C #(pronounced "C-sharp")

- ✓ is a multi-paradigm programming language developed by Microsoft as part of its .NET framework. It is widely used for developing Windows applications, web applications, and games (using the Unity game engine).

Key Features of C#:

1. **Ease of Use**: C# is designed to be simple and easy to learn, especially for those who have experience with C++ or Java.
2. **Automatic Memory Management**: C# includes automatic garbage collection, which helps manage memory efficiently.
3. **Safety**: C# provides strong typing and exception handling features, which enhance the safety and reliability of the code.
4. **Object-Oriented Programming (OOP)**: C# fully supports object-oriented programming, making it easier to create and maintain complex applications.
5. **Integration with .NET Framework**: C# can utilize the rich library of the .NET framework, which simplifies many common programming tasks.
6. **Support for Modern Features**: C# includes modern programming features like Language Integrated Query (LINQ), asynchronous programming, and the Task Parallel Library.

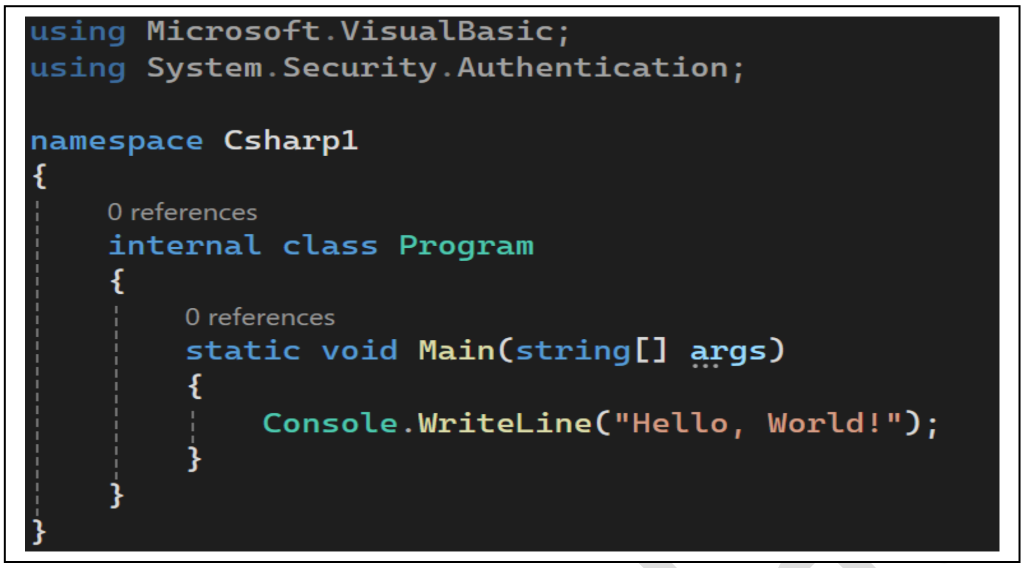
Examples of Applications Developed with C#:

- **Desktop Applications**: Using Windows Forms or WPF.
- **Web Applications**: Using ASP.NET.
- **Games**: Using Unity.
- **Services**: Such as Web Services and REST APIs.

Tools Used for Developing in C#:

- **Visual Studio**: The most common Integrated Development Environment (IDE) for C# development.
- **Rider**: A popular IDE developed by JetBrains that supports C#.

A Simple C# Code Example:



```
using Microsoft.VisualBasic;
using System.Security.Authentication;

namespace Csharp1
{
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

using Microsoft.VisualBasic;:

- This line imports the Microsoft Visual Basic namespace, which contains functions and tools to facilitate interaction with the Visual Basic programming language. In this example, the namespace is not used, but it could be useful for accessing specific features if needed.

using System.Security.Authentication;:

- This line imports the System Security Authentication namespace, which contains types related to authentication and security. In this example, the namespace is also not used, but it could be used to add security features to the application.

namespace Csharp1:

- The namespace Csharp1 is used to organize the code within it. You can think of namespaces as a way to group related classes together to avoid name conflicts.

internal class Program:

- This line defines a new class called Program. Classes are the basic units in object-oriented programming (OOP) and contain data (properties) and functions (methods).
- The keyword internal means that this class is accessible only within the same assembly.

static void Main (string [] args):

- This is the main entry point of the program. Every C# program must have at least one Main method.

- The keyword static means that this method belongs to the class itself rather than an instance of the class.
- void means that this method does not return any value.
- String [] args is a parameter passed to the Main method that contains any command-line arguments passed to the program.

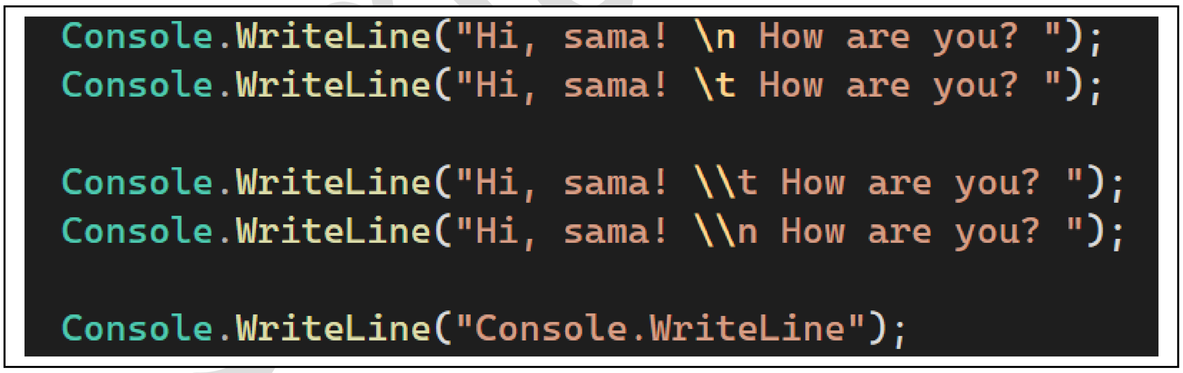
Console.WriteLine("Hello, World!");:

- This line prints the text "Hello, World!" to the console.
- Console is a class that provides functions to interact with the console.
- WriteLine is a method of the Console class used to print a line of text to the console.

How the Program Works:

1. When the program runs, the system calls the Main method.
2. The Main method executes all the instructions inside it.
3. In this case, the only instruction is Console.WriteLine("Hello, World!");, which prints the text "Hello, World!" to the console.
4. After executing all the instructions, the program ends.

Language Syntax:

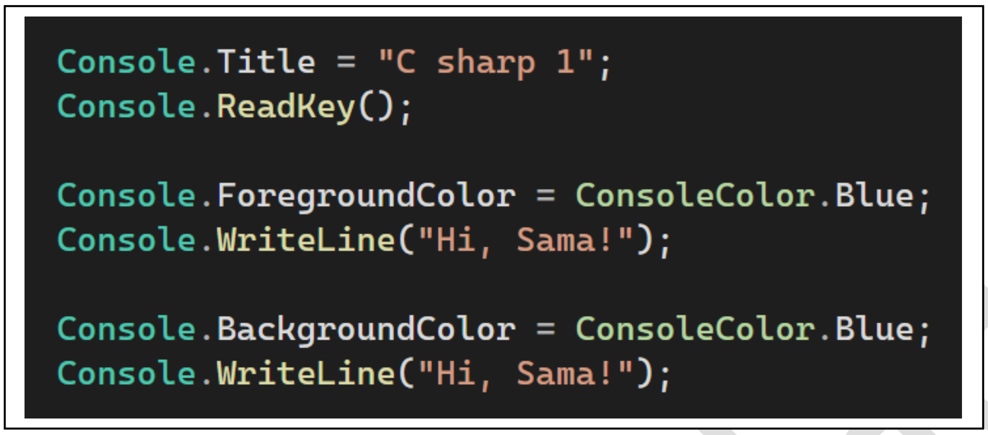


