cadena = num.ToString();
Console.WriteLine(cadena); // muestra "123" en la consola
```

En este ejemplo, el método ToString se utiliza para convertir el número entero *num* a una cadena. La cadena resultante se almacena en la variable *cadena* y se muestra en la consola utilizando el método WriteLine de la clase Console.

En resumen, el método ToString es una herramienta útil para convertir objetos a su representación en formato de cadena. Si se desea personalizar la forma en que se muestra un objeto, es posible sobrescribir este método en una clase personalizada.

Otros métodos útiles en cadenas

En C#, las cadenas tienen una serie de métodos útiles que pueden ser utilizados para realizar diversas operaciones. Algunos de los métodos más comunes son los siguientes:

- **Insert**: este método permite insertar una cadena en una posición específica dentro de otra cadena. Se puede utilizar de la siguiente manera:

```
string cadena = "Hola mundo!";
cadena = cadena.Insert(5, " querido");
Console.WriteLine(cadena); // Output: Hola querido mundo!
```

- **EndsWith**: este método comprueba si una cadena termina con un valor específico. Devuelve un valor booleano que indica si es verdadero o falso. Se puede utilizar de la siguiente manera:

```
string cadena = "Hola mundo!";
bool resultado = cadena.EndsWith("!");
Console.WriteLine(resultado); // Output: True
```

- **LastIndexOf**: este método busca la última aparición de una cadena dentro de otra cadena. Devuelve un entero que indica la posición donde se encuentra la última aparición de la cadena. Se puede utilizar de la siguiente manera:

```
string cadena = "Hola mundo!";
int posicion = cadena.LastIndexOf("o");
Console.WriteLine(posicion); // Output: 7
```

- **Trim**: este método elimina los espacios en blanco del principio y final de una cadena. Se puede utilizar de la siguiente manera:

```
string cadena = " Hola mundo! ";
cadena = cadena.Trim();
Console.WriteLine(cadena); // Output: Hola mundo!
```

- **string.Format**: este método permite dar formato a una cadena mediante la inclusión de valores variables. Se puede utilizar de la siguiente manera:

```
string nombre = "Juan";
int edad = 35;
string resultado = string.Format("Mi nombre es {0} y tengo {1} years.", nombre, edad);
Console.WriteLine(resultado); // Output: Mi nombre es Juan y tengo 35 years.
```

- **DateTime.ToString**: este método permite dar formato a una fecha y hora en una cadena de texto. Se puede utilizar de la siguiente manera:

```
DateTime fecha = new DateTime(2023, 4, 25, 15, 30, 0);
string resultado = fecha.ToString("dd/MM/yyyy hh:mm:ss");
Console.WriteLine(resultado); // Output: 25/04/2023 03:30:00 PM
```

1.6. Colecciones

La programación en C# proporciona varios tipos de colecciones para almacenar y manipular conjuntos de datos. Algunos de los tipos de colecciones más comunes en C# son los siguientes:

ArrayLists

Un ArrayList es una colección de objetos que se pueden agregar o eliminar dinámicamente en tiempo de ejecución. El tamaño de un ArrayList puede crecer o disminuir automáticamente según sea necesario. Los elementos en un ArrayList se almacenan en orden secuencial, lo que significa que cada elemento tiene un índice numérico que representa su posición en la lista.

Para utilizar un ArrayList en C#, es necesario importar el espacio de nombres System.Collections. A continuación, se muestra un ejemplo de cómo crear y usar un ArrayList:

```
using System.Collections;

ArrayList miArrayList = new ArrayList();

miArrayList.Add("Hola");
miArrayList.Add(123);
miArrayList.Add(true);
miArrayList.Add(4.56);

foreach (object elemento in miArrayList){
    Console.WriteLine(elemento);
    }
```

Pilas o stacks

Una pila (o stack, en inglés) es una colección de elementos que se agregan y eliminan por el mismo extremo, conocido como “extremo superior” o “cima de la pila”. Esto significa que el último elemento en agregarse a la pila es el primero en salir de ella.

En C#, una pila se puede implementar utilizando la clase Stack, que se encuentra en el espacio de nombres System.Collections. Aquí hay un ejemplo:

```
using System.Collections;

Stack miPila = new Stack();

miPila.Push("uno");
miPila.Push("dos");
miPila.Push("tres");

Console.WriteLine("Cima de la pila: " + miPila.Peek());

while (miPila.Count > 0){
    Console.WriteLine(miPila.Pop());
    }
```

Colas o Queues

Una cola (o queue, en inglés) es una colección de elementos que se agregan por un extremo y se eliminan por el otro, conocido como "extremo final". Esto significa que el primer elemento en agregarse a la cola es el primero en salir de ella.

En C#, una cola se puede implementar utilizando la clase Queue, que se encuentra en el espacio de nombres System.Collections. Aquí hay un ejemplo:

```
using System.Collections;

Queue miCola = new Queue();

miCola.Enqueue("uno");
miCola.Enqueue("dos");
miCola.Enqueue("tres");

Console.WriteLine("Primer elemento de la cola: " + miCola.Peek());

while (miCola.Count > 0){
    Console.WriteLine(miCola.Dequeue());
    }
```

HashTables

Hashtable es una estructura de datos que permite almacenar una colección de pares clave-valor. Cada elemento en un Hashtable está identificado por una clave única y asociado con un valor.

Para crear una Hashtable en C#, se utiliza la clase Hashtable que se encuentra en el espacio de nombres System.Collections.

A continuación, se muestra un ejemplo de cómo crear una Hashtable y agregar elementos a ella:

```
Hashtable ht = new Hashtable();
ht.Add("Juan", 25);
ht.Add("Maria", 30);
ht.Add("Pedro", 40);
```

En este ejemplo, se ha creado una Hashtable llamada ht y se han agregado tres elementos a ella. La clave de cada elemento es un string que representa el nombre de una persona, y el valor es un entero que representa su edad.

Para acceder a los elementos de una Hashtable, se utiliza la clave correspondiente. Por ejemplo:

```
int edadMaria = (int)ht["Maria"];
```

En este ejemplo, se ha accedido al elemento cuya clave es "Maria" y se ha almacenado su valor en la variable edadMaria. Como el valor almacenado es un entero, se ha utilizado un casting para convertirlo a ese tipo de datos.

Es importante tener en cuenta que al utilizar una Hashtable, las claves deben ser únicas. Si se intenta agregar un elemento con una clave que ya existe en la Hashtable, se producirá una excepción.

2. Programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación que se basa en la utilización de objetos para representar y manipular los datos y su comportamiento en un programa. En POO, los objetos son instancias de una clase, que es una plantilla que define las características y el comportamiento de los objetos.

El enfoque en objetos permite la modularidad y el encapsulamiento, lo que hace que el código sea más fácil de mantener y extender, el encapsulamiento permite modelar ideas de seguridad y niveles de acceso entre clases. Además, la POO hace uso de conceptos como la herencia y el polimorfismo, que permiten una mayor flexibilidad y reutilización de código.

Los pilares de la programación orientada a objetos son:

1. Abstracción: este pilar se refiere a la capacidad de representar objetos del mundo real de manera simplificada, enfocándonos en las características importantes y dejando de lado lo que no es relevante para nuestro objetivo. En POO, una clase es una abstracción de un objeto.

2. Encapsulamiento: este pilar se refiere a la capacidad de ocultar ciertos datos y comportamientos de un objeto, de manera que sólo se puedan acceder a ellos mediante una interfaz definida. En POO, esto se logra utilizando modificadores de acceso (public, private, protected) para controlar la visibilidad de los miembros de una clase.

3. Herencia: este pilar se refiere a la capacidad de crear nuevas clases a partir de clases existentes, heredando sus características y comportamientos. En POO, una clase hija hereda los miembros de la clase padre y puede añadir o sobrescribir miembros según sus necesidades.

4. Polimorfismo: este pilar se refiere a que diferentes objetos pueden responder al "mismo" metodo de diferentes maneras. Es usual que hayan clases que tengan el metodo "Get", que se encarga de mostrar los atributos de la clase en cuestiónm, Es inmediato entonces entender que dos objetos podrán responder "al mismo metodo" de distintas formas.

2.1. Clases y objetos

En programación orientada a objetos, una clase es una plantilla o modelo que describe las propiedades y métodos que tendrán los objetos que son instancia de ella. Por lo tanto, una clase es un concepto abstracto que define la estructura y comportamiento de los objetos.

Por otro lado, un objeto es una instancia concreta de una clase. En otras palabras, es una representación tangible de la clase. Cuando se crea un objeto, se pueden establecer sus propias propiedades y métodos, aunque hereda las características definidas por la clase.

Veamos un ejemplo de cómo se declara una clase en C#:

```
public class Person {
    // atributos encapsulados
    private string name;
    private int age;

    // metodo constructor (luego veremos con detalle este concepto)
    public Person(string name, int age){
        this.name = name;
        this.age = age;
    }

    // metodo
    public void greet(){
        Console.WriteLine("Hi, my name is " + name + " and I'm " + age + " years old.");
    }
}
```

En este ejemplo, se ha creado una clase llamada Persona con dos propiedades (nombre y edad) y dos métodos (el método constructor y el método Saludar). El método constructor se utiliza para inicializar las propiedades cuando se crea un objeto, mientras que el método Saludar se utiliza para imprimir un mensaje en la consola.

Para crear un objeto de la clase Persona, se puede hacer lo siguiente:

```
Person person1 = new Person("Juan", 30);
```

En este caso, se está creando un objeto llamado persona1 de la clase Persona, y se están pasando como argumentos al método constructor los valores "Juan" y 30 para las propiedades nombre y edad, respectivamente.

Una vez creado el objeto, se puede llamar a su método Saludar de la siguiente manera:

```
persona1.greet();
```

Este código imprimirá en la consola el mensaje "Hi, my name is Juan and I'm 30 years old.", utilizando los valores de las propiedades del objeto creado.

2.2. Herencia

La herencia es un concepto importante en la programación orientada a objetos que permite que una clase adquiera las propiedades y métodos de otra clase existente. La clase existente se conoce como la clase base o superclase, mientras que la clase que hereda de ella se conoce como la clase derivada o

subclase. Puede ser útil para crear una jerarquía de clases que comparten propiedades y comportamientos comunes. Al heredar de una clase base, una clase derivada puede reutilizar el código existente y extender o modificar su comportamiento. Además de la herencia de clases, también existen las interfaces funcionales y las clases abstractas. Las interfaces funcionales son un tipo especial de interfaz que define exactamente un método abstracto y pueden ser implementadas por cualquier clase que proporcione una implementación de este método. Las clases abstractas son clases que no se pueden instanciar directamente y deben ser extendidas por clases derivadas. Las clases abstractas pueden contener métodos abstractos, que deben ser implementados por cualquier clase derivada, así como métodos concretos que se pueden usar directamente en la clase derivada.

Te muestro un ejemplo de cómo la herencia puede hacer que el código sea más modular:

```
public abstract class Animal{
    public abstract string Sound();
    }

public class Dog : Animal{
    public override string Sound(){
        return "Woof!";
    }
}

public class Cat : Animal{
    public override string Sound(){
        return "Meow!";
    }
}

    public class AnimalSound{
    public static void Main(){
        Animal dog = new Dog();
        Animal cat = new Cat();

        Console.WriteLine(dog.Sound());
        Console.WriteLine(cat.Sound());
        }
}
```

En este ejemplo, se define una clase abstracta llamada Animal que tiene un método abstracto llamado Sound(). La clase Dog y la clase Cat heredan de la clase Animal y proporcionan una implementación del método Sound(). La clase AnimalSound utiliza las clases Dog y Cat para imprimir los sonidos de un perro y un gato en la consola. Este código es más modular y fácil de mantener, ya que el comportamiento común se define en la clase Animal y las clases derivadas pueden proporcionar su propia implementación única del método Sound().

2.3. Abstracción

La abstracción es un concepto dentro de la programación orientada a objetos que se refiere al proceso mediante el cual se toman los aspectos de una entidad, asociada al objeto que queremos utilizar, y así pensar en la clase (plantilla) que al instanciar generará este objeto en cuestión. No se abstraen todos los aspectos que una entidad tiene en el mundo real, sino solo los que son realmente pertinentes en la aplicación en cuestión.

Por ejemplo, si estamos escribiendo una aplicación bancaria, podríamos crear una clase "CuentaBancaria" que tenga propiedades como "balance" y "número de cuenta", así como métodos como "depósito" y "retiro". Un programador que utilice esta clase puede utilizar estos métodos y propiedades sin necesidad de conocer los detalles internos de cómo se implementan. Además, entendemos que una cuenta bancaria puede tener otras propiedades, como un dueño, y este a su vez un numero relativamente grande de propiedades, que realmente en este contexto no nos van a interesar en este contexto.

2.4. Encapsulamiento

El encapsulamiento es otro concepto clave en la programación orientada a objetos que se refiere a la ocultación de los detalles internos de una clase o estructura de datos, de manera que solo se expongan los métodos y propiedades que son necesarios para interactuar con ella. En otras palabras, se protege la información dentro de la clase para que no pueda ser modificada por otros objetos externos.

Por ejemplo, en la clase "CuentaBancaria" mencionada anteriormente, podríamos encapsular la propiedad "balance" y el método "retiro", de manera que solo se pueda acceder a ellos desde dentro de la clase. Esto evita que otros objetos externos a la clase modifiquen directamente el balance de la cuenta o realicen retiros no autorizados.

La sintaxis para encapsular una propiedad en C# es la siguiente:

```
private double balance;
public double Balance
{
    get { return balance; }
    private set { balance = value; }
}
```

En este ejemplo, la propiedad "balance" es privada, lo que significa que solo se puede acceder a ella desde dentro de la clase. Sin embargo, hemos creado un método público llamado "Balance" que se utiliza para obtener el valor actual de la propiedad. También hemos creado un método privado "set" para permitir que la propiedad sea modificada desde dentro de la clase, pero no desde fuera de ella.

Otro ejemplo de encapsulamiento puede ser un método que calcule el interés de una cuenta bancaria, el cual puede ser un proceso complejo que involucre varios cálculos. Este método se puede encapsular en la clase "CuentaBancaria" y solo exponer el resultado final para ser utilizado por otros objetos externos. Esto

hace que el código sea más modular y fácil de mantener.

2.5. Polimorfismo

El polimorfismo es un concepto clave en la programación orientada a objetos que se refiere a la capacidad de objetos de diferentes clases de responder al mismo mensaje o método de diferentes maneras. Esto significa que un método puede ser implementado de manera diferente por diferentes clases, pero su llamada por parte del objeto no varía, es decir, el método se llama de la misma manera independientemente de la clase a la que pertenece el objeto.

Un ejemplo común de polimorfismo es la función "draw()" en un programa de gráficos. Diferentes objetos gráficos, como un círculo, un cuadrado o un triángulo, podrían tener su propia implementación de la función "draw()", que dibuja el objeto correspondiente. A pesar de que cada objeto tiene su propia implementación de la función "draw()", se puede llamar a la función de la misma manera para todos los objetos, lo que permite una programación más modular y flexible.

El polimorfismo se puede implementar en POO utilizando la herencia y las interfaces. En la herencia, una clase hija puede tener una implementación diferente de un método heredado de la clase padre. En las interfaces, varias clases pueden implementar una misma interfaz, cada una con su propia implementación de los métodos de la interfaz.

En términos de sintaxis, el polimorfismo se logra a través de la creación de clases con métodos que tienen la misma firma (nombre y parámetros) pero diferentes implementaciones. Cuando se llama a un método en un objeto, se ejecuta la implementación correspondiente de ese objeto.

Te presento un ejemplo:

