C# en .Net

Autor:

Kevin Cárdenas.

2023

Índice

1. Introducción **2**

1.1. Gramatica y Tipos 3

1.2. Operadores 5

1.3. Bucles, Sentencias if y Switch 7

1.4. Metodos y funciones 12

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.