```
Console.WriteLine("Hi, sama! \n How are you? ");
Console.WriteLine("Hi, sama! \t How are you? ");

Console.WriteLine("Hi, sama! \\t How are you? ");
Console.WriteLine("Hi, sama! \\n How are you? ");

Console.WriteLine("Console.WriteLine");
```

Special Characters

- **\n (newline): Used to create a new line.**
- **\\ n not newline**
- **\t (tab): Used to insert a tab space (usually equivalent to 4 or 8 blank spaces, depending on settings).**
- **\\t not tab**
- **" " (String Literal) : Used to specify a text string.**

Writing to Console:



```
Console.Title = "C sharp 1";
Console.ReadKey();

Console.ForegroundColor = ConsoleColor.Blue;
Console.WriteLine("Hi, Sama!");

Console.BackgroundColor = ConsoleColor.Blue;
Console.WriteLine("Hi, Sama!");
```

- Title : to change the title of console
- Readkey () :This command is necessary because the program naturally executes and then closes. This command is necessary in order to display the address that I wrote.
- ForegroundColor : to change the color of text.
- BackgroundColor : to change the colour of background.

Variables

Because there is a variable, I must start with its data type

1.Basic rules for naming variables:

- ✓ The name must begin with a letter or an underscore (_).
- ✓ A name cannot start with a number.
- ✓ The name can contain letters (a-z, A-Z), numbers (0-9), and underscores (_).
- ✓ Special characters such as @, #, $, etc. are not allowed. Reserved words (such as int, class, public, etc.) cannot be used as names.
- ✓ A reserved word can be used if an @ sign is added before it, but this is not recommended.

2. Naming Conventions:

- ✓ The name begins with a lowercase letter, and each subsequent word begins with a capital letter.
- ✓ Used for local variables and parameters.
- ✓
- ✓ The name begins with a capital letter, and each subsequent word also begins with a capital letter.
- ✓ Used for classes, functions, properties, and events. Special fields begin with an underscore followed by Camel Case.

- ✓ Used to distinguish special fields in the category.

3. Additional rules:

- ✓ Clarification: Names should be clear and express the purpose of the variable, function, or class
- ✓ Appropriate length: Names should be neither too long nor too short.
- ✓ Consistency: Naming standards must be adhered to across the entire code to achieve consistency.

4.Types of variables:

Primitive types

Values are stored directly in memory and managed directly.

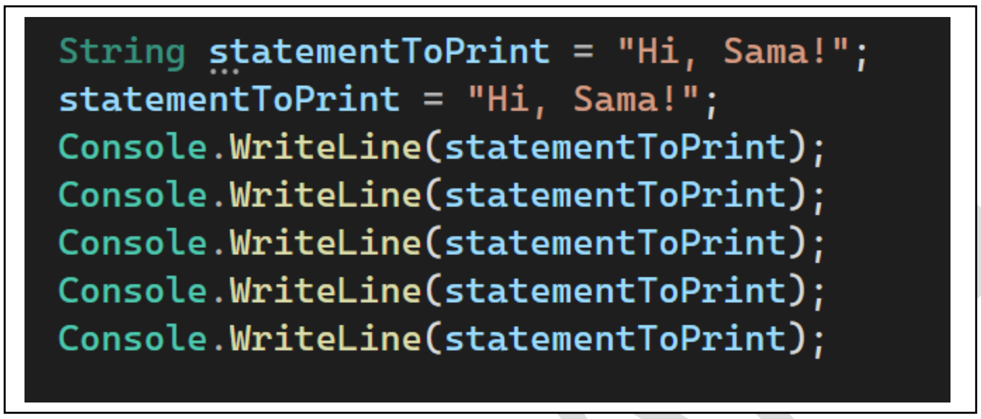
Type	**Discribtion**	**Example**
Int	represents a 32-bit integer.	int age = 30;
Long	represents a 64-bit integer.	long population = 7800000000;
Short	represents a 16-bit integer	short smallNumber = 32000;
Byte	represents an 8-bit positive integer	byte b = 255;
Float	represents a 32-bit float.	float temperature = 36.6f;
Double	represents a 64-bit decimal number.	double pi = 3.14159;
Decimal	represents a 128-bit decimal number, ideal for financial calculations.	decimal price = 19.99m;
Bool	represents a true or false boolean value.	bool isAlive = true;
Char	represents a single character in memory.	char initial = 'A';
Struct	represents a data structure consisting of a set of members.	struct Point { public int X; public int Y; }
Enum	represents an array of numeric constants.	enum Days { Sunday, Monday, Tuesday, Wednesday ….}

Reference Types

It stores an object's address in memory and allows flexible management of data and objects.

Type	**Description**	**Example**
String	represents a text string of Unicode characters.	string greeting = "Hello, World!";
Object	represents an object assignable to any type.	object obj = 42;
class	represents a data type that can contain data members and functions.	class Person { public string Name; public int Age; }

Array	represents an array of elements of the same type.	int[] numbers = new int[5];
Interface	represents a set of functional signatures that must be implemented by the classes that implement it.	interface IShape { void Draw(); }
Delegate	represents a type that allows a reference to one or more methods to be securely stored.	delegate void Operation(int x, int y);
Event	represents events in a C# application that are used to associate a constructor with a consumer.	event EventHandlerButtonClick;