```
using System;

public class Animal
{
    public virtual void HacerSonido()
    {
        Console.WriteLine("Hace algun sonido");
    }
}

public class Perro : Animal
{
    public override void HacerSonido()
    {
        Console.WriteLine("Ladra");
    }
}

```

```
public class Gato : Animal
{
    public override void HacerSonido()
    {
        Console.WriteLine("Maulla");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Animal miAnimal = new Animal();
        miAnimal.HacerSonido();  // output: "Hace algun sonido"

        miAnimal = new Perro();
        miAnimal.HacerSonido();  // output: "Ladra"

        miAnimal = new Gato();
        miAnimal.HacerSonido();  // output: "Maulla"
    }
}
```

En este ejemplo, tenemos una clase Animal que tiene un método virtual HacerSonido(). Luego creamos dos clases, Perro y Gato, que heredan de Animal y sobrescriben el método HacerSonido(). Finalmente, en el método Main creamos instancias de Animal, Perro y Gato y llamamos al método HacerSonido(). Dependiendo del tipo de objeto al que se hace referencia, se invoca la implementación correspondiente del método HacerSonido(). Esto es un ejemplo de polimorfismo en acción.

Estructuras vs clases

Las estructuras son muy prácticas cuando se desea manejar datos que tienen un vínculo entre sí. Por ejemplo, en una aplicación contable, los datos relativos a los clientes (código cliente, apellido, nombre, dirección) se pueden gestionar más fácilmente bajo la forma de una estructura que por variables individuales.

- Las estructuras son tipos de valor, mientras que las clases son tipos de referencia. Esto significa que cuando una variable de estructura se asigna a otra variable, se realiza una copia de los datos de la estructura, mientras que cuando se asigna una variable de clase a otra, se está haciendo referencia al mismo objeto en memoria.

- Las estructuras usan la asignación de pila, lo que significa que se almacenan en la memoria del programa que se encuentra en la parte superior de la pila. Las clases, por otro lado, usan la asignación del montón, lo que significa que se almacenan en la memoria dinámica del programa.

- Los miembros de una estructura son públicos de forma predeterminada, mientras que en una clase, solo los métodos y las propiedades públicas son públicos de forma predeterminada.
- Una estructura no requiere un constructor, mientras que una clase sí lo requiere. Además, las estructuras solo pueden tener constructores no compartidos si toman parámetros, mientras que las clases pueden tener constructores con o sin parámetros.

Te dejo un ejemplo:

```
using System;

namespace EjemploEstructuraYClase
{
    // Declaracndo de una estructura
    struct PuntoEstructura
    {
        public int X;
        public int Y;

        public PuntoEstructura(int x, int y)
        {
            X = x;
            Y = y;
        }
    }

    // Declaracndo de una clase
    class PuntoClase
    {
        public int X;
        public int Y;

        public PuntoClase(int x, int y)
        {
            X = x;
            Y = y;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // creando de un objeto de la estructura
            PuntoEstructura puntoEstructura = new PuntoEstructura(5, 10);

```

```
            // creando de un objeto de la clase
            PuntoClase puntoClase = new PuntoClase(10, 20);

            // Modificando del valor de X en el objeto de la estructura
            puntoEstructura.X = 7;

            // Modificando del valor de X en el objeto de la clase
            puntoClase.X = 15;

            // inprimir los valores de X e Y en el objeto de la estructura
            Console.WriteLine("PuntoEstructura: X = {0}, Y = {1}", puntoEstructura.X, puntoEstructura.Y);

            // inprimir los valores de X e Y en el objeto de la clase
            Console.WriteLine("PuntoClase: X = {0}, Y = {1}", puntoClase.X, puntoClase.Y);
        }
    }
}
```

En este ejemplo, se define una estructura PuntoEstructura y una clase PuntoClase que tienen los mismos campos X e Y. Luego, se crean objetos de ambas y se modifica el valor del campo X en cada uno de ellos. Finalmente, se imprimen los valores de los campos X e Y de ambos objetos.

La principal diferencia entre una estructura y una clase es que las estructuras son tipos de valor y las clases son tipos de referencia. Esto significa que cuando se crea un objeto de una estructura, se almacena en la pila de la memoria del programa, mientras que cuando se crea un objeto de una clase, se almacena en el montón de la memoria del programa y se guarda una referencia (un puntero) a su ubicación en la memoria. En este ejemplo, podemos ver que cuando se modifica el valor de X en el objeto de la estructura, se modifica directamente el valor almacenado en la pila. En cambio, cuando se modifica el valor de X en el objeto de la clase, se modifica el valor almacenado en el montón, y se actualiza la referencia a ese objeto en el programa.

2.6. ¿Cómo operar con clases?

Para trabajar con clases en la programación orientada a objetos, es necesario definir los atributos y métodos que conforman la clase. Los atributos son las características o datos que tiene un objeto de esa clase, mientras que los métodos son las acciones que se pueden realizar con ese objeto.

Por ejemplo, si estamos creando una clase “Perro”, podríamos definir los atributos “nombre”, “raza” y “edad”, y los métodos “ladrar” y “correr”. La sintaxis para definir una clase en C# sería la siguiente:

```
public class Perro {
    // atributos
    public string nombre;
    public string raza;
```

```
    public int edad;

    // metodos
    public void ladrar() {
        Console.WriteLine("Guau");
    }

    public void correr() {
        Console.WriteLine("Estoy corriendo");
    }
    }
```

En este ejemplo, la clase “Perro” tiene tres atributos (nombre, raza y edad) y dos métodos (ladrar y correr). Los atributos son públicos, lo que significa que se pueden acceder desde cualquier parte del código. Sin embargo, esto puede no ser deseable en algunos casos, por lo que se puede usar el encapsulamiento para proteger los datos de la clase.

Para encapsular los datos de una clase, se pueden utilizar los modificadores de acceso “private”, “protected” o “internal” para controlar quién puede acceder a los atributos y métodos de la clase. Por ejemplo, podríamos modificar la clase “Perro” para que los atributos sean privados y solo se puedan acceder a través de métodos públicos:

```
public class Perro {
    // atributos privados
    private string nombre;
    private string raza;
    private int edad;

    // metodos publicos para acceder a los atributos
    public string GetNombre() {
        return nombre;
    }

    public void SetNombre(string nombre) {
        this.nombre = nombre;
    }

    public string GetRaza() {
        return raza;
    }

    public void SetRaza(string raza) {
        this.raza = raza;
    }

```

```
    public int GetEdad() {
        return edad;
    }

    public void SetEdad(int edad) {
        this.edad = edad;
    }

    // metodos publicos
    public void Ladrar() {
        Console.WriteLine("Guau");
    }

    public void Correr() {
        Console.WriteLine("Estoy corriendo");
    }
}
```

En este ejemplo, los atributos “nombre”, “raza” y “edad” son privados y solo se pueden acceder a través de los métodos “Get” y “Set” correspondientes. Los métodos “Ladrar” y “Correr” siguen siendo públicos y se pueden acceder desde cualquier parte del código.

Para acceder a los métodos y atributos de una clase en la programación orientada a objetos, primero se debe instanciar la clase, lo que crea un objeto que es una instancia de esa clase. Luego, se puede acceder a los métodos y atributos a través de ese objeto utilizando el operador de punto (.) seguido del nombre del método o atributo.

En el ejemplo anterior de la clase “Perro” que has proporcionado, los métodos y atributos están definidos con modificadores de acceso “public” y “private”. Esto significa que los atributos “nombre”, “raza” y “edad” son privados, lo que significa que solo se pueden acceder a ellos desde dentro de la propia clase. Sin embargo, hay métodos “Get” y “Set” públicos para cada uno de estos atributos, lo que permite a otros objetos acceder y modificar estos atributos desde fuera de la clase.

Para establecer el nombre de un objeto de la clase “Perro”, se puede llamar al método “SetNombre” y pasar el nombre deseado como argumento:

```
    Perro miPerro = new Perro();
    miPerro.SetNombre("Fido");
```

Para recuperar el valor del atributo “nombre”, se puede llamar al método “GetNombre”:

```
    string nombreDelPerro = miPerro.GetNombre();
```

También hay dos métodos públicos adicionales, “Ladrar” y “Correr”, que pueden ser llamados para realizar las acciones correspondientes en un objeto de la clase “Perro”.

En general, el uso de métodos “Get” y “Set” públicos para acceder y modificar atributos privados se conoce como encapsulamiento. Esto ayuda a asegurar que los datos de la clase sean manejados de manera

controlada y consistente, lo que puede ser importante para evitar errores y garantizar la integridad de los datos.

2.7. Niveles de acceso

En la programación orientada a objetos, los niveles de acceso determinan la visibilidad de los miembros de una clase (atributos, métodos, propiedades) desde el exterior de la misma.

Existen 4 niveles de acceso en C#:

- **Public:** El miembro es accesible desde cualquier parte del programa.
- **Private:** El miembro solo es accesible desde dentro de la misma clase.
- **Protected:** El miembro es accesible desde dentro de la misma clase y desde las clases derivadas (heredadas).
- **Internal:** El miembro es accesible desde cualquier clase del mismo ensamblado (assembly). Un ensamblado es un archivo .dll o .exe que contiene uno o varios módulos.

La sintaxis para definir el nivel de acceso de un miembro es mediante los modificadores de acceso **public**, **private**, **protected** o **internal**, seguidos del tipo de miembro y su nombre. Por ejemplo:

```
public class MiClase {
    public int miVariablePublica;
    private int miVariablePrivada;
    protected int miVariableProtegida;
    internal int miVariableInterna;
}
```

En este ejemplo, se han declarado cuatro variables de diferentes niveles de acceso en una clase llamada `MiClase`. La variable `miVariablePublica` es pública y, por tanto, puede ser accedida desde cualquier parte del programa. La variable `miVariablePrivada` es privada, por lo que solo es accesible desde dentro de la misma clase. La variable `miVariableProtegida` es protegida, lo que significa que es accesible desde dentro de la misma clase y desde cualquier clase derivada de ella. La variable `miVariableInterna` es interna, por lo que es accesible desde cualquier clase dentro del mismo ensamblado, un ensamblado es una colección de tipos y recursos compilados para funcionar en conjunto y formar una unidad lógica de funcionalidad. Los ensamblados adoptan la forma de un archivo ejecutable (.exe) o de biblioteca de vínculos dinámicos (.dll), y son los bloques de creación de las aplicaciones.

Es importante destacar que el uso adecuado de los niveles de acceso contribuye a escribir código más seguro y fácil de mantener, ya que restringe el acceso a los miembros que no deberían ser modificados o accedidos desde fuera de la clase.

2.8. Clases Abstractas

En la programación orientada a objetos, una clase abstracta es una clase que no puede ser instanciada y solo puede ser utilizada como una clase base para otras clases. Es decir, una clase abstracta solo puede ser heredada y sus métodos abstractos deben ser implementados en las clases derivadas. Por otro lado, una clase virtual es una clase que puede ser instanciada, pero sus métodos virtuales pueden ser reemplazados por las clases derivadas.

A continuación, se muestra un ejemplo de una clase abstracta Animal con un método abstracto HacerSonido:

```
public abstract class Animal {
    public abstract void HacerSonido();
}

public class Perro : Animal {
    public override void HacerSonido() {
        Console.WriteLine("Guau guau!");
    }
}

public class Gato : Animal {
    public override void HacerSonido() {
        Console.WriteLine("Miau miau!");
    }
}
```

En este ejemplo, la clase Animal es una clase abstracta que define un método abstracto HacerSonido. Las clases Perro y Gato heredan de la clase Animal y deben implementar el método HacerSonido.

Por otro lado, las clases virtuales permiten la creación de métodos virtuales que pueden ser reemplazados por las clases derivadas. A continuación, se muestra un ejemplo de una clase virtual Figura con un método virtual CalcularArea:

```
public class Figura {
    public virtual double CalcularArea() {
        return 0;
    }
}

public class Circulo : Figura {
    private double radio;

    public Circulo(double radio) {
        this.radio = radio;
    }

```

```
    public override double CalcularArea() {
        return Math.PI * radio * radio;
    }
    public Circulo(double radio) {
        this.radio = radio;
    }

    public override double CalcularArea() {
        return Math.PI * radio * radio;
    }
    public Rectangulo(double baseFigura, double altura) {
        this.baseFigura = baseFigura;
        this.altura = altura;
    }

    public override double CalcularArea() {
        return baseFigura * altura;
    }
}
```

En este ejemplo, la clase Figura es una clase virtual que define un método virtual CalcularArea. Las clases Circulo y Rectangulo heredan de la clase Figura y reemplazan el método CalcularArea con su propia implementación.

2.9. Clases selladas y estáticas

En C#, las clases selladas y estáticas son dos conceptos que se utilizan para restringir la herencia y la creación de objetos, respectivamente.

2.9.1. Clases selladas

Las clases selladas son aquellas que no se pueden heredar. Es decir, una vez definida una clase sellada, no se pueden crear subclases a partir de ella. Para declarar una clase sellada, se utiliza la palabra clave sealed. Por ejemplo:

```
sealed class MiClaseSellada {
// Codigo de la clase
}
```

En este caso, la clase MiClaseSellada es sellada y no se puede heredar de ella.

Las clases selladas son útiles cuando queremos evitar que se modifique el comportamiento de una clase en subclases o cuando queremos garantizar que una clase se mantendrá inmutable.

2.9.2. Clases estáticas

Las clases estáticas son aquellas que no se pueden instanciar. Es decir, no se pueden crear objetos a partir de ellas. En lugar de eso, los miembros de una clase estática se acceden utilizando el nombre de la clase. Para declarar una clase estática, se utiliza la palabra clave static. Por ejemplo:

```
static class MiClaseEstatica {
public static void MiMetodoEstatico() {
// Codigo del metodo
}
}
```

En este caso, la clase MiClaseEstatica es estática y el método MiMetodoEstatico se puede llamar utilizando el nombre de la clase:

```
MiClaseEstatica.MiMetodoEstatico();
```

Las clases estáticas son útiles cuando queremos agrupar métodos y propiedades relacionados, pero no necesitamos crear instancias de la clase. También son útiles cuando queremos evitar el gasto de memoria que implica crear múltiples instancias de una clase.

2.10. Metodos de extensión, constructores y destructores

Los métodos de extensión, los constructores y los destructores son conceptos importantes en la programación orientada a objetos.

Métodos de extensión

Un método de extensión es un método estático que se define en una clase estática y que se utiliza para agregar funcionalidad a una clase existente sin tener que modificar la clase original. Los métodos de extensión son útiles cuando no se tiene acceso al código fuente de la clase original o cuando se desea mantener la compatibilidad con versiones anteriores de la clase.

La sintaxis de un método de extensión es similar a la de un método normal, pero con la adición de la palabra clave "this.ªntes del primer parámetro. El primer parámetro es el objeto al que se aplicará el método de extensión.

Un ejemplo de un método de extensión en C# que agrega una función de “reverse” a la clase “string” sería:

```
public static class StringExtensions {
public static string Reverse(this string str) {
    char[] chars = str.ToCharArray();
    Array.Reverse(chars);
    return new string(chars);
    }
```

```
}

// Uso del metodo de extension
string s = "Hola mundo";
string reversed = s.Reverse();
Console.WriteLine(reversed);
```

Constructores

Un constructor es un método especial que se llama automáticamente cuando se crea un objeto de una clase. El constructor se utiliza para inicializar los valores de los campos y propiedades de la clase.

En C#, el constructor tiene el mismo nombre que la clase y no tiene tipo de retorno. Puede haber varios constructores con diferentes parámetros, lo que se conoce como sobrecarga de constructores.

Un ejemplo de un constructor en C# sería:

```
public class Persona {
    public string Nombre { get; set; }
    public int Edad { get; set; }

    public Persona(string nombre, int edad) {
        this.Nombre = nombre;
        this.Edad = edad;
    }
}

// Uso del constructor
Persona persona = new Persona("Juan", 30);
```

Destructores

Un destructor es un método especial que se llama automáticamente cuando un objeto de una clase es destruido por el recolector de basura. El destructor se utiliza para liberar recursos no administrados como archivos, conexiones de bases de datos, etc.

En C#, el destructor tiene el mismo nombre que la clase y comienza con el símbolo -". El destructor no tiene parámetros ni tipo de retorno.

Un ejemplo de un destructor en C# sería:

```
public class ConexionBD {
    private string cadenaConexion;
    private SqlConnection conexion;

    public ConexionBD(string cadenaConexion) {
        this.cadenaConexion = cadenaConexion;
```

```
        this.conexion = new SqlConnection(cadenaConexion);
        this.conexion.Open();
    }

    ~ConexionBD() {
        this.conexion.Close();
    }
}

// Uso del destructor
ConexionBD conexion = new ConexionBD("cadena de conexion");
```

Métodos con cuerpo de expresión (Expression-bodied members)

En C#, los métodos, propiedades, constructores y destructores pueden ser definidos utilizando el cuerpo de expresión en lugar de un bloque de código. Esto es conocido como "métodos con cuerpo de expresión" o "expression-bodied members" en inglés.

Por ejemplo, en lugar de definir un método como:

```
public int Sumar(int a, int b) {
return a + b;
}
```

Se puede definir utilizando el cuerpo de expresión de la siguiente manera:

```
public int Sumar(int a, int b) => a + b;
```

Los métodos con cuerpo de expresión pueden ser útiles cuando se tiene una función simple que se puede definir en una sola línea de código. También pueden hacer que el código sea más legible al reducir la cantidad de caracteres utilizados.

También se pueden utilizar los métodos con cuerpo de expresión en propiedades, constructores y destructores. Por ejemplo:

```
public class Persona {
public string Nombre { get; set; }
public int Edad { get; set; }
public Persona(string nombre, int edad) => (Nombre, Edad) = (nombre, edad);

~Persona() => Console.WriteLine($"Adios {Nombre}"); \\$
}
```

En este ejemplo, el constructor utiliza el cuerpo de expresión para inicializar las propiedades "Nombre" y "Edad" en una sola línea de código. El destructor utiliza el cuerpo de expresión para escribir un mensaje de despedida en la consola.

2.11. Sobrecarga de métodos

La sobrecarga de métodos (en inglés, method overloading) es una técnica de programación en la que se definen varios métodos con el mismo nombre en una misma clase, pero con diferentes parámetros. En otras palabras, la sobrecarga de métodos permite tener varios métodos con el mismo nombre, pero que aceptan diferentes tipos de datos como argumentos.

Un ejemplo de sobrecarga de métodos sería una clase “Calculadora” que tenga varios métodos con el nombre “Sumar”, pero que acepten diferentes tipos de argumentos. Por ejemplo, podría tener los siguientes métodos:

```
public int Sumar(int a, int b)
{
    return a + b;
}

public float Sumar(float a, float b)
{
    return a + b;
}

public double Sumar(double a, double b)
{
    return a + b;
}
```

En este ejemplo, la clase “Calculadora” tiene tres métodos diferentes llamados “Sumar”, pero cada uno acepta un tipo de datos diferente: enteros, números en coma flotante y números de doble precisión. Esto permite que el usuario pueda llamar al método "Sumarçon diferentes tipos de datos y obtener el resultado esperado.

2.12. Interfaces

Una interfaz es un tipo de referencia similar a una clase que solo contiene miembros abstractos, como métodos, propiedades, eventos e indexadores. Las interfaces no proporcionan implementaciones para estos miembros, sino que definen una lista de miembros que deben estar presentes en cualquier clase o estructura que implemente la interfaz.

Las interfaces se utilizan para definir contratos que las clases deben cumplir, lo que significa que si una clase implementa una interfaz, debe proporcionar implementaciones para todos los miembros de la interfaz. Esto permite a los desarrolladores crear código más modular y extensible.

Por ejemplo, podemos definir una interfaz IVehiculo que contenga métodos como Arrancar(), Acelerar() y Frenar(). Luego, podemos tener varias clases que implementen esta interfaz, como Automovil,

Motocicleta y Camión. Cada una de estas clases proporcionaría una implementación única para los métodos de la interfaz, pero todas tendrían los mismos métodos.

La herencia múltiple se refiere a la capacidad de una clase de heredar de más de una clase o interfaz. En C#, las clases solo pueden heredar de una clase base, pero pueden implementar varias interfaces. Esto permite que una clase tenga comportamientos de múltiples fuentes sin la necesidad de heredar de múltiples clases, lo que puede resultar en una jerarquía de clases compleja y difícil de mantener.

Por ejemplo, podemos tener una clase VehiculoElectrico que implemente tanto la interfaz IVehiculo como una interfaz IElectrico, que contiene métodos para cargar y descargar la batería. De esta manera, podemos asegurarnos de que todos los vehículos eléctricos tengan los mismos métodos básicos de vehículo, mientras que también proporcionamos funcionalidad específica para los vehículos eléctricos.

Aquí un ejemplo:

```
public interface IVehiculo {
    void Arrancar();
    void Acelerar();
    void Frenar();
    void Apagar();
}

public class Automovil : IVehiculo {
    public void Arrancar() {
        Console.WriteLine("El automovil arranco.");
    }

    public void Acelerar() {
        Console.WriteLine("El automovil esta acelerando.");
    }

    public void Frenar() {
        Console.WriteLine("El automovil esta frenando.");
    }

    public void Apagar() {
        Console.WriteLine("El automovil se apago.");
    }
}

public class Moto : IVehiculo {
    public void Arrancar() {
        Console.WriteLine("La moto arranco.");
    }

    public void Acelerar() {
```

```
32          Console.WriteLine("La moto esta acelerando.");
33      }
34
35      public void Frenar() {
36          Console.WriteLine("La moto esta frenando.");
37      }
38
39      public void Apagar() {
40          Console.WriteLine("La moto se apago.");
41      }
42  }
```

En este ejemplo, la interfaz IVehiculo define los métodos Arrancar(), Acelerar(), Frenar() y Apagar(). La clase Automovil y Moto implementan esta interfaz y proporcionan una implementación concreta para estos métodos. Esto permite que las instancias de Automovil y Moto se traten como instancias de IVehiculo y se puedan pasar a cualquier método que acepte un parámetro del tipo IVehiculo.

La interfaz IVehiculo también se puede usar para realizar herencia múltiple de manera limitada en C#, ya que una clase puede implementar múltiples interfaces. Esto permite que una clase tenga la funcionalidad de varias interfaces diferentes, lo que puede ser útil en situaciones donde una clase necesita proporcionar comportamiento para varias funcionalidades diferentes.

3. Conceptos avanzados

A medida que se avanza en el aprendizaje de la programación en C#, se encuentran conceptos más avanzados que pueden mejorar la calidad y eficiencia de nuestro código. Estos conceptos incluyen delegados, métodos anónimos, tipos dinámicos, enumerados, espacios de nombres, tuplas, diccionarios, boxing y unboxing, serialización JSON, eventos, tratamiento de excepciones, trabajo con archivos y streams, y programación asíncrona.

Cada uno de estos temas tiene su propia utilidad y aplicabilidad en diferentes escenarios y situaciones de programación. En esta sección, se explicará cada uno de ellos con ejemplos prácticos y claros para que puedas entender su funcionamiento y cómo puedes aplicarlos en tus proyectos de C#.

3.1. Delegados

Los delegados son una característica importante de C# que permiten la encapsulación de métodos en objetos y su uso como argumentos y valores de retorno de otros métodos. Un delegado es esencialmente un tipo de referencia que puede apuntar a un método con una firma determinada. Se puede pensar en un delegado como un puntero a un método en otros lenguajes de programación.

Para crear un delegado, primero se debe definir un tipo de delegado que tenga la misma firma que el método al que se va a hacer referencia. Esto se hace mediante la definición de un delegado utilizando la palabra clave delegate y especificando los parámetros y el tipo de retorno del método al que se va a hacer referencia. A continuación se muestra un ejemplo:

```
delegate int OperacionMatematica(int a, int b);
```

En este ejemplo, se define un delegado llamado OperacionMatematica que tiene dos parámetros enteros y devuelve un valor entero. Una vez que se ha definido el tipo de delegado, se puede crear una instancia del delegado y asignarle una referencia a un método que tenga la misma firma que el delegado. A continuación se muestra un ejemplo:

```
int Suma(int a, int b) {
return a + b;
}

OperacionMatematica operacion = Suma;
```

En este ejemplo, se define un método Suma que toma dos parámetros enteros y devuelve la suma de estos parámetros. A continuación, se crea una instancia del delegado OperacionMatematica y se le asigna una referencia al método Suma mediante el operador de asignación =.

Los delegados también pueden utilizarse para añadir y eliminar métodos de un conjunto de métodos que se ejecutarán cuando se llame al delegado. Esto se hace mediante el operador += para añadir un método y -= para eliminar un método. A continuación se muestra un ejemplo:

```
OperacionMatematica operacion = Suma;
operacion += Resta;
operacion += Multiplicacion;
operacion -= Resta;
```

En este ejemplo, se crea una instancia del delegado OperacionMatematica y se le asigna una referencia al método Suma. A continuación, se añaden los métodos Resta y Multiplicacion al delegado mediante el operador +=. Finalmente, se elimina el método Resta del delegado mediante el operador -=.

Es importante tener en cuenta que los métodos añadidos a un delegado se ejecutan en el orden en que se añadieron. Por lo tanto, en el ejemplo anterior, se ejecutará primero el método Suma, seguido del método Multiplicacion.

Además, los delegados también pueden utilizarse como atributos en clases y estructuras, lo que permite crear eventos que pueden ser suscritos y desuscritos por otros métodos. Los eventos son un concepto importante en la programación de GUI y se utilizan para notificar a los objetos cuando ocurren ciertos eventos, como hacer clic en un botón.

Te dejo un ejemplo:

```
using System;

// Declaracion de un delegado
delegate void MiDelegado(int parametro);

class Program
{
    static void Main(string[] args)
    {
        // Creacion de una instancia del delegado y asignacion del metodo que queremos ejecutar
        MiDelegado miDelegado = new MiDelegado(MetodoEjemplo);

        // Invocacion del delegado
        miDelegado(5); // Salida: "El parametro recibido es: 5"

        // Agregar otro metodo al delegado
        miDelegado += MetodoAdicional;

        // Invocacion del delegado nuevamente
        miDelegado(10); // Salida: "El parametro recibido es: 10" y "Este es otro metodo."