```
String statementToPrint = "Hi, Sama!";
statementToPrint = "Hi, Sama!";
Console.WriteLine(statementToPrint);
Console.WriteLine(statementToPrint);
Console.WriteLine(statementToPrint);
Console.WriteLine(statementToPrint);
Console.WriteLine(statementToPrint);
```

- statementToPrint: It is a text string, number, or any other type that can be converted to text, the value of which will be displayed.

- **Boolean Data Type**

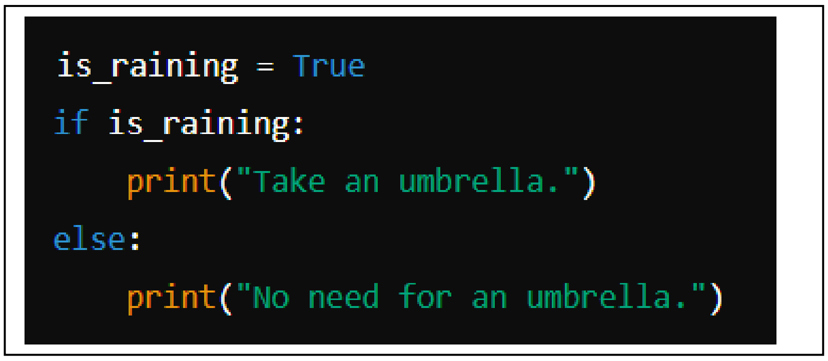
✓ is a data type that has one of two possible values: true or false. These values represent the two truth values of logic and Boolean algebra.

Booleans are used in programming for various purposes, including:

Conditional Statements:

1.**Conditional Statements:**

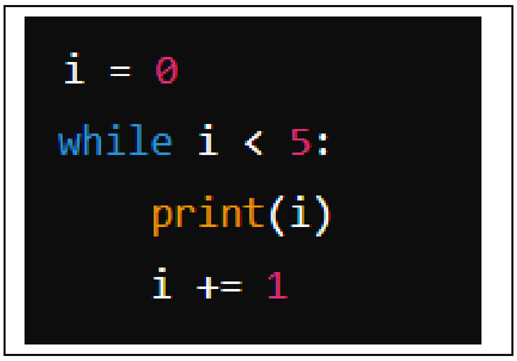
- Example: if, else, and switch statements.



```
is_raining = True
if is_raining:
    print("Take an umbrella.")
else:
    print("No need for an umbrella.")
```

2.Loops:

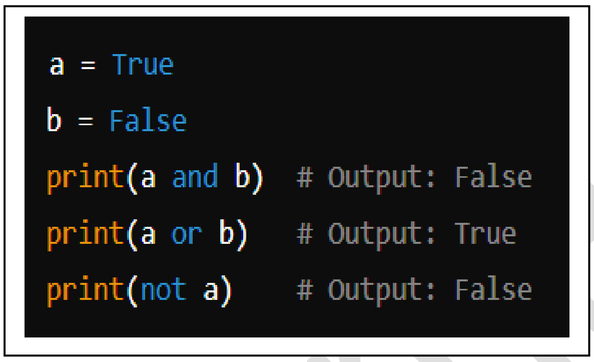
- Example: while loops.



```
i = 0
while i < 5:
    print(i)
    i += 1
```

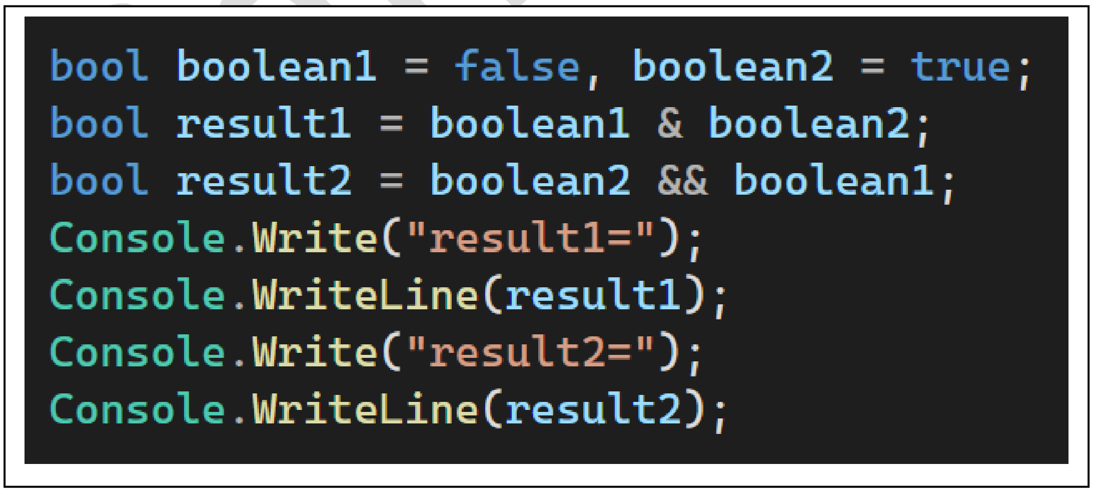
3. **Logical Operations:**

- Examples include AND, OR, and NOT operations:



```
a = True
b = False
print(a and b)  # Output: False
print(a or b)   # Output: True
print(not a)    # Output: False
```

- Examples for AND (& , &&) :



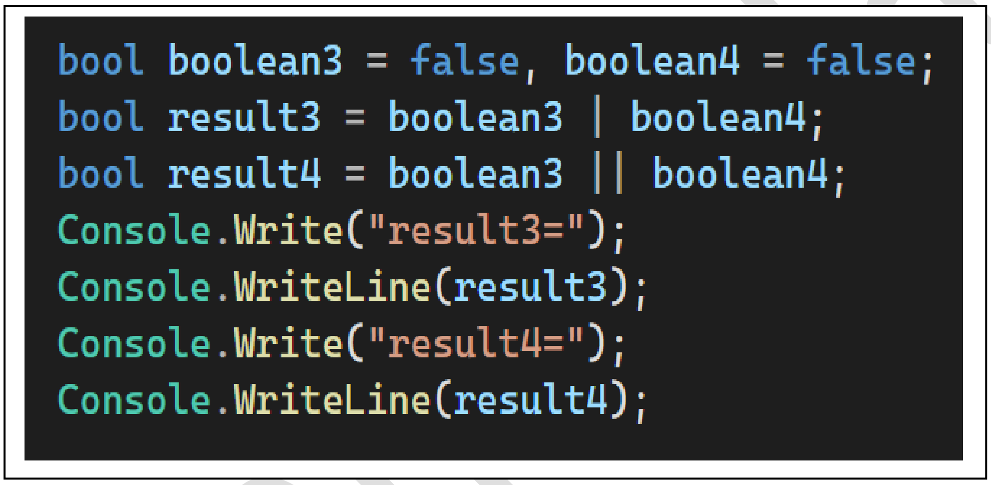
```
bool boolean1 = false, boolean2 = true;
bool result1 = boolean1 & boolean2;
bool result2 = boolean2 && boolean1;
Console.Write("result1=");
Console.WriteLine(result1);
Console.Write("result2=");
Console.WriteLine(result2);
```

&: Correct execution on each bit of the two values. If technical values are used, check the value completely and no short-term evaluation (short circuit evaluation) is performed.

&&: This operation is performed on Boolean values. Decide by short evaluation, which means that if the first value is false, you will not check with the second value and the final result will obviously be false.

Boolean1	**Boolean2**	**Boolean1& Boolean2** **Boolean1& Boolean2**
False	False	False
True	True	True
False	True	False
True	False	False

- Examples for OR (|| , |) :



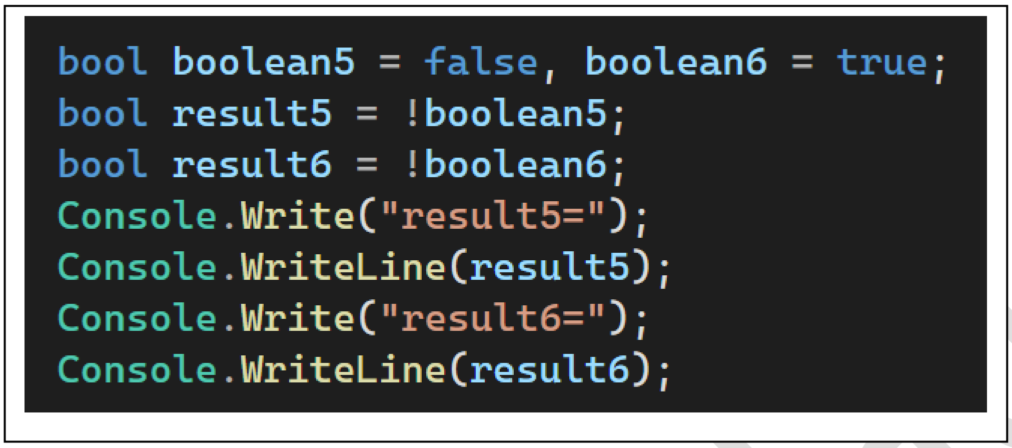
```
bool boolean3 = false, boolean4 = false;
bool result3 = boolean3 | boolean4;
bool result4 = boolean3 || boolean4;
Console.Write("result3=");
Console.WriteLine(result3);
Console.Write("result4=");
Console.WriteLine(result4);
```

|: The operation is performed on each bit of the two values. If used with Boolean values, it checks the full value but does not short-circuit evaluation.

||: This operation is performed on Boolean values and short-circuits the evaluation, which means that if the first value is true, it will not check the second value because it knows that the final result will be true

Boolean1	**Boolean2**	**Boolean1 | Boolean2** **Boolean1 || Boolean2**
False	False	False
True	True	True
False	True	True
True	False	True

- Examples for NOT (!) :

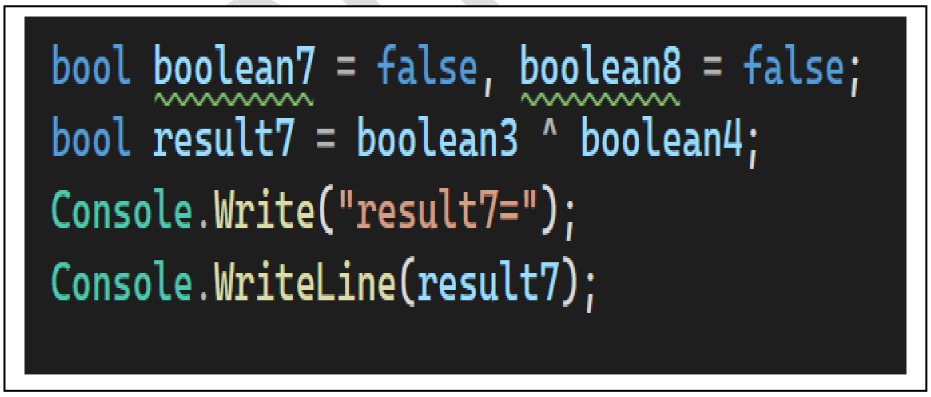


```
bool boolean5 = false, boolean6 = true;
bool result5 = !boolean5;
bool result6 = !boolean6;
Console.Write("result5=");
Console.WriteLine(result5);
Console.Write("result6=");
Console.WriteLine(result6);
```

It is a logical operation used to invert a Boolean value. If the original value is true, the result of the NOT operation will be false, and vice versa. In many programming languages, the NOT operation is represented using the !

Boolean1	**! Boolean**
False	True
True	False

- Examples for XOR (^) :



```
bool boolean7 = false, boolean8 = false;
bool result7 = boolean3 ^ boolean4;
Console.Write("result7=");
Console.WriteLine(result7);
```

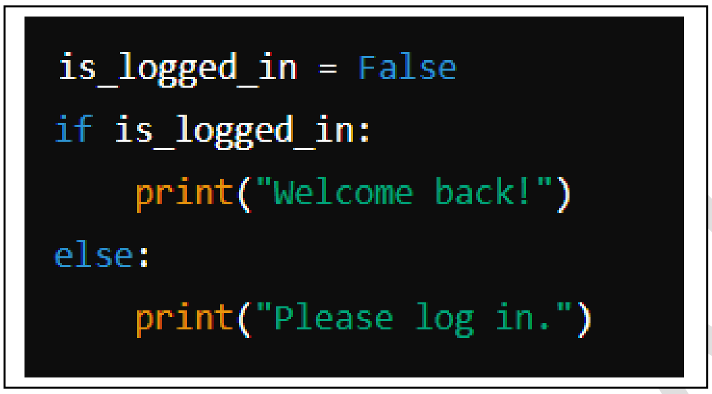
It is a logical operation used to determine whether two Boolean values are different. Returns true only if one value is true and the other is false. In many programming languages, the XOR operation is represented using the ^ sign.

Boolean1	**Boolean2**	**Boolean ^ Boolean2**
False	False	False

True	True	False
False	True	True
True	False	True

4. **Flags:**

- Boolean variables can be used as flags to indicate the status of a condition or to control the flow of a program.

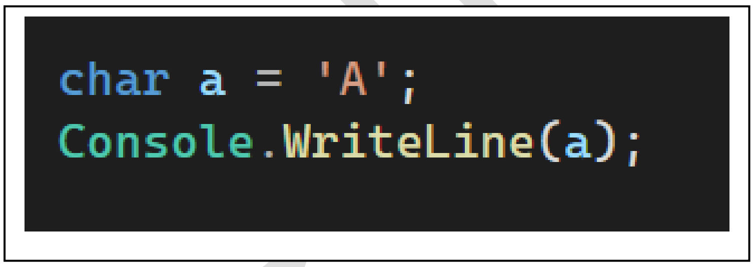