        // Eliminar un metodo del delegado
        miDelegado -= MetodoEjemplo;

        // Invocacion del delegado una ultima vez
        miDelegado(15); // Salida: "Este es otro metodo."
    }
```

```
28
29      static void MetodoEjemplo(int parametro)
30      {
31          Console.WriteLine("El parametro recibido es: " + parametro);
32      }
33
34      static void MetodoAdicional(int parametro)
35      {
36          Console.WriteLine("Este es otro metodo.");
37      }
38  }
```

En este ejemplo, se define un delegado llamado MiDelegado que acepta un parámetro entero y no devuelve nada (void). Luego, se crea una instancia de este delegado y se le asigna el método MetodoEjemplo. Al invocar el delegado, se ejecuta el método asignado (MetodoEjemplo) con el parámetro indicado.

Posteriormente, se añade otro método al delegado (MetodoAdicional) y se invoca el delegado nuevamente, lo que hace que se ejecuten ambos métodos en orden. Luego, se elimina el método MetodoEjemplo del delegado y se invoca el delegado una última vez, lo que hace que solo se ejecute el método restante (MetodoAdicional).

3.2. Metodos anonimos

Los métodos anónimos son bloques de código que se pueden utilizar como parámetros de delegados. A diferencia de los métodos regulares, los métodos anónimos no tienen un nombre y se definen en línea. Estos métodos son útiles en situaciones donde se necesita pasar un pequeño bloque de código como parámetro a otro método o función.

Un ejemplo de esto es la siguiente definición de un delegado que acepta un método anónimo como parámetro:

```
1  delegate void MiDelegado(int x, int y);
2
3  class Program
4  {
5  static void Main(string[] args)
6      {
7      // Definir un delegado con un metodo anonimo como parametro
8      MiDelegado miDelegado = delegate (int x, int y) {
9          Console.WriteLine($"La suma de {x} y {y} es: {x + y}"); \\$
10         };
11     // Llamar al delegado
12     miDelegado(2, 3);
13     }
14 }
```

En este ejemplo, se define un delegado llamado `MiDelegado` que acepta dos parámetros enteros y no devuelve ningún valor. A continuación, se crea una instancia del delegado y se define un método anónimo dentro de él que toma los dos parámetros y los suma. Luego, se llama al delegado y se le pasan los valores 2 y 3 como argumentos.

Cabe destacar que los métodos anónimos también pueden ser utilizados en otros contextos, como en la definición de eventos o en la implementación de LINQ (Language Integrated Query).

Tipos dinamicos

Los tipos dinámicos son un tipo de datos que se introdujo en C# 4.0 que permiten omitir la comprobación de tipos estáticos en tiempo de compilación y posponerla hasta tiempo de ejecución. Esto significa que los objetos de tipos dinámicos pueden contener cualquier tipo de objeto, y se puede llamar a cualquier método o propiedad en tiempo de ejecución sin generar un error de compilación.

Para definir un objeto dinámico, se utiliza la palabra clave "dynamic". Por ejemplo:

```
dynamic miObjeto = "Hola mundo";
Console.WriteLine(miObjeto.GetType()); // Imprime "System.String"
miObjeto = 123;
Console.WriteLine(miObjeto.GetType()); // Imprime "System.Int32"
```

En el ejemplo anterior, se crea un objeto dinámico "miObjetoz se le asigna primero un valor de cadena y luego un valor entero. En tiempo de ejecución, se llama al método "GetType()"para imprimir el tipo de objeto que contiene en ese momento.

Es importante tener en cuenta que el uso excesivo de tipos dinámicos puede hacer que el código sea más difícil de entender y depurar. Por lo tanto, se recomienda utilizarlos con moderación y solo cuando sea necesario.

3.3. Enumerados

Un enumerado es un tipo de valor que tiene un conjunto de constantes con nombre. Cada constante enum tiene un valor asignado implícitamente y es del tipo del enumerado.

Un ejemplo de enumerado es el siguiente:

```
enum Direccion {
Norte,
Sur,
Este,
Oeste
}
```

En este ejemplo, definimos un tipo de enumerado llamado “Direccion” con cuatro constantes enumeradas: “Norte”, “Sur”, “Este” y “Oeste”.

Podemos usar un enumerado de la siguiente manera:

```
Direccion dir = Direccion.Norte;
if (dir == Direccion.Norte) {
Console.WriteLine("Voy hacia el norte");
}
```

También podemos iterar sobre todos los valores de un enumerado utilizando el método estático "GetValues" de la clase "Enum":

```
foreach (Direccion d in Enum.GetValues(typeof(Direccion))) {
Console.WriteLine(d);
}
```

Este código imprimirá lo siguiente:

```
Norte
Sur
Este
Oeste
```

3.4. Espacios de nombres

Los espacios de nombres en C# son una forma de agrupar y organizar clases, estructuras, interfaces, enumeraciones y otros tipos en grupos lógicos y coherentes. Un espacio de nombres puede ser considerado como un contenedor para un conjunto de tipos relacionados.

Para acceder a los miembros de un espacio de nombres en C#, podemos utilizar la palabra reservada üsing". Esto nos permite omitir la necesidad de especificar el espacio de nombres completo cada vez que queramos utilizar una clase dentro del mismo. Por ejemplo, si queremos utilizar la clase "StringBuilder"del espacio de nombres "System.Text", podemos incluir una directiva üsing.ªl principio de nuestro archivo:

```
using System.Text;

// Ahora podemos utilizar la clase StringBuilder sin especificar el espacio de nombres completo
StringBuilder sb = new StringBuilder();
```

También es posible utilizar un alias para un espacio de nombres si queremos hacer referencia a él de una manera más corta o clara. Por ejemplo, podemos crear un alias para el espacio de nombres "System.Text"de la siguiente manera:

```
using txt = System.Text;

// Ahora podemos utilizar el alias "txt" en lugar del espacio de nombres completo "System.Text"
txt.StringBuilder sb = new txt.StringBuilder();
```

Un buen uso de los espacios de nombres nos ayuda a organizar nuestro código y a evitar conflictos de nombres entre diferentes clases y librerías.

3.5. Tuplas

Las tuplas son un tipo de dato que permite agrupar múltiples elementos de diferentes tipos en un solo objeto. Las tuplas también pueden contener métodos y se pueden iterar usando un foreach. Sin embargo, es importante tener en cuenta que las tuplas tienen algunos límites. Por ejemplo, no se pueden agregar ni eliminar elementos de una tupla después de su creación, y las tuplas con más de siete elementos no se pueden utilizar con ciertas características de C# como el parámetro .ºut.en métodos.

En cuanto a si son más optimas que otros tipos de datos, depende del caso de uso específico. Las tuplas son útiles para agrupar elementos de diferentes tipos en una sola estructura, lo que puede mejorar la legibilidad del código. Pero en términos de rendimiento, puede haber casos en los que otros tipos de datos sean más eficientes.

Un ejemplo de uso de tuplas en C# es el siguiente:

```
// Crear una tupla
var miTupla = (1, "Hola", true);

// Acceder a los elementos de la tupla
Console.WriteLine(miTupla.Item1); // salida: 1
Console.WriteLine(miTupla.Item2); // salida: Hola
Console.WriteLine(miTupla.Item3); // salida: True

// Declarar tuplas con nombres de elementos
var otraTupla = (edad: 30, nombre: "Juan", ciudad: "Bogota");

// Acceder a los elementos de la tupla por nombre
Console.WriteLine(otraTupla.edad); // salida: 30
Console.WriteLine(otraTupla.nombre); // salida: Juan
Console.WriteLine(otraTupla.ciudad); // salida: Bogota

// Tuplas como parametros y retorno de metodo
(int, int) Calcular(int a, int b) {
int suma = a + b;
int producto = a * b;
return (suma, producto);
}

// Llamar al metodo y desempaquetar los valores de la tupla resultante
var resultado = Calcular(2, 3);
Console.WriteLine(resultado.Item1); // salida: 5
Console.WriteLine(resultado.Item2); // salida: 6
```

En este ejemplo, se muestra cómo crear y acceder a los elementos de tuplas, cómo declarar tuplas con nombres de elementos, cómo usar tuplas como parámetros y retorno de métodos, y cómo desempaquetar los valores de la tupla resultante.

3.6. Diccionarios

Un diccionario en C# es una colección de pares clave-valor. Cada clave debe ser única en la colección y se utiliza para buscar su valor correspondiente. Los valores pueden ser de cualquier tipo de datos, incluyendo tipos de referencia y tipos de valor. Los diccionarios son útiles para almacenar y recuperar rápidamente información en función de una clave, y son una herramienta poderosa para la manipulación de datos en C#.

- Los diccionarios son una colección de pares de clave-valor.
- Se pueden agregar, eliminar y modificar elementos en un diccionario.
- Los métodos comunes en los diccionarios incluyen `Add()`, `ContainsKey()`, `ContainsValue()`, `Remove()`, `Clear()`.
- Se puede iterar sobre un diccionario usando un `foreach` loop.
- Los diccionarios son útiles para almacenar y buscar datos por una clave específica.

Ejemplo de creación y uso de un diccionario en C#:

```
Dictionary<string, int> diccionario = new Dictionary<string, int>();
diccionario.Add("uno", 1);
diccionario.Add("dos", 2);
diccionario.Add("tres", 3);

foreach (KeyValuePair<string, int> kvp in diccionario)
{
Console.WriteLine("Clave: {0}, Valor: {1}", kvp.Key, kvp.Value);
}

if (diccionario.ContainsKey("dos"))
{
int valor = diccionario["dos"];
Console.WriteLine("El valor para la clave 'dos' es {0}", valor);
}

diccionario.Remove("tres");

Console.WriteLine("Numero de elementos en el diccionario: {0}", diccionario.Count);
```

En este ejemplo, se crea un diccionario que asocia cadenas con enteros. Se agregan tres elementos al diccionario usando el método `Add()`. Luego, se itera sobre el diccionario usando un `foreach` loop, mostrando la clave y el valor de cada elemento. Se verifica si el diccionario contiene una clave específica usando el método `ContainsKey()` y se accede al valor correspondiente usando la clave como índice. Se elimina un elemento

del diccionario usando el método `Remove()`. Finalmente, se muestra el número de elementos restantes en el diccionario usando la propiedad `Count`.

Boxing y unboxing

El boxing y el unboxing son dos conceptos importantes en C# que se utilizan para convertir tipos de datos de valor en objetos de referencia y viceversa.

Boxing se refiere al proceso de convertir un tipo de valor (como int, double, etc.) en un objeto de referencia (como object), mientras que unboxing se refiere al proceso inverso, es decir, convertir un objeto de referencia en su tipo de valor original.

Conceptos clave:

- Boxing: Conversión de un valor de tipo valor a un objeto de tipo objeto.
- Unboxing: Conversión de un objeto de tipo objeto a un valor de tipo valor.
- Overhead de Boxing y Unboxing.

Cuando se trabaja con tipos de valor en C#, a veces es necesario convertirlos en objetos de tipo object para trabajar con ellos en un contexto que requiera un objeto. A esto se le llama "boxing". Por ejemplo, si se quiere almacenar un valor de tipo int en una colección de objetos, es necesario convertir el valor int en un objeto. Esto se puede hacer con la palabra clave box.

Por otro lado, cuando se tiene un objeto de tipo object que se sabe que es de un tipo de valor, es necesario extraer el valor de ese objeto para trabajar con él como un valor de tipo valor. A esto se le llama ünboxing". Esto se puede hacer con la palabra clave unbox.

El proceso de boxing y unboxing puede tener un costo alto de rendimiento, por lo que se debe tener cuidado al utilizarlo en situaciones críticas de rendimiento.

A continuación se presenta un ejemplo de boxing y unboxing:

```
int x = 10;
object o = x; // boxing
int y = (int)o; // unboxing
```

Genericos

Los genéricos son un mecanismo que permite definir clases, estructuras, interfaces y métodos que puedan trabajar con diferentes tipos de datos sin especificarlos previamente. Se utilizan para crear componentes reutilizables que pueden funcionar con diferentes tipos de datos.

El parámetro de tipo es la variable de tipo que se define al crear una clase o un método genérico. Se utiliza para indicar qué tipo de datos se va a utilizar en el momento de la creación de la instancia de la clase o llamada al método.

La sintaxis para definir un parámetro de tipo es la siguiente:

```
class MiClase<T>
{
//...
}

void MiMetodo<T>(T parametro)
{
//...
}
```

En el ejemplo anterior, se define un parámetro de tipo T en la clase `MiClase` y en el método `MiMetodo`. Este parámetro puede ser cualquier tipo de dato, y se especifica en el momento de la creación de la instancia o llamada al método.

La utilización de genéricos permite escribir código más limpio y seguro, ya que se evitan conversiones innecesarias y se garantiza la compatibilidad de tipos en tiempo de compilación. Además, los genéricos son más eficientes en términos de rendimiento que la utilización de objetos de tipo `Object`, ya que no requieren boxing y unboxing.

3.7. Serialización JSON

La serialización es el proceso de convertir un objeto en una secuencia de bytes para almacenarlo o transmitirlo. En el contexto de la programación, la serialización se utiliza para compartir datos entre diferentes aplicaciones o para almacenar datos en un formato que se pueda leer y escribir fácilmente.

JSON (JavaScript Object Notation) es un formato de intercambio de datos ligero y fácil de leer y escribir. La serialización de objetos a JSON se utiliza comúnmente en la comunicación entre aplicaciones web y en el almacenamiento de datos en sistemas de bases de datos NoSQL.

En C#, la serialización de objetos a JSON se realiza utilizando la clase JsonSerializer que se encuentra en el espacio de nombres System.Text.Json. Esta clase proporciona una forma fácil de serializar y deserializar objetos a JSON.

La serialización de un objeto a JSON es tan sencillo como llamar al método Serialize de la clase JsonSerializer y pasarle el objeto a serializar como parámetro. El resultado de la serialización es una cadena JSON.

Aquí hay un ejemplo que serializa un objeto de tipo Persona a JSON:

```
using System;
using System.Text.Json;

class Persona {
public string Nombre { get; set; }
public int Edad { get; set; }
```

```
}

class Program {
static void Main(string[] args) {
Persona p = new Persona { Nombre = "Juan", Edad = 30 };
string json = JsonSerializer.Serialize(p);
Console.WriteLine(json);
}
}
```

La salida de este ejemplo será la siguiente:

```
{"Nombre":"Juan","Edad":30}
```

La deserialización de un objeto JSON a un objeto C# se realiza utilizando el método Deserialize de la clase JsonSerializer. El método toma dos parámetros: la cadena JSON a deserializar y el tipo de objeto C# al que se debe deserializar.

Aquí hay un ejemplo que deserializa un objeto JSON en un objeto de tipo Persona:

```
using System;
using System.Text.Json;

class Persona {
public string Nombre { get; set; }
public int Edad { get; set; }
}

class Program {
static void Main(string[] args) {
string json = "{"Nombre":"Juan","Edad":30}";
Persona p = JsonSerializer.Deserialize<Persona>(json);
Console.WriteLine(p.Nombre);
Console.WriteLine(p.Edad);
}
}
```

La salida de este ejemplo será la siguiente:

```
Juan
30
```

3.8. Eventos

Los eventos son un mecanismo que permite a un objeto comunicar a otros objetos cuando cierto evento ocurre. Los eventos son muy utilizados en el patrón de diseño .ºbserver.º .ºbservable"donde un objeto .ºbservable"notifica a los .ºbservadoresçuando ocurre un cambio. En C#, los eventos son definidos

como miembros de una clase, y están compuestos por un delegado y una lista de métodos suscritos. Los métodos suscritos son llamados cuando el evento es disparado.

La declaración de un evento es similar a la de un delegado, con la adición de la palabra clave event. Por ejemplo:

```
public class EventExample {
public delegate void MyEventHandler(object sender, EventArgs e);
public event MyEventHandler MyEvent;
public void DoSomething() {
    // Algo de codigo aqui
    // ...