```python
is_logged_in = False
if is_logged_in:
    print("Welcome back!")
else:
    print("Please log in.")
```

Char Data Type:

The char type is used to store characters. The char type in C# represents a Unicode character and is stored as 16 bits (2 bytes), which means it can store 65,536 different values, which include letters, numbers, special characters, and any Unicode code.

Definition and initialization of a variable of type char:

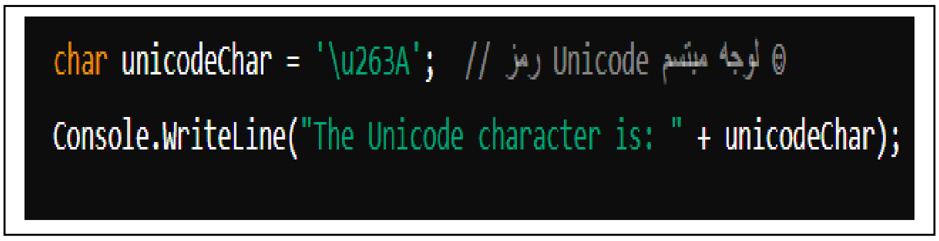


```csharp
char a = 'A';
Console.WriteLine(a);
```

Uses of the char type:

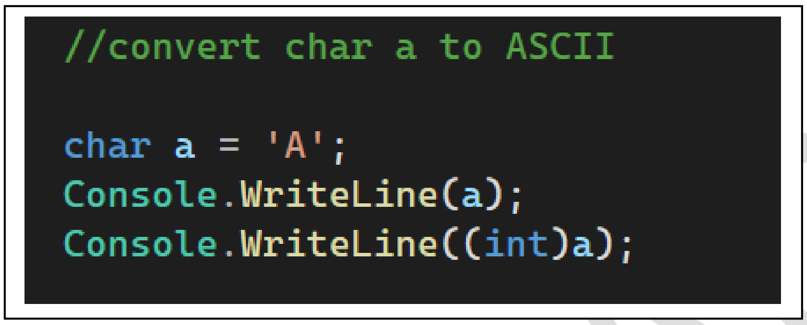
- **Storing Single Characters:** The char type is mainly used to store a single character.
- **Use of special characters**: Special characters such as '\n' (newline), '\t' (tab), and '\' (backslash) can be stored.
- **Use Unicode codes:** You can use Unicode codes directly.

Example of Unicode codes:



```
char unicodeChar = '\u263A';  // رمز Unicode لوجه مبتسم ☺
Console.WriteLine("The Unicode character is: " + unicodeChar);
```

Conversion to other types:



```
//convert char a to ASCII

char a = 'A';
Console.WriteLine(a);
Console.WriteLine((int)a);
```

Task1

Print the capital letters A to Z and then convert them into a system ASCI ?

Task2

Print the small letters a to z and then convert them into a system ASCI ?

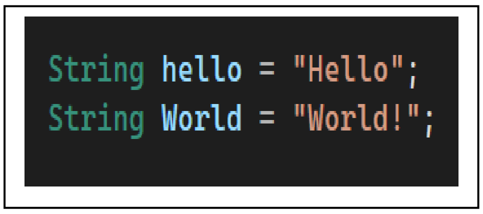
Task 3

Print the numbers from 0 to 9 and then convert them into a system ASCI?

String data type:

It is a reference data type used to store a string of characters. Strings in C# are immutable, which means that the content of a string cannot be changed after it is created. When you make modifications to a text string, a new string is created.

Definition and initialization of a variable of type string:



```
String hello = "Hello";
String World = "World!";
```

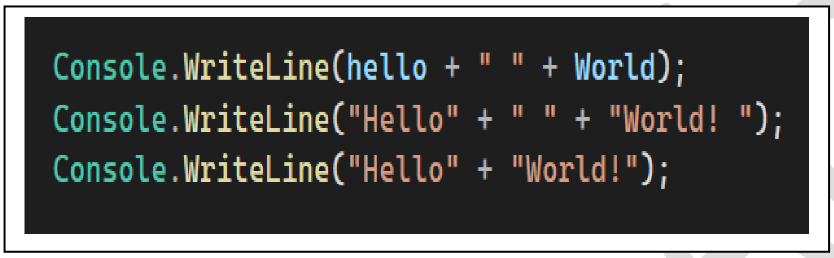
Characteristics and features of the string type:

- **Immutable:** The string cannot be modified after it is created.
- **Null pointer:** can contain the value null.
- **Full support for Unicode symbols:** can contain any Unicode symbol.
- **Built-in functions and features:** It contains many built-in functions and features for text processing

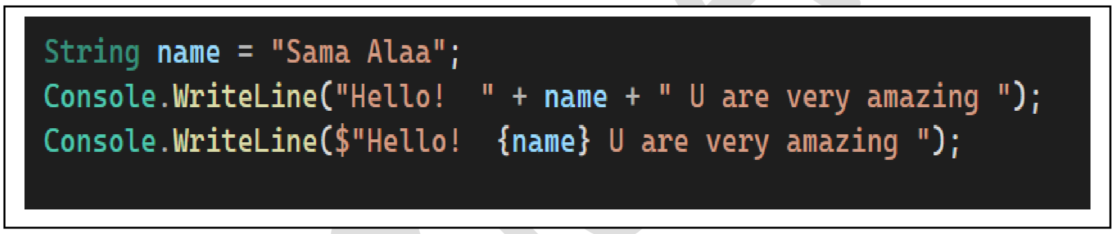
Basic operations on type string:

1.Contact

You can combine two text strings using the + operator or using the Concat function:



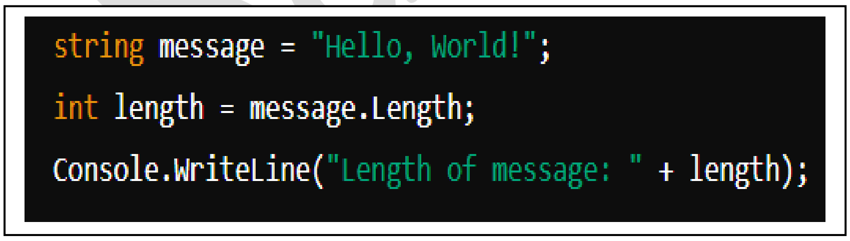
```
Console.WriteLine(hello + " " + World);
Console.WriteLine("Hello" + " " + "World! ");
Console.WriteLine("Hello" + "World!");
```