    // Disparar el evento
    MyEvent?.Invoke(this, EventArgs.Empty);
}
}
```

En el ejemplo anterior, la clase EventExample tiene un evento MyEvent del tipo MyEventHandler. La clase también tiene un método DoSomething() que dispara el evento usando el operador ?.Invoke(). Los métodos suscritos al evento serán llamados cuando el evento sea disparado.

Para suscribir un método a un evento, se usa el operador +=. Por ejemplo:

```
public class EventSubscriber {
public void OnMyEvent(object sender, EventArgs e) {
Console.WriteLine("MyEvent was fired!");
}
}

var obj = new EventExample();
var sub = new EventSubscriber();
obj.MyEvent += sub.OnMyEvent;

// ...

obj.DoSomething(); // Imprime "MyEvent was fired!"
```

En el ejemplo anterior, se crea una instancia de EventSubscriber y se suscribe su método OnMyEvent al evento MyEvent de la instancia EventExample. Cuando DoSomething() es llamado, el evento es disparado y el método OnMyEvent() es llamado.

depuración

Para depurar un programa en Visual Studio, se pueden utilizar diferentes técnicas, una de las más comunes es el uso de puntos de interrupción. Un punto de interrupción es una instrucción que se inserta

en el código fuente del programa, que le indica al depurador que detenga la ejecución del programa en ese punto y permita inspeccionar el estado del programa en ese momento.

Otra técnica común es el uso de accesos del teclado en Visual Studio, que permiten activar y desactivar puntos de interrupción, avanzar paso a paso en la ejecución del programa, inspeccionar variables y expresiones, entre otras acciones.

En general, la depuración es un proceso importante en el desarrollo de software, que permite identificar y corregir errores en el código, lo que puede ahorrar tiempo en el largo plazo.

3.9. Tratamiento de excepciones

En C#, las excepciones son errores que ocurren durante la ejecución de un programa y que interrumpen el flujo normal de ejecución. Los errores que ocurren durante la ejecución del programa son llamados excepciones.

Existen diferentes tipos de excepciones predefinidas en C#, tales como la excepción de división por cero, excepción de índice fuera de rango, excepción de referencia nula, etc. Además, también se pueden crear excepciones personalizadas definidas por el usuario.

Para crear una excepción personalizada, se debe heredar de la clase Exception y crear un constructor que reciba una cadena de texto que describa el error. Por ejemplo:

```
class MyException : Exception
{
    public MyException(string message) : base(message)
    {}
}
```

Para lanzar una excepción, se utiliza la palabra clave throw seguida de una instancia de la excepción que se va a lanzar. Por ejemplo:

```
if (x == 0)
{
    throw new MyException("Error: x no puede ser cero");
}
```

Para capturar una excepción, se utiliza el bloque try-catch. El bloque try contiene el código que se va a ejecutar y que puede lanzar una excepción. El bloque catch captura la excepción y maneja el error. Por ejemplo:

```
try
{
    int x = 0;
    int y = 10 / x;
}
catch (DivideByZeroException ex)
```

```
{
    Console.WriteLine("Error: division por cero");
}
catch (MyException ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
    catch (Exception ex)
{
    Console.WriteLine("Error general: " + ex.Message);
}
```

En este ejemplo, si se intenta dividir por cero, se lanzará una excepción de tipo DivideByZeroException. Si se lanza una excepción de tipo MyException, se mostrará el mensaje de error que se pasó al constructor de la excepción. Si se lanza cualquier otra excepción, se mostrará un mensaje genérico de error.

Es importante capturar las excepciones de manera adecuada para evitar que el programa se bloquee y proporcionar una respuesta adecuada al usuario en caso de que ocurra un error.

3.10. Trabajar con archivos y Streams

El manejo de archivos y streams es una parte importante de la programación y en C# contamos con la librería System.IO para facilitar el acceso a los mismos.

En esta sección hablaremos de algunas herramientas y conceptos importantes para trabajar con archivos y streams en C#.

MemoryStream

MemoryStream es una clase que nos permite trabajar con datos en memoria. Podemos escribir y leer datos de un objeto MemoryStream como si estuviéramos trabajando con un archivo.

Para trabajar con un objeto MemoryStream primero debemos instanciarlo:

```
MemoryStream ms = new MemoryStream();
```

Una vez que tenemos nuestro objeto MemoryStream podemos escribir datos en él utilizando el método Write:

```
ms.Write(buffer, 0, buffer.Length);
```

Donde buffer es un arreglo de bytes que contiene los datos que queremos escribir.

Para leer datos de un MemoryStream podemos utilizar el método Read:

```
ms.Read(buffer, 0, buffer.Length);
```

Donde buffer es un arreglo de bytes que contendrá los datos leídos del MemoryStream.

También es posible posicionar el cursor en un MemoryStream utilizando el método Seek:

```
ms.Seek(0, SeekOrigin.Begin);
```

Este ejemplo posiciona el cursor al inicio del MemoryStream.

Archivos

Para trabajar con archivos podemos utilizar las clases File y FileInfo de la librería System.IO.

La clase File nos permite leer y escribir datos en archivos de forma sencilla:

```
File.WriteAllText("archivo.txt", "contenido del archivo");
string contenido = File.ReadAllText("archivo.txt");
```

El primer ejemplo escribe el contenido "contenido del archivo" en el archivo "archivo.txt". El segundo ejemplo lee el contenido del archivo "archivo.txt" y lo almacena en la variable contenido.

La clase FileInfo nos permite obtener información sobre un archivo específico:

```
FileInfo fileInfo = new FileInfo("archivo.txt");
long size = fileInfo.Length;
DateTime creationTime = fileInfo.CreationTime;
```

Este ejemplo obtiene el tamaño y la fecha de creación del archivo .ªrchivo.txt".

Streams

Un stream es una secuencia de bytes que puede ser leída o escrita. En C# existen varios tipos de streams, como FileStream, MemoryStream y NetworkStream, entre otros.

Para trabajar con streams podemos utilizar las clases Stream y StreamReader/StreamWriter de la librería System.IO.

La clase Stream es la clase base para todos los streams en C#. Podemos utilizarla para leer y escribir datos en un stream:

```
Stream stream = new MemoryStream();
byte[] buffer = new byte[1024];
int bytesRead = stream.Read(buffer, 0, buffer.Length);
stream.Write(buffer, 0, bytesRead);
```

Este ejemplo lee datos de un MemoryStream y los escribe en el mismo stream.

También podemos utilizar las clases StreamReader y StreamWriter para leer y escribir texto en un stream:

```
StreamWriter sw = new StreamWriter("archivo.txt");
sw.Write("contenido del archivo");
sw.Close();

StreamReader sr = new StreamReader("archivo.txt");
string contenido = sr.ReadToEnd();
sr.Close();
```

Aquí hay un ejemplo de cómo leer el contenido de un archivo usando la clase StreamReader:

```
try
{
    using (StreamReader sr = new StreamReader("archivo.txt"))
    {
        string linea;
        while ((linea = sr.ReadLine()) != null)
        {
            Console.WriteLine(linea);
        }
    }
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
```

En este ejemplo, se usa un bloque try-catch para manejar cualquier excepción que pueda ocurrir mientras se lee el archivo. Primero, se crea una instancia de la clase StreamReader pasando el nombre del archivo como argumento. Luego, se utiliza un bucle while para leer cada línea del archivo y escribirla en la consola. Finalmente, se cierra el StreamReader utilizando la declaración using, lo que asegura que se libere correctamente cualquier recurso utilizado por la clase.

Además de la clase StreamReader, la biblioteca System.IO proporciona una variedad de clases y métodos para trabajar con archivos y streams en C#. Estos incluyen la clase FileStream para leer y escribir datos en archivos, la clase BinaryReader y BinaryWriter para leer y escribir datos binarios, y la clase Directory para manipular directorios en el sistema de archivos.

3.11. Programación asíncrona

Para trabajar con programación asíncrona en C#, se utiliza la palabra clave "async" junto con "await". Esto permite que el código se ejecute en un hilo diferente al hilo principal, lo que puede mejorar el rendimiento de la aplicación. Además, esto permite realizar varias tareas al mismo tiempo y esperar a que todas se completen antes de continuar con la siguiente instrucción.

Por ejemplo, supongamos que tenemos un método "DescargarArchivoAsync" que descarga un archivo de Internet. Podríamos usar el siguiente código para llamar a este método de forma asíncrona:

```
public async void DescargarAsync()
{
    await DescargarArchivoAsync();
    Console.WriteLine("Archivo descargado exitosamente");
}
```

Aquí, usamos la palabra clave “async” en el método “DescargarAsync” y “await” en la llamada a “DescargarArchivoAsync”. Esto indica que estamos esperando a que el método “DescargarArchivoAsync” se complete antes de continuar con la siguiente instrucción, que en este caso es imprimir un mensaje en la consola.

Además, podemos iniciar varias tareas simultáneamente y esperar a que todas se completen antes de continuar con la siguiente instrucción. Por ejemplo, si tenemos dos métodos “DescargarArchivo1Async” y “DescargarArchivo2Async”, podemos usar el siguiente código para iniciar ambas tareas al mismo tiempo y esperar a que ambas se completen antes de continuar con la siguiente instrucción:

```
public async void DescargarArchivosAsync()
{
    Task tarea1 = DescargarArchivo1Async();
    Task tarea2 = DescargarArchivo2Async();

    await Task.WhenAll(tarea1, tarea2);

    Console.WriteLine("Archivos descargados exitosamente");
}
```

En este caso, usamos la clase “Task” para iniciar las tareas “DescargarArchivo1Async” y “DescargarArchivo2Async” al mismo tiempo. Luego, usamos el método “Task.WhenAll” para esperar a que ambas tareas se completen antes de continuar con la siguiente instrucción, que en este caso es imprimir un mensaje en la consola.

La programación asíncrona también permite la composición de tareas. Por ejemplo, si tenemos un método “ProcesarArchivoAsync” que procesa un archivo después de descargarlo, podemos usar el siguiente código para llamar a este método de forma asíncrona después de descargar el archivo:

```
public async void DescargarYProcesarAsync()
{
    await DescargarArchivoAsync();
    await ProcesarArchivoAsync();
    Console.WriteLine("Archivo procesado exitosamente");
}
```

Aquí, usamos la palabra clave “async” y “await” para llamar a los métodos “DescargarArchivoAsync” y “ProcesarArchivoAsync” de forma asíncrona. Esto nos permite descargar el archivo y procesarlo al mismo tiempo sin bloquear el hilo principal.

Finalmente, también podemos esperar a la finalización de una tarea antes de continuar con la siguiente instrucción. Por ejemplo, si tenemos un método “EsperarAsync” que espera 5 segundos antes de continuar, podemos usar el siguiente código para esperar a que el método se complete antes de continuar con la siguiente instrucción:

```
public async void EsperarYContinuarAsync()
```

```
{
    await EsperarAsync();
    Console.WriteLine("Espera completada");
}
```

En este caso, usamos la palabra clave “async” y “await” para llamar al método “Esperar”. La palabra clave “async” le dice al compilador que este método puede contener operaciones asincrónicas, mientras que “await” se utiliza para esperar la finalización de una tarea asincrónica antes de continuar con la ejecución del código.

El ejemplo anterior también muestra cómo iniciar tareas simultáneas con la clase Task. Podemos usar el método Task.WhenAll para esperar a que todas las tareas finalicen antes de continuar con la ejecución del código.

La programación asíncrona es especialmente útil cuando trabajamos con operaciones que pueden tardar mucho tiempo en completarse, como operaciones de entrada/salida o de red. En lugar de bloquear el subproceso principal mientras esperamos a que se complete una tarea, podemos permitir que el subproceso principal continúe ejecutando otras tareas.

Además, la programación asíncrona nos permite componer tareas de forma más eficiente. Podemos iniciar varias tareas simultáneamente y esperar a que todas finalicen antes de continuar con la ejecución del código.

4. LINQ

LINQ es una tecnología de Microsoft que permite realizar consultas sobre datos en diversos orígenes de datos, incluyendo bases de datos, colecciones en memoria y servicios web, utilizando una sintaxis unificada y orientada a objetos. LINQ significa "Language Integrated Query", y está integrado en el lenguaje de programación C# y otros lenguajes de .NET Framework. Con LINQ, se puede escribir consultas de una manera más fácil, eficiente y segura en comparación con otras técnicas de consulta de datos.

lambdas

En LINQ, las lambdas son funciones anónimas que se utilizan para representar una expresión o un bloque de código que se ejecutará en el contexto de una consulta LINQ.

Las lambdas se usan para definir las condiciones de filtrado, las operaciones de proyección y otras operaciones que se aplican a los datos en una consulta LINQ.

Por ejemplo, en la siguiente consulta LINQ, se utiliza una lambda para definir la condición de filtrado:

```
var productos = from p in listaDeProductos
                where p.Precio > 50
                select p;
```

En este caso, la lambda se utiliza para definir la condición de filtrado p.Precio >50. La variable p representa cada objeto en la lista de productos, y la expresión p.Precio >50 devuelve verdadero o falso para cada objeto según si su precio es mayor a 50 o no.

Las lambdas también se pueden utilizar para definir las operaciones de proyección, que son las que se aplican a los datos para transformarlos en otra forma. Por ejemplo:

```
var nombresDeProductos = listaDeProductos.Select(p => p.Nombre);
```

En este caso, la lambda p =>p.Nombre se utiliza para definir la operación de proyección que devuelve el nombre de cada producto en la lista. La variable p representa cada objeto en la lista de productos, y la expresión p.Nombre devuelve el nombre del producto.

En resumen, las lambdas son una herramienta fundamental en LINQ, ya que permiten definir de manera concisa y clara las operaciones de filtrado y proyección que se aplican a los datos en una consulta LINQ.

4.1. Sentencias from, join, let, where, order by, select, group by

En LINQ, las consultas se construyen utilizando una combinación de cláusulas. Estas cláusulas se pueden utilizar en cierto orden y se encadenan para construir una consulta más compleja. A continuación, se describen algunas de las cláusulas más comunes:

- from: especifica la fuente de los datos en la consulta. Por lo general, es una colección o una fuente de datos que implementa la interfaz IQueryable.
- join: se utiliza para combinar dos o más fuentes de datos en una única fuente. Es similar a la cláusula SQL "JOIN".
- let: permite declarar variables locales dentro de la consulta que se pueden utilizar en cláusulas posteriores.
- where: se utiliza para filtrar los datos en función de una condición especificada.order by: se utiliza para ordenar los resultados de la consulta en orden ascendente o descendente en función de una propiedad específica.
- select: se utiliza para seleccionar una propiedad o un conjunto de propiedades de los objetos en la fuente de datos.
- group by: se utiliza para agrupar los resultados de la consulta en función de una propiedad específica.

Cada una de estas cláusulas se puede encadenar para construir una consulta más compleja. Por ejemplo, una consulta LINQ puede tener el siguiente aspecto:

```
var query = from p in db.Personas
            join d in db.Direcciones on p.Id equals d.PersonaId
            where p.Edad > 18
            orderby p.Nombre ascending
            select new {
                Nombre = p.Nombre,
                Direccion = d.Direccion
            };
```

Esta consulta selecciona todas las personas mayores de 18 años de la tabla "Personas" y las combina con sus respectivas direcciones de la tabla "Direcciones". Luego, filtra los resultados para incluir solo aquellos cuyo nombre empieza por una letra específica y los ordena alfabéticamente por nombre. Finalmente, selecciona el nombre de la persona y su dirección, y devuelve los resultados como una lista de objetos anónimos.

Operadores

En LINQ, los operadores son métodos que se utilizan para realizar operaciones específicas en los objetos de consulta. Estos operadores se dividen en dos categorías: operadores de consulta y operadores de método.

Los operadores de consulta son palabras clave que se utilizan para definir una consulta LINQ, como from, where, select, orderby, etc. Estos operadores tienen una sintaxis similar a la de una cláusula de SQL.

Por otro lado, los operadores de método son métodos estáticos que se definen en la clase Enumerable y que se utilizan para ejecutar operaciones en los objetos de consulta. Algunos de los operadores de método más comunes son Where, Select, OrderBy, Join, etc.

Ambos tipos de operadores son muy útiles en LINQ para realizar consultas y operaciones complejas de manera fácil y eficiente.

Por ejemplo, el operador Where se utiliza para filtrar los elementos de una secuencia según un predicado determinado. El siguiente código muestra un ejemplo de cómo utilizar este operador para obtener todos los números pares de una secuencia de números:

```
int[] numeros = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var numerosPares = numeros.Where(n => n  % 2 == 0);

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

En este caso, la expresión lambda n =>n % 2 == 0 se utiliza como predicado para el operador Where. Esta expresión se evalúa para cada elemento de la secuencia numeros, devolviendo solo aquellos elementos que cumplen la condición de ser divisibles por 2, es decir, los números pares.

proveedores

Los proveedores de LINQ son las bibliotecas que implementan el motor de consulta LINQ para un origen de datos específico. Proporcionan el acceso a los datos, el procesamiento de consultas y la devolución de resultados a través de LINQ. Algunos de los proveedores más comunes son:

1. LINQ to Objects: permite realizar consultas en colecciones en memoria.
2. LINQ to SQL: proporciona acceso a datos en bases de datos relacionales utilizando SQL Server.
3. LINQ to Entities: proporciona acceso a datos en bases de datos relacionales a través del Entity Framework.
4. LINQ to XML: proporciona acceso a datos XML.