```
String name = "Sama Alaa";
Console.WriteLine("Hello!  " + name + " U are very amazing ");
Console.WriteLine($"Hello!  {name} U are very amazing ");
```

2. Length

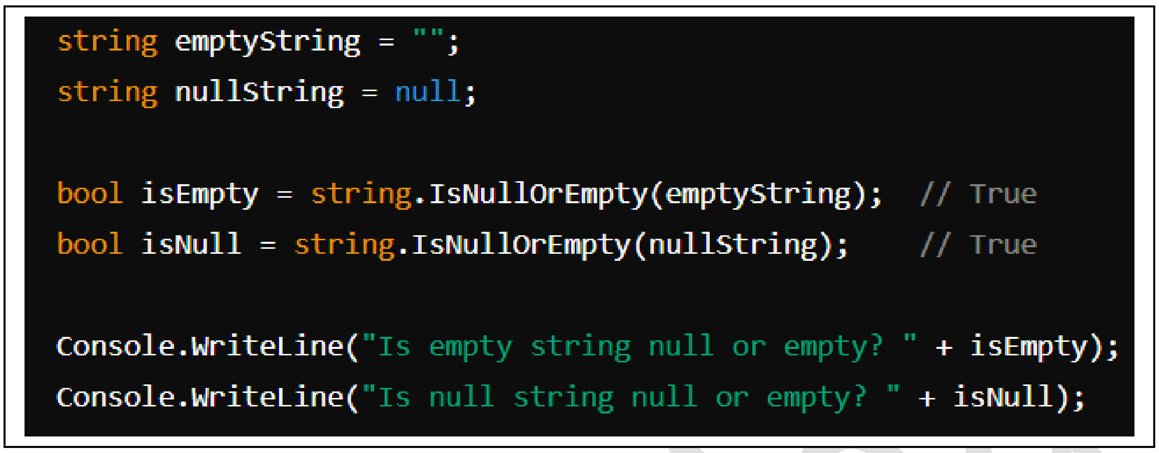
You can get the length of the string using the Length property:



```
string message = "Hello, World!";
int length = message.Length;
Console.WriteLine("Length of message: " + length);
```

3. Check for blank or zero signal

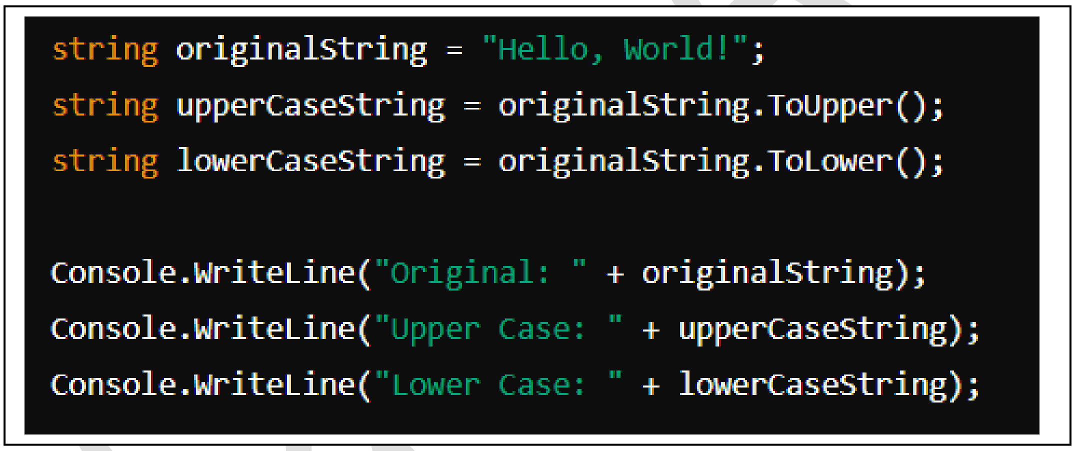
You can check if a string is empty or equal to null using the IsNullOrEmpty and IsNullOrWhiteSpace functions:



```
string emptyString = "";
string nullString = null;

bool isEmpty = string.IsNullOrEmpty(emptyString);  // True
bool isNull = string.IsNullOrEmpty(nullString);    // True

Console.WriteLine("Is empty string null or empty? " + isEmpty);
Console.WriteLine("Is null string null or empty? " + isNull);
```

4. Convert to uppercase and lowercase letters

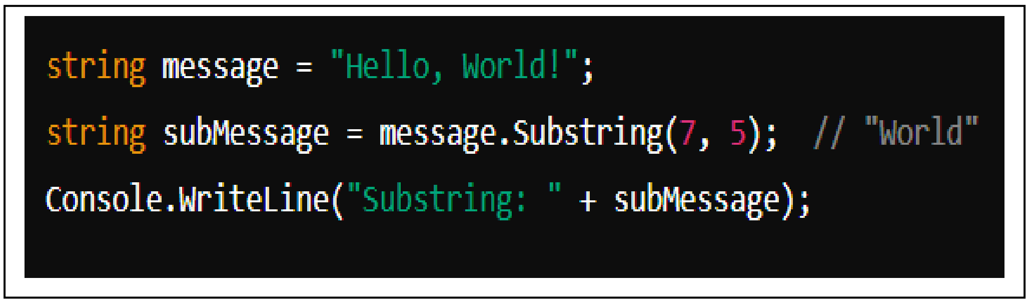
You can convert text string to uppercase or lowercase using the ToUpper and ToLower functions:



```
string originalString = "Hello, World!";
string upperCaseString = originalString.ToUpper();
string lowerCaseString = originalString.ToLower();