Cada proveedor tiene su propia implementación de los métodos de extensión de LINQ para admitir el procesamiento de consultas en su origen de datos específico.

4.2. Ejemplos

Aquí hay algunos ejemplos de consultas LINQ usando las diferentes cláusulas y operadores mencionados:

1. Ejemplo usando la cláusula from y where para filtrar una lista de estudiantes según su edad:

```
List<Student> students = GetStudents(); // Obtener lista de estudiantes

var filteredStudents = from s in students
                       where s.Age > 18
                       select s;
```

2. Ejemplo usando la cláusula join para unir dos listas de estudiantes y departamentos según su identificador:

```
List<Student> students = GetStudents(); // Obtener lista de estudiantes
List<Department> departments = GetDepartments(); // Obtener lista de departamentos

var joinedData = from s in students
                 join d in departments on s.DepartmentId equals d.Id
                 select new { StudentName = s.Name, DepartmentName = d.Name };
```

3. Ejemplo usando la cláusula let para crear una variable intermedia y utilizarla en una expresión posterior:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

var result = from n in numbers
             let squared = n * n
             where squared > 10
             select squared;
```

4. Ejemplo usando la cláusula order by para ordenar una lista de estudiantes por su nombre de forma ascendente:

```
List<Student> students = GetStudents(); // Obtener lista de estudiantes

var sortedStudents = from s in students
                     orderby s.Name ascending
                     select s;
```

5. Ejemplo usando la cláusula group by para agrupar una lista de estudiantes por su departamento:

```
List<Student> students = GetStudents(); // Obtener lista de estudiantes

var groupedStudents = from s in students
                      group s by s.DepartmentId into g
                      select new { DepartmentId = g.Key, Students = g.ToList() };
```

5. Arquitectura Hexagonal

La arquitectura hexagonal, también conocida como arquitectura de puertos y adaptadores, es un patrón de arquitectura de software que se enfoca en separar la lógica de negocio de la implementación técnica.

En la arquitectura hexagonal, el sistema se divide en tres capas principales: la capa de dominio, la capa de aplicación y la capa de infraestructura. La capa de dominio es donde se encuentra la lógica de negocio de la aplicación, mientras que la capa de aplicación es responsable de orquestar las diferentes partes de la aplicación para que funcionen juntas. La capa de infraestructura es donde se encuentran los detalles técnicos de la aplicación, como la base de datos, el sistema de archivos y las interfaces de usuario.

El objetivo principal de la arquitectura hexagonal es crear una aplicación que sea fácil de mantener y evolucionar con el tiempo. Al separar la lógica de negocio de la implementación técnica, se reduce la dependencia de la aplicación a tecnologías específicas y se facilita la integración con otros sistemas. Además, al tener una capa de dominio claramente definida, se facilita la creación de pruebas automatizadas y la verificación de que la aplicación funciona correctamente.

5.1. Participantes en una arquitectura hexagonal

1. Capa de Dominio:
   - Entidades: Representan los objetos del dominio, encapsulando tanto los datos como la lógica de negocio relacionada.
   - Interfaces de Repositorio: Definen los contratos para la persistencia y recuperación de las entidades.
   - Servicios de dominio: Implementan la lógica de negocio más compleja que no se puede asignar directamente a una entidad específica.
2. Capa de Aplicación:
   - Puertos de entrada: Son los puntos de entrada a la aplicación desde el exterior. Los puertos de entrada son interfaces que definen cómo se pueden recibir y procesar las solicitudes externas.
   - Puertos de salida: Son los puntos de salida de la aplicación hacia el exterior. Los puertos de salida son interfaces que definen cómo se pueden enviar respuestas o datos a sistemas externos.
   - Adaptadores de entrada: Son implementaciones concretas de los puertos de entrada. Los adaptadores de entrada se encargan de recibir las solicitudes externas y mapearlas a los comandos o consultas internas de la aplicación. Estos adaptadores también se encargan de la validación de entrada y la traducción de datos hacia el formato interno de la aplicación.

- Adaptadores de salida: Son implementaciones concretas de los puertos de salida. Los adaptadores de salida se encargan de tomar los resultados o datos generados por la aplicación y enviarlos a sistemas externos.

3. Capa de Infraestructura:

   - Implementaciones de Repositorio: Proporcionan la implementación concreta de los contratos de las interfaces de repositorio, realizando operaciones de persistencia y recuperación de datos.
   - Adaptadores: Se encargan de adaptar los componentes de la aplicación a las interfaces de la infraestructura externa, como bases de datos, sistemas de archivos, servicios web, etc.

4. Patrones y principios adicionales:

   - DTOs (Data Transfer Objects): Utilizados para transferir datos entre los componentes de la aplicación.
   - Modelos: Representan estructuras de datos utilizadas internamente en la aplicación, a menudo mapeadas desde las entidades o DTOs.
   - Inversión de Dependencias (Dependency Inversion Principle): Se enfoca en invertir las dependencias para permitir una arquitectura más flexible y modular.
   - Principio de Separación de Preocupaciones (Separation of Concerns): Se busca separar las diferentes responsabilidades y preocupaciones en componentes distintos.

En una arquitectura hexagonal bien diseñada, los DTOs se utilizan para transferir datos entre los diferentes componentes de la aplicación, los modelos se utilizan internamente para representar estructuras de datos y las entidades se utilizan para modelar los objetos y la lógica de negocio del dominio. Los controladores en la capa de aplicación coordinan las interacciones entre los diferentes componentes, mientras que los adaptadores de infraestructura se encargan de adaptar la aplicación a las interfaces de la infraestructura externa.

Estos componentes y principios trabajan en conjunto para lograr una arquitectura flexible, mantenible y escalable, donde la lógica de negocio se mantiene independiente de las tecnologías y detalles de implementación, y donde los diferentes componentes son intercambiables y se pueden probar de forma aislada.

5.2. Flujo en una arquitectura hexagonal

En la arquitectura hexagonal, el flujo de datos sigue un patrón conocido como .ªrquitectura de puertos y adaptadores". A continuación, se describe el flujo de datos típico en esta arquitectura:

1. Entrada de datos: Los datos ingresan a la aplicación a través de los puertos de entrada. Estos puertos pueden ser interfaces de usuario, servicios web, mensajes, archivos, entre otros. Los datos de entrada

son recibidos por adaptadores específicos que se encargan de transformarlos en un formato adecuado para ser procesados por la aplicación.

2. Capa de aplicación: Los datos procesados en los adaptadores son enviados a la capa de aplicación. Aquí se encuentra la lógica de negocio de la aplicación. Los adaptadores hacen uso de los servicios y objetos definidos en esta capa para realizar las operaciones necesarias.

3. Capa de dominio: La capa de dominio es el núcleo de la aplicación y contiene la lógica de negocio principal. Aquí se encuentran las entidades, los repositorios y los servicios relacionados con el dominio específico de la aplicación. Los objetos en esta capa pueden realizar operaciones en los datos recibidos y aplicar reglas de negocio.

4. Capa de infraestructura: Una vez que la capa de dominio realiza las operaciones necesarias, los resultados son enviados de vuelta a los adaptadores a través de los puertos de salida. Los adaptadores se encargan de transformar los datos en un formato adecuado para su posterior envío o almacenamiento. Estos puertos de salida pueden ser interfaces de usuario, servicios web, bases de datos, sistemas de archivos, entre otros.

El flujo de datos en la arquitectura hexagonal sigue un patrón en el cual los datos ingresan a través de los puertos de entrada, son procesados por la capa de aplicación y la capa de dominio, y finalmente se envían de vuelta a través de los puertos de salida. Esta arquitectura promueve la separación de la lógica de negocio de los detalles de implementación técnica, lo que facilita la modularidad, la prueba y la evolución de la aplicación.

Aquí tienes un ejemplo sencillo de flujo de datos en la arquitectura hexagonal para un sistema de gestión de usuarios:

1. Entrada de datos:
   - Puerto de entrada: Interfaz de usuario (por ejemplo, una página web).
   - Adaptador: Controlador de la interfaz de usuario.
2. Capa de aplicación:
   - Objeto de la capa de aplicación: Servicio de gestión de usuarios.
   - Operación: Crear un nuevo usuario.
3. Capa de dominio:
   - Entidad: Usuario.
   - Repositorio: Repositorio de usuarios.
   - Regla de negocio: Validar que el usuario tenga un nombre único.

4. Capa de infraestructura:
    - Puerto de salida: Base de datos.
    - Adaptador: Adaptador de base de datos.

    El flujo de datos sería el siguiente:

1. Un usuario ingresa los datos (nombre, correo electrónico, contraseña, etc.) en el formulario de registro de la interfaz de usuario.
2. El controlador de la interfaz de usuario recibe los datos y los pasa al servicio de gestión de usuarios de la capa de aplicación.
3. El servicio de gestión de usuarios llama al repositorio de usuarios de la capa de dominio para crear un nuevo usuario.
4. Antes de crear el usuario, la capa de dominio verifica si el nombre del usuario ya existe en el repositorio de usuarios.
5. Si el nombre del usuario no está duplicado, se crea un objeto de entidad de usuario y se guarda en el repositorio de usuarios.
6. El adaptador de base de datos toma el objeto de entidad de usuario y lo guarda en la base de datos.

En este ejemplo, los datos fluyen desde la interfaz de usuario hasta la capa de aplicación, luego a la capa de dominio y finalmente se almacenan en la base de datos a través de los adaptadores correspondientes. La arquitectura hexagonal permite que la lógica de negocio se mantenga independiente de los detalles de implementación técnica, lo que facilita la prueba, la reutilización y la evolución del sistema.

Un puerto de entrada y un puerto de salida en la arquitectura hexagonal son interfaces que definen cómo se comunican los componentes externos con la aplicación y cómo la aplicación se comunica con los componentes externos, respectivamente. A nivel de diseño, un puerto de entrada y un puerto de salida se representan generalmente como interfaces en el lenguaje de programación utilizado.

Por ejemplo

Port de entrada

```
public interface IUserInputPort
{
    void CreateUser(string name, string email, string password);
}
```

Port de salida

```
public interface IUserOutputPort
{
    void UserCreated(User user);
}
```

En este ejemplo, IUserInputPort representa un puerto de entrada y IUserOutputPort representa un puerto de salida.

El puerto de entrada (IUserInputPort) define un método CreateUser que toma como parámetros el nombre, el correo electrónico y la contraseña de un usuario. Este puerto es utilizado por los adaptadores de entrada, como un controlador de interfaz de usuario o un servicio web, para enviar datos a la aplicación.

El puerto de salida (IUserOutputPort) define un método UserCreated que recibe un objeto de tipo User como parámetro. Este puerto es utilizado por los adaptadores de salida, como un adaptador de base de datos o un servicio web, para recibir datos de la aplicación y realizar las acciones necesarias, como guardar los datos en una base de datos o enviar una respuesta al cliente.

Los adaptadores son componentes responsables de la comunicación entre los puertos de entrada/-salida y la lógica de la aplicación. Los adaptadores actúan como intermediarios para transformar los datos recibidos desde los puertos de entrada en un formato adecuado para su procesamiento en la aplicación y viceversa, convirtiendo los datos de salida de la aplicación en un formato adecuado para ser enviado a través de los puertos de salida.

Existen dos tipos de adaptadores comunes en la arquitectura hexagonal: adaptadores de entrada y adaptadores de salida.

1. Adaptadores de entrada: Los adaptadores de entrada son responsables de recibir los datos provenientes de los puertos de entrada y convertirlos en una forma que la aplicación pueda entender. Algunos ejemplos de adaptadores de entrada pueden incluir:
    - Controladores de interfaz de usuario: Estos adaptadores reciben las solicitudes del usuario a través de la interfaz de usuario, extraen los datos necesarios y los pasan a la capa de aplicación para su procesamiento.
    - Adaptadores de servicios web: Estos adaptadores reciben las solicitudes HTTP de los clientes, interpretan los datos de la solicitud y los convierten en una forma que la aplicación pueda manejar.
2. Adaptadores de salida: Los adaptadores de salida son responsables de recibir los resultados de la aplicación y convertirlos en un formato adecuado para enviarlos a través de los puertos de salida. Algunos ejemplos de adaptadores de salida pueden incluir:
    - Adaptadores de bases de datos: Estos adaptadores se encargan de interactuar con una base de datos específica y realizar operaciones de lectura o escritura de datos.
    - Adaptadores de servicios web: Estos adaptadores toman los resultados de la aplicación y los envían a través de servicios web para ser consumidos por otros sistemas.

- Adaptadores de mensajes: Estos adaptadores se utilizan para enviar mensajes a través de sistemas de mensajería o colas, permitiendo la integración con otros componentes o sistemas.

Cada adaptador implementa los métodos definidos en los puertos de entrada/salida correspondientes, transformando los datos de acuerdo con las necesidades de la aplicación y los sistemas externos con los que interactúa. Los adaptadores juegan un papel fundamental en la arquitectura hexagonal, ya que permiten la separación de la lógica de la aplicación de los detalles de comunicación y tecnología específicos.

Por ejemplo

Adaptador de entrada

```
public class UserInputAdapter : IUserInputPort
{
    private readonly IUserOutputPort _outputPort;

    public UserInputAdapter(IUserOutputPort outputPort)
    {
        _outputPort = outputPort;
    }

    public void CreateUser(string name, string email, string password)
    {
        // Logic para adaptar los datos recibidos a la forma requerida por la app
        User user = new User(name, email, password);

        // Llamada al puerto de salida para indicar que se ha creado el usuario
        _outputPort.UserCreated(user);
    }
}
```

Adaptador de salida

```
public class UserOutputAdapter : IUserOutputPort
{
        public void UserCreated(User user)
        {
            // Logic para adaptar los datos de salida y enviarlos a traves del puerto de salida
            Console.WriteLine("El usuario" + user.Name + "  ha sido creado exitosamente.");
        }
    }
```

En el ejemplo anterior, UserInputAdapter es un adaptador de entrada que implementa el puerto de entrada IUserInputPort. Recibe una instancia del adaptador de salida IUserOutputPort a través de su constructor, que se utiliza para llamar al método UserCreated del puerto de salida.

UserInputAdapter implementa el método CreateUser, que recibe los datos de entrada (nombre, correo electrónico, contraseña) y los adapta en un objeto User para su procesamiento en la aplicación.

Luego, utiliza el puerto de salida para indicar que se ha creado un nuevo usuario, pasando el objeto User como parámetro.

UserOutputAdapter es un adaptador de salida que implementa el puerto de salida IUserOutputPort. En este ejemplo, simplemente muestra un mensaje en la consola indicando que el usuario ha sido creado exitosamente. En una implementación real, este adaptador podría realizar acciones adicionales, como guardar los datos en una base de datos o enviar una respuesta a través de un servicio web.

Estos adaptadores son ejemplos básicos, pero ilustran cómo se implementan y cómo se adaptan los datos entre los puertos de entrada y salida en la arquitectura hexagonal.

5.3. DTO (Data Transfer Object)

Un DTO (Data Transfer Object) es un patrón de diseño utilizado para transferir datos entre diferentes componentes de una aplicación. Los DTOs son objetos simples que contienen campos de datos y no contienen lógica de negocio adicional. Su propósito principal es facilitar la transferencia eficiente de datos entre capas de una aplicación o entre diferentes sistemas.

Aquí hay algunos puntos clave sobre los DTOs:

1. Transferencia de datos: Los DTOs se utilizan para transferir datos entre componentes de la aplicación, como capas de la arquitectura o sistemas externos. Sirven como estructuras de datos que encapsulan los datos necesarios para una operación específica.

2. Estructura de datos simples: Los DTOs generalmente son clases simples que contienen propiedades o campos para almacenar los datos relevantes. No contienen métodos con lógica de negocio compleja, ya que su propósito es simplemente contener y transportar datos.

3. Independientes de la interfaz de usuarioación: Los DTOs son independientes de la capa de presentación y no deben contener referencias a frameworks o tecnologías específicas. De esta manera, se pueden utilizar en diferentes capas o en diferentes interfaces de usuario sin problemas de dependencia.

4. Adaptación de datos: Los DTOs se adaptan para ajustarse a los requisitos de cada componente. Por ejemplo, en una capa de servicio web, los DTOs pueden ser serializados en formato JSON o XML para ser enviados a través de una solicitud HTTP. En una capa de acceso a datos, los DTOs pueden ser mapeados a las entidades del modelo de datos.

5. Eficiencia en la transferencia de datos: Los DTOs permiten reducir el tráfico de datos y mejorar el rendimiento de la aplicación. Al enviar solo los datos necesarios en lugar de objetos completos con lógica de negocio, se reduce el tamaño de los mensajes y se mejora la eficiencia en la transferencia de datos.

6. Separación de preocupaciones: Los DTOs ayudan a mantener una separación clara entre la capa de presentación y la capa de dominio. Al utilizar objetos DTO para transferir datos, se evita la exposición directa de las entidades de dominio a la capa de presentación, lo que facilita el mantenimiento y la evolución de la aplicación.

Los DTOs son objetos simples utilizados para transferir datos entre componentes de una aplicación. Proporcionan una estructura de datos eficiente y desacoplada que facilita la comunicación entre diferentes capas o sistemas externos, mejorando la modularidad y el rendimiento de la aplicación.

5.4. Entidades

En el contexto de la arquitectura de software, las entidades son objetos que representan conceptos del dominio de la aplicación. Las entidades encapsulan datos y comportamiento relacionados a un concepto específico y son fundamentales para modelar y manipular la información dentro de la aplicación.

Aquí tienes algunas características y puntos clave sobre las entidades:

1. Representación de conceptos del dominio: Las entidades representan objetos del mundo real o abstracto que son relevantes para el dominio de la aplicación. Por ejemplo, en una aplicación de gestión de usuarios, podría haber una entidad "Usuario" que contiene los atributos como nombre, dirección de correo electrónico, contraseña, etc.

2. Encapsulación de datos: Las entidades encapsulan los datos relacionados a un concepto específico. Estos datos se almacenan en propiedades o atributos de la entidad y pueden ser de diferentes tipos, como cadenas de texto, números, fechas, listas u otros objetos relacionados.

3. Lógica de negocio: Las entidades pueden contener lógica de negocio relacionada a su comportamiento y reglas de validación. Por ejemplo, la entidad "Usuario" podría tener métodos para verificar la validez de una contraseña o para calcular información basada en sus atributos.

4. Identidad única: Cada entidad generalmente tiene una identidad única que la distingue de otras entidades del mismo tipo. Esta identidad puede ser representada por un identificador único, como un número o una cadena de caracteres.

5. Persistencia y almacenamiento: Las entidades pueden ser utilizadas para representar datos que se almacenan en una base de datos o en algún otro medio de almacenamiento. En estos casos, se utilizan técnicas de mapeo objeto-relacional (ORM) u otras técnicas para persistir y recuperar las entidades en el almacenamiento.

6. Relaciones entre entidades: Las entidades pueden tener relaciones entre sí, formando una estructura de datos más compleja. Por ejemplo, en un sistema de gestión de pedidos, podría haber una entidad "Pedido" que tiene una relación con una entidad "Cliente".

En la arquitectura hexagonal, las entidades suelen pertenecer a la capa de dominio, representando conceptos fundamentales del negocio. Estas entidades son independientes de las capas externas y no contienen dependencias de infraestructura o detalles de implementación.

Las entidades desempeñan un papel central en la arquitectura, ya que representan la información clave y la lógica de negocio de la aplicación. Al mantener una separación clara entre las entidades y las capas externas, se logra un diseño más modular, flexible y mantenible.

5.5. Inyección de dependencias

La inyección de dependencias es un patrón de diseño utilizado en la programación orientada a objetos que permite gestionar las dependencias entre los componentes de una aplicación. En lugar de que un componente cree directamente sus dependencias, estas se inyectan desde el exterior, lo que proporciona una mayor flexibilidad, reutilización y facilita la prueba unitaria.

En la inyección de dependencias, se establece una separación clara entre la creación de un objeto y su uso, permitiendo que el objeto dependiente (llamado cliente) reciba las dependencias necesarias a través de algún mecanismo de inyección. Esto puede lograrse de varias formas:

1. Inyección de dependencias constructor: Las dependencias se pasan como argumentos al constructor del objeto cliente durante su creación. Este enfoque garantiza que las dependencias requeridas estén disponibles desde el principio y promueve la creación de objetos inmutables.

2. Inyección de dependencias por métodos/setters: Las dependencias se establecen a través de métodos o setters en el objeto cliente después de su creación. Esto permite una mayor flexibilidad en la configuración de las dependencias y permite la actualización de las mismas en tiempo de ejecución.

3. Inyección de dependencias por contexto o contenedor: Se utiliza un contenedor de inversión de control (IoC) o un contexto que se encarga de gestionar la creación y la resolución de las dependencias. El cliente simplemente especifica las dependencias requeridas y el contenedor se encarga de inyectarlas automáticamente.

Beneficios de la inyección de dependencias:

- Mayor flexibilidad: Permite cambiar las implementaciones de las dependencias sin modificar el código del cliente, lo que facilita la adopción de diferentes configuraciones y la implementación de cambios sin impactar en otros componentes.

- Reutilización: Las dependencias se pueden compartir y reutilizar en diferentes componentes de la aplicación, evitando la duplicación de código y promoviendo una arquitectura más modular.

- Facilita la prueba unitaria: Al inyectar las dependencias, se pueden utilizar objetos simulados o de prueba para aislar el componente bajo prueba, lo que facilita la escritura de pruebas unitarias más robustas y reduce las dependencias externas.
- Desacoplamiento: La inyección de dependencias reduce el acoplamiento entre los componentes, ya que los objetos dependientes no están fuertemente acoplados a las implementaciones concretas de sus dependencias.

Por ejemplo. Supongamos que tienes una clase Cliente que necesita utilizar una implementación de la interfaz IServicioEmail para enviar correos electrónicos. En lugar de que la clase Cliente cree directamente una instancia de la implementación concreta de IServicioEmail, se utilizará la inyección de dependencias para recibirla desde el exterior.

```
    public interface IServicioEmail
{
    void EnviarCorreo(string destinatario, string asunto, string contenido);
}

public class ServicioEmail : IServicioEmail
{
    public void EnviarCorreo(string destinatario, string asunto, string contenido)
    {
        // Implementacion para enviar el correo
        Console.WriteLine("Enviando correo a " + destinatario + ": " + asunto " - " contenido);
    }
}

public class Cliente
{
    private readonly IServicioEmail _servicioEmail;

    public Cliente(IServicioEmail servicioEmail)
    {
        _servicioEmail = servicioEmail;
    }

    public void MetodoEjemplo()
    {
        // Utiliza _servicioEmail para enviar un email
        _servicioEmail.EnviarEmail("destinatario@example.com", "Hola, esto es un ejemplo de email");
    }
}
```

En este ejemplo, la clase Cliente tiene una dependencia en la interfaz IServicioEmail, que representa la capacidad de enviar correos electrónicos. En el constructor de la clase Cliente, se espera una instancia

de IServicioEmail para ser inyectada. Esto permite que el cliente sea independiente de la implementación concreta de IServicioEmail.

```
services.AddTransient<IServicioEmail, ServicioEmail>();
```

En cualquier lugar donde necesites utilizar IServicioEmail, puedes hacerlo a través de la inyección de dependencia.

Con este enfoque, puedes cambiar fácilmente la implementación de IServicioEmail sin modificar la clase Cliente. Por ejemplo, podrías tener una implementación de IServicioEmail específica para pruebas unitarias que no envíe correos electrónicos reales, sino que los simule.

Recuerda que el alcance transitorio (AddTransient) crea una nueva instancia cada vez que se resuelve, lo cual puede ser adecuado para servicios livianos y sin estado. Si necesitas un comportamiento diferente, como un alcance de instancia única o un alcance por solicitud, puedes utilizar AddSingleton o AddScoped respectivamente en lugar de AddTransient en la configuración de inyección de dependencia.

Tipos de inyección de dependencias

Las siguientes son configuraciones de inyección de dependencia en ASP.NET Core:

1. **Singleton**: Con AddSingleton, se registra una instancia única de una clase en el contenedor de inyección de dependencia. Esto significa que se crea una única instancia de la clase y se reutiliza cada vez que se resuelve esa dependencia. Es adecuado para objetos que deben compartirse en toda la aplicación. Aquí tienes un ejemplo:

```
services.AddSingleton<IServicioEmail, ServicioEmail>();
```

2. **Scoped**: Con AddScoped, se registra una instancia por solicitud en el contenedor de inyección de dependencia. Esto significa que se crea una nueva instancia de la clase para cada solicitud HTTP entrante y se reutiliza en el ámbito de esa solicitud. Es útil cuando deseas que una instancia se mantenga durante la duración de una solicitud, pero no necesariamente en toda la aplicación. Aquí tienes un ejemplo:

```
services.AddScoped<IServicioEmail, ServicioEmail>();
```

3. **Transitorio**: Con AddTransient, se registra una nueva instancia de la clase en cada solicitud de resolución en el contenedor de inyección de dependencia. Esto significa que se crea una nueva instancia cada vez que se resuelve la dependencia. Es útil cuando deseas una nueva instancia de la clase cada vez que se solicita. Aquí tienes un ejemplo:

```
services.AddTransient<IServicioEmail, ServicioEmail>();
```

6. Unit Test

Las pruebas unitarias son una práctica esencial en el desarrollo de software que consiste en probar de manera automatizada las unidades más pequeñas de código, como métodos y clases, de forma aislada. El objetivo principal de las pruebas unitarias es verificar que cada unidad de código funcione correctamente de manera individual, asegurando que cumplan con los requisitos y evitando la introducción de errores.

Al aplicar pruebas unitarias, se obtienen diversos beneficios, entre ellos:

1. **Detección temprana de errores:** Las pruebas unitarias permiten identificar problemas y defectos en el código de manera temprana, lo que facilita su corrección antes de que se propaguen a otras partes del sistema.

2. **Mantenimiento y refactorización:** Las pruebas unitarias brindan confianza al realizar cambios en el código, ya que proporcionan una forma de verificar rápidamente que las funcionalidades previamente implementadas sigan siendo válidas después de los cambios.

3. **Documentación de uso:** Las pruebas unitarias sirven como una documentación ejecutable de cómo se debe utilizar una unidad de código. Al leer las pruebas, los desarrolladores pueden comprender rápidamente cómo utilizar y cómo debería comportarse el código.

4. **Facilita el trabajo en equipo:** Las pruebas unitarias promueven la colaboración entre desarrolladores al proporcionar un conjunto de casos de prueba compartidos que definen el comportamiento esperado de las unidades de código.

Veamos un ejemplo. Supongamos que tenemos una clase llamada MathUtils que contiene varios métodos estáticos para realizar operaciones matemáticas simples. Vamos a escribir una prueba unitaria utilizando NUnit para el método Multiply, que calcula la multiplicación de dos números enteros.

Aquí está la implementación de la clase MathUtils:

```
1  public class MathUtils
2  {
3      public static int Add(int a, int b)
4      {
5          return a + b;
6      }
7
8      public static int Multiply(int a, int b)
9      {
10         return a * b;
11     }
12
13     public static int Divide(int dividend, int divisor)
14     {
```

```
        if (divisor == 0)
        {
            throw new DivideByZeroException("Cannot divide by zero.");
        }

        return dividend / divisor;
    }
}
```

Y ahora vamos a escribir la prueba unitaria utilizando NUnit:

```
using NUnit.Framework;

[TestFixture]
public class MathUtilsTests
{
    [Test]
    public void Multiply_WhenCalledWithTwoNumbers_ReturnsProduct()
    {
        // Arrange
        int number1 = 5;
        int number2 = 10;

        // Act
        int result = MathUtils.Multiply(number1, number2);

        // Assert
        Assert.AreEqual(50, result);
    }

    [Test]
    public void Multiply_WhenCalledWithZero_ReturnsZero()
    {
        // Arrange
        int number1 = 5;
        int number2 = 0;

        // Act
        int result = MathUtils.Multiply(number1, number2);

        // Assert
        Assert.AreEqual(0, result);
    }
}
```

En este ejemplo, hemos escrito dos pruebas unitarias para el método Multiply de la clase MathUtils.

La primera prueba verifica que el método devuelva el producto correcto cuando se le pasan dos números enteros. La segunda prueba verifica que el método devuelva cero cuando uno de los números es cero.

Al ejecutar estas pruebas utilizando NUnit, se verificará si los resultados obtenidos coinciden con los valores esperados. Si todos los resultados son correctos, las pruebas pasarán sin errores. De lo contrario, si alguna aserción falla, se generará una excepción y la prueba se considerará fallida.

6.1. Patron AAA

El patrón AAA (Arrange-Act-Assert) es una convención comúnmente utilizada en las pruebas unitarias para estructurar las pruebas de manera clara y legible. Este patrón divide la prueba en tres partes distintas: Arrange, Act y Assert.

- **Arrange (Preparación):** En esta etapa, se prepara el entorno necesario para la prueba. Se configuran las condiciones iniciales, se crean objetos y se establecen las dependencias necesarias. Aquí se definen los datos de entrada y se configuran los valores esperados.
- **Act (Acción):** En esta etapa, se realiza la acción o el comportamiento que se quiere probar. Se invoca el método o se lleva a cabo la operación que se desea evaluar. Aquí se obtiene el resultado de la acción realizada.
- **Assert (Verificación):** En esta etapa, se verifica si el resultado obtenido de la acción realizada es el esperado. Se utilizan aserciones (assertions) para comparar el resultado actual con el valor esperado. Si la aserción es verdadera, la prueba pasa; de lo contrario, la prueba falla y se muestra un mensaje de error.

El uso del patrón AAA facilita la comprensión de la intención de la prueba y mejora su mantenibilidad, ya que separa claramente las diferentes etapas de la misma. A continuación, se muestra un ejemplo de cómo se aplicaría el patrón AAA en una prueba unitaria utilizando el framework NUnit:

```
1  [TestClass]
2  public class MyMathTests
3  {
4      [TestMethod]
5      public void Add_WhenCalledWithTwoNumbers_ReturnsSum()
6      {
7          // Arrange
8          int number1 = 5;
9          int number2 = 10;
10         int expectedSum = 15;
11
12         MyMath myMath = new MyMath();
13
```

```
        // Act
        int result = myMath.Add(number1, number2);

        // Assert
        Assert.AreEqual(expectedSum, result);
    }
}
```

6.2. TDD (Desarrollo Dirigido por Pruebas)

El Desarrollo Dirigido por Pruebas (Test-Driven Development o TDD) es una metodología de desarrollo de software que se basa en escribir las pruebas unitarias antes de implementar el código de producción. Es un enfoque iterativo e incremental en el cual el ciclo de desarrollo consiste en tres pasos: Escribir una prueba, hacerla pasar y refactorizar.

El proceso de TDD sigue los siguientes pasos, iterando al agregar una nueva funcionalidad:

1. **Escribir una prueba (Write a test):** En esta etapa, se escribe una prueba unitaria que defina el comportamiento deseado del código a implementar. La prueba inicialmente fallará, ya que el código aún no existe. Este paso es llamado el paso rojo (red).

2. **Hacer pasar la prueba (Make the test pass):** En esta etapa, se implementa la funcionalidad mínima necesaria en el código para que la prueba pase. No se busca una implementación completa en esta etapa, solo lo suficiente para satisfacer la prueba. Este paso es llamado el paso verde (green).

3. **Refactorizar (Refactor):** Una vez que la prueba pasa, se realiza la refactorización del código para mejorar su calidad, eliminar duplicaciones y aplicar buenas prácticas de programación. Se asegura de que las pruebas sigan pasando después de cada refactorización.

El TDD se centra en el desarrollo incremental y en la retroalimentación constante proporcionada por las pruebas unitarias. Al seguir esta metodología, se obtienen diversos beneficios, como un código más limpio y modular, mayor confianza en la calidad del código y una cobertura de pruebas más completa. También promueve un enfoque más centrado en el diseño y la arquitectura del código, ya que las pruebas ayudan a definir las interfaces y los requisitos antes de la implementación.

Por ejemplo: Supongamos que queremos desarrollar una clase estatica que tenga algunos metodos de la teoría de números, “Residuo modular” y “factorial”.

Empezamos escribiendo las pruebas unitarias para estos metodos:

```
using Pruebas.metodos;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TeoriaDeNumerosTests
```

```
{
    [TestClass]
    public class TeoriaDeNumerosTest
    {
        [TestMethod]
        public void TestResuduo()
        {
            int divisor = 2;
            int dividendo = 5;

            int resultado = TeoriaDeNumeros.Resuduo(dividendo, divisor);

            Assert.AreEqual(1, resultado);
        }
        [TestMethod]
        public void TestFactorial()
        {
            int numero = 5;

            int resultado = TeoriaDeNumeros.Factorial(numero);

            Assert.AreEqual(120, resultado);
        }
    }
}
```

Procedemos a escribir el codigo necesario para que estas pruebas pasen:

```
namespace Pruebas.metodos
{
    public static class TeoriaDeNumeros
    {
        public static int Resuduo(int dividendo, int divisor)
        {
            int cociente = 0;
            int residuo = new int();
            do{
                residuo = dividendo -  divisor * cociente;
                cociente++;
            }
            while (residuo >= 2) ;

            return residuo;
        }
        public static int Factorial(int numero)
        {
```

```
            if (numero == 0) return 1;
            else return numero * Factorial(numero - 1);
        }
    }
}
```

Y luego lo refactorizamos, de ser necesario.

6.3. Moq

Moq es un popular framework de moqueo para el lenguaje de programación C#. Permite crear objetos simulados y configurar su comportamiento durante las pruebas unitarias. A continuación, se presentan algunos conceptos y características clave de Moq:

- **Instalación**: Moq se puede instalar a través del administrador de paquetes NuGet en Visual Studio. El paquete se llama "Moq" y está disponible en la galería de paquetes de NuGet.
- **Creación de objetos simulados**: Moq permite crear objetos simulados utilizando el método estático `Mock.Of<T>()`, donde T representa el tipo de la interfaz o clase que se desea simular. También se puede utilizar el constructor de la clase Mock<T>() para crear el objeto simulado.
- **Configuración de comportamiento**: Una vez creado el objeto simulado, se puede configurar su comportamiento utilizando el método `Setup()` de Moq. Por ejemplo, se puede especificar el valor de retorno de un método simulado, lanzar una excepción, establecer el comportamiento de propiedades, etc.
- **Verificación de llamadas**: Moq proporciona métodos para verificar si se realizaron llamadas a métodos específicos en el objeto simulado. Por ejemplo, se puede utilizar el método `Verify()` para comprobar si se llamó a un método en particular con los parámetros esperados.
- **ArgumentMatchers**: Moq ofrece soporte para ArgumentMatchers, que permiten definir reglas más flexibles al configurar el comportamiento de un método. Por ejemplo, se puede utilizar `It.IsAny<T>()` para aceptar cualquier valor de un tipo determinado como argumento.
- **Callback**: Moq permite utilizar el método `Callback()` para realizar acciones personalizadas cuando se llama a un método simulado. Esto puede ser útil para realizar verificaciones adicionales o ejecutar código adicional durante las pruebas.
- **Propiedades**: Moq permite configurar el comportamiento de propiedades en objetos simulados. Esto incluye establecer valores, permitir o evitar la lectura o escritura, y más.
- **Inyección de dependencias**: Moq se puede utilizar en combinación con patrones de inyección de dependencias para simular dependencias externas en las pruebas unitarias. Al utilizar Moq junto con

un contenedor de inyección de dependencias, se pueden proporcionar objetos simulados en lugar de las implementaciones reales durante las pruebas.

Proporciona una sintaxis fácil de usar para configurar objetos simulados y verificar su comportamiento durante las pruebas unitarias. A continuación, profundizaremos en algunas de las características clave de Moq y brindaremos un ejemplo para ilustrar su uso.

Una de las características principales de Moq es la capacidad de configurar el comportamiento de los métodos simulados. Puedes especificar valores de retorno, lanzar excepciones y realizar otras acciones personalizadas. Veamos un ejemplo:

Supongamos que tenemos una interfaz ICalculadora que define un método Sumar para sumar dos números enteros:

```
public interface ICalculadora
{
    int Sumar(int a, int b);
}
```

Ahora, queremos escribir una prueba unitaria para verificar si la clase CalculadoraServicio utiliza correctamente el método Sumar de ICalculadora. Usaremos Moq para simular ICalculadora en nuestra prueba.

Aquí está el ejemplo completo:

```
using Moq;

[TestClass]
public class CalculadoraServicioTests
{
    [TestMethod]
    public void CalcularSuma_UtilizaMetodoSumarDeCalculadora()
    {
        // Arrange
        var calculadoraMock = new Mock<ICalculadora>();
        calculadoraMock.Setup(calculadora => calculadora.Sumar(2, 3)).Returns(5); // Configuracion del metodo
      Sumar

        var servicio = new CalculadoraServicio(calculadoraMock.Object);

        // Act
        int resultado = servicio.CalcularSuma(2, 3);

        // Assert
        Assert.AreEqual(5, resultado);

```

```
        calculadoraMock.Verify(calculadora => calculadora.Sumar(2, 3), Times.Once); // Verificacion de
    llamada al metodo Sumar
    }
}
```

Moq es un framework de moqueo altamente flexible y poderoso para C#. Proporciona una sintaxis fácil de usar para configurar objetos simulados y verificar su comportamiento durante las pruebas unitarias. A continuación, profundizaremos en algunas de las características clave de Moq y brindaremos un ejemplo para ilustrar su uso.

Una de las características principales de Moq es la capacidad de configurar el comportamiento de los métodos simulados. Puedes especificar valores de retorno, lanzar excepciones y realizar otras acciones personalizadas. Veamos un ejemplo:

Supongamos que tenemos una interfaz ICalculadora que define un método Sumar para sumar dos números enteros:

```
public interface ICalculadora
{
    int Sumar(int a, int b);
}
```

Ahora, queremos escribir una prueba unitaria para verificar si la clase CalculadoraServicio utiliza correctamente el método Sumar de ICalculadora. Usaremos Moq para simular ICalculadora en nuestra prueba.

Aquí está el ejemplo completo:

```
[TestClass]
public class CalculadoraServicioTests
{
    [TestMethod]
    public void CalcularSuma_UtilizaMetodoSumarDeCalculadora()
    {
        // Arrange
        var calculadoraMock = new Mock<ICalculadora>();
        calculadoraMock.Setup(calculadora => calculadora.Sumar(2, 3)).Returns(5); // Configuracion del metodo
     Sumar

        var servicio = new CalculadoraServicio(calculadoraMock.Object);

        // Act
        int resultado = servicio.CalcularSuma(2, 3);

        // Assert
        Assert.AreEqual(5, resultado);

        calculadoraMock.Verify(calculadora => calculadora.Sumar(2, 3), Times.Once); // Verificacion de
    llamada al metodo Sumar
```

```
    }
}
```

En este ejemplo, creamos un objeto simulado de ICalculadora utilizando Mock<ICalculadora>. Luego, utilizamos el método Setup para configurar el comportamiento del método Sumar cuando se llame con los argumentos especificados (2 y 3). En este caso, configuramos que el método Sumar debe retornar 5.

Después, creamos una instancia de CalculadoraServicio y la pasamos el objeto simulado ICalculadora mediante calculadoraMock.Object. Luego, llamamos al método CalcularSuma en el servicio y verificamos que el resultado sea 5 utilizando Assert.AreEqual.

Finalmente, utilizamos Verify para verificar que el método Sumar de ICalculadora haya sido llamado exactamente una vez con los argumentos 2 y 3.

Este ejemplo ilustra cómo Moq nos permite configurar el comportamiento de los objetos simulados y verificar su uso durante las pruebas unitarias. Con Moq, podemos simular fácilmente dependencias y definir su comportamiento de manera controlada para asegurarnos de que nuestras clases se comporten correctamente en diferentes escenarios.

Simular un Repositorio con Moq

Veamos un ejemplo de cómo simular un repositorio utilizando Moq.

Supongamos que tenemos una interfaz IRepositorio<T>que define las operaciones básicas de un repositorio genérico, como agregar, obtener por ID y eliminar elementos:

```
public interface IRepositorio<T>
{
    void Agregar(T entidad);
    T ObtenerPorId(int id);
    void Eliminar(T entidad);
}

public class Servicio<T>
{
    private readonly IRepositorio<T> repositorio;

    public Servicio(IRepositorio<T> repositorio)
    {
        this.repositorio = repositorio;
    }

    public void Agregar(T entidad)
    {
        repositorio.Agregar(entidad);
    }
}
```

Queremos probar la clase Servicio<T>que depende de un repositorio y realiza operaciones utilizando ese repositorio. Utilizaremos Moq para simular el repositorio en nuestra prueba.

Aquí está el ejemplo completo:

```
[TestClass]
public class ServicioTests
{
    [TestMethod]
    public void Agregar_Entidad_LlamaMetodoAgregarDelRepositorio()
    {
        // Arrange
        var repositorioMock = new Mock<IRepositorio<Entidad>>();

        var servicio = new Servicio<Entidad>(repositorioMock.Object);

        var entidad = new Entidad();

        // Act
        servicio.Agregar(entidad);

        // Assert
        repositorioMock.Verify(repositorio => repositorio.Agregar(entidad), Times.Once);
    }
}
```

`Mock<IRepositorio<Entidad>>`. Luego, creamos una instancia de `Servicio<Entidad>` y le pasamos el objeto simulado del repositorio mediante `repositorioMock.Object`.

Después, creamos una entidad y llamamos al método Agregar en el servicio. Luego utilizamos Verify para verificar que el método Agregar del repositorio haya sido llamado exactamente una vez con la entidad especificada.