Console.WriteLine("Original: " + originalString);
Console.WriteLine("Upper Case: " + upperCaseString);
Console.WriteLine("Lower Case: " + lowerCaseString);
```

5. Built-in functions and features Substring

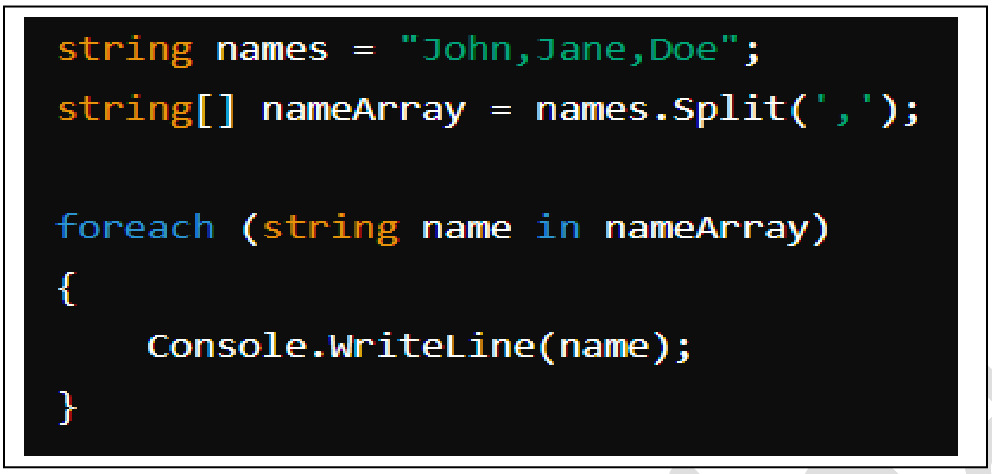
Extract part of the text string:



```
string message = "Hello, World!";
string subMessage = message.Substring(7, 5);  // "World"
Console.WriteLine("Substring: " + subMessage);
```

6. Split

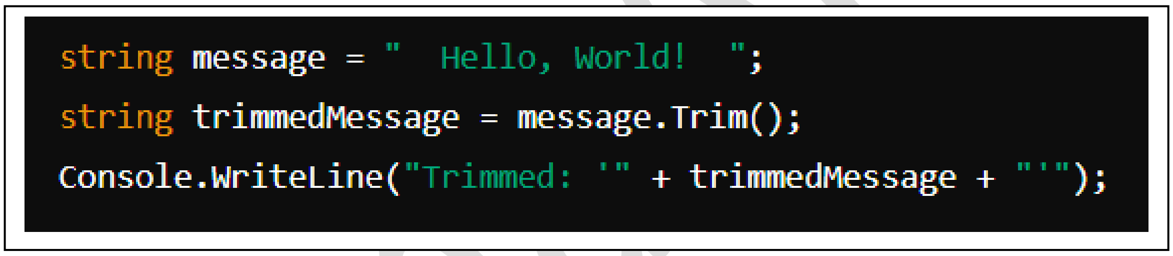
Splitting the text string into parts:



```
string names = "John,Jane,Doe";
string[] nameArray = names.Split(',');

foreach (string name in nameArray)
{
    Console.WriteLine(name);
}
```

7. Trim

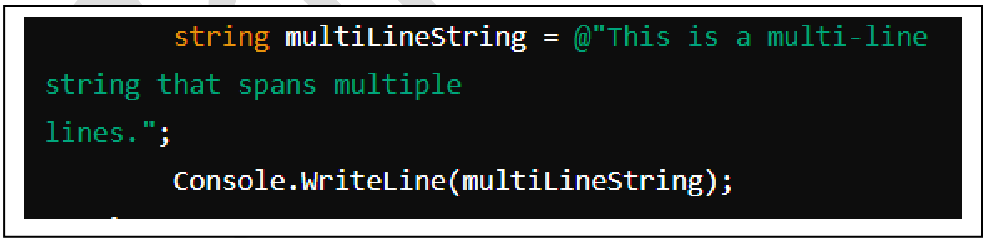
Remove spaces from the beginning and end of the string:



```
string message = "  Hello, World!  ";
string trimmedMessage = message.Trim();
Console.WriteLine("Trimmed: '" + trimmedMessage + "'");
```

8. Multi-line texts

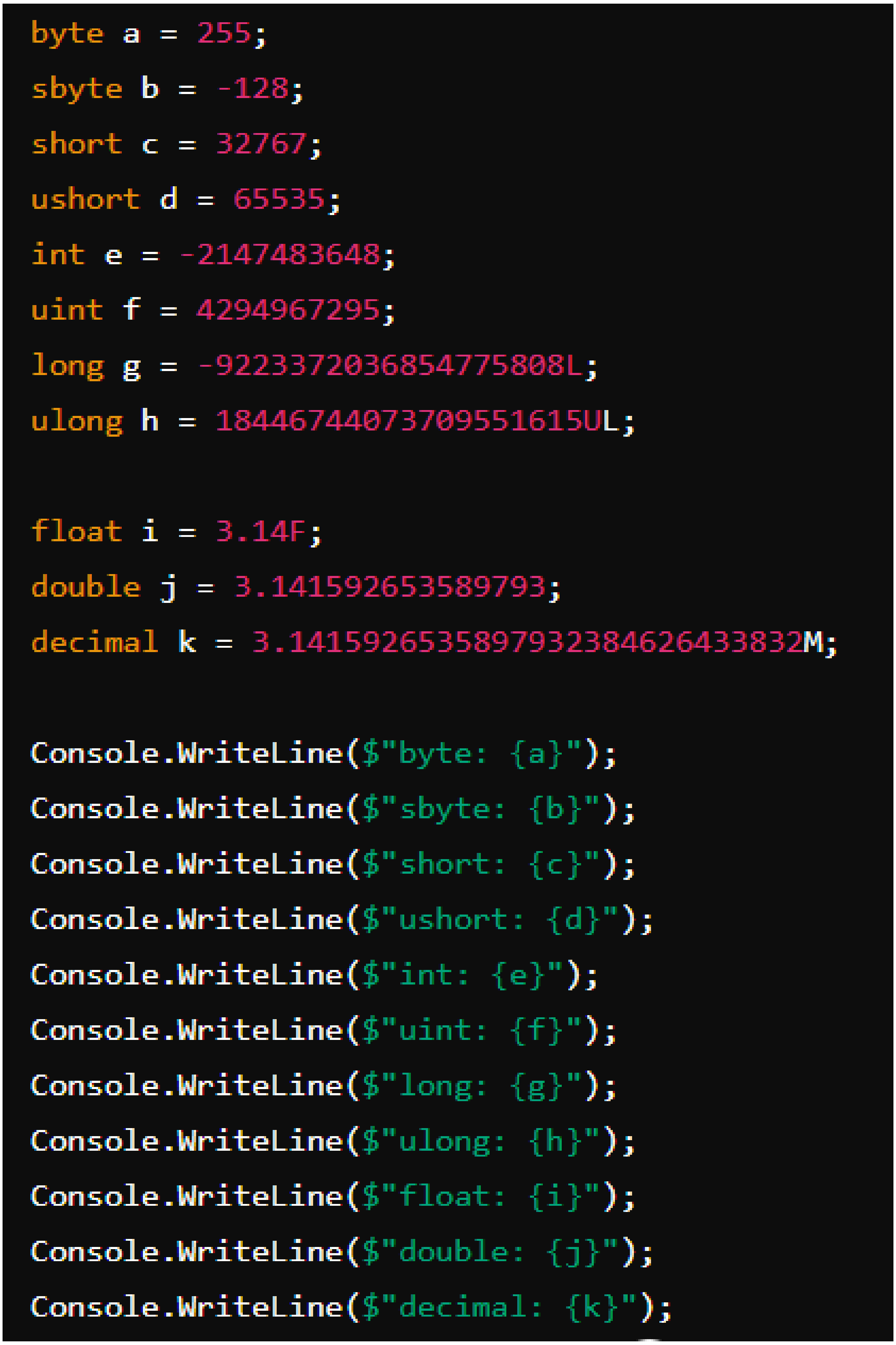
You can use multiline text using @ expressions:



```
        string multiLineString = @"This is a multi-line
string that spans multiple
lines.";
        Console.WriteLine(multiLineString);
```

Numeric Data Type:

There are several numeric data types that can be used as needed. These types differ in size (the number of bytes they use) and value range (the values they can store). Here is a list of basic numeric data types in C#.



```csharp
byte a = 255;
sbyte b = -128;
short c = 32767;
ushort d = 65535;
int e = -2147483648;
uint f = 4294967295;
long g = -9223372036854775808L;
ulong h = 18446744073709551615UL;

float i = 3.14F;
double j = 3.141592653589793;
decimal k = 3.1415926535897932384626433832M;

Console.WriteLine($"byte: {a}");
Console.WriteLine($"sbyte: {b}");
Console.WriteLine($"short: {c}");
Console.WriteLine($"ushort: {d}");
Console.WriteLine($"int: {e}");
Console.WriteLine($"uint: {f}");
Console.WriteLine($"long: {g}");
Console.WriteLine($"ulong: {h}");
Console.WriteLine($"float: {i}");
Console.WriteLine($"double: {j}");
Console.WriteLine($"decimal: {k}");
```

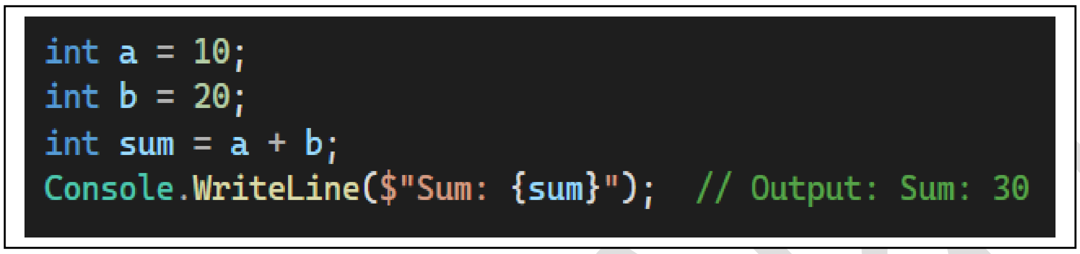
Arithmetic Operations :

You can use basic arithmetic operations to perform operations such as addition, subtraction, multiplication, division, and remainder. Let's review these processes with examples for each one:

1. Addition

Operation: +

Description: Adds two values together.

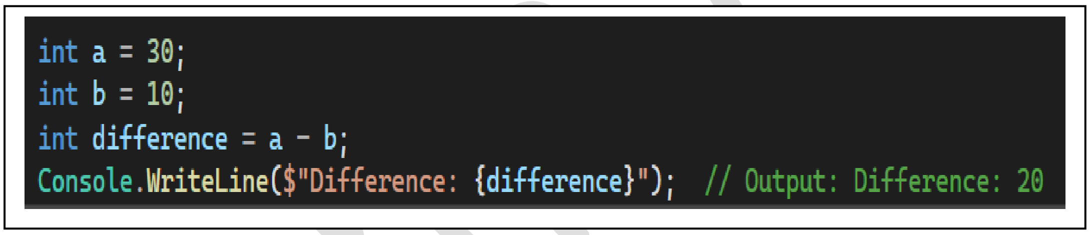


```
int a = 10;
int b = 20;
int sum = a + b;
Console.WriteLine($"Sum: {sum}");  // Output: Sum: 30
```

2.Subtraction

the operation: -

Description: Subtract one value from another.

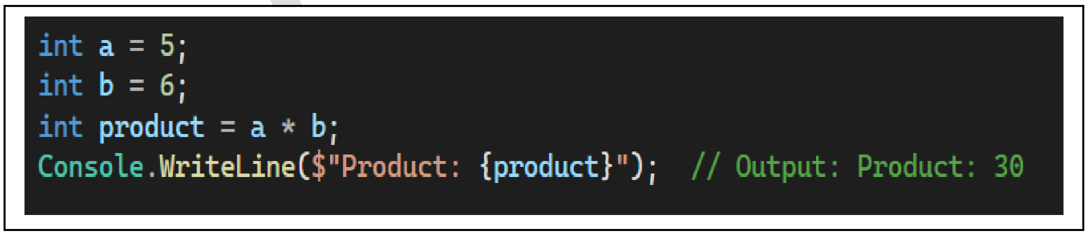


```
int a = 30;
int b = 10;
int difference = a - b;
Console.WriteLine($"Difference: {difference}");  // Output: Difference: 20
```

3. Multiplication

the operation: *

Description: Multiply two values together.

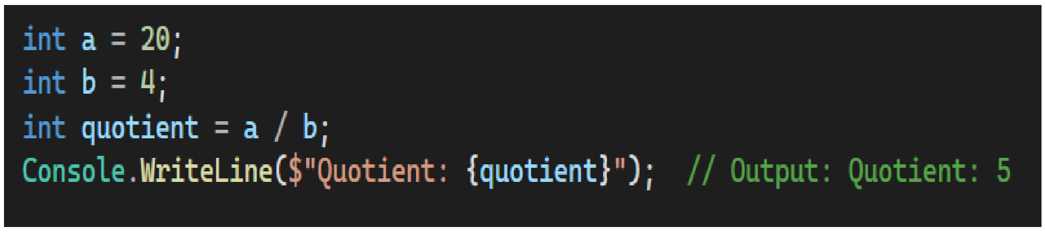


```
int a = 5;
int b = 6;
int product = a * b;
Console.WriteLine($"Product: {product}");  // Output: Product: 30
```

4. Division

the operation: /

Description: Divide one value by another. Division between integers gives an integer (rounded to zero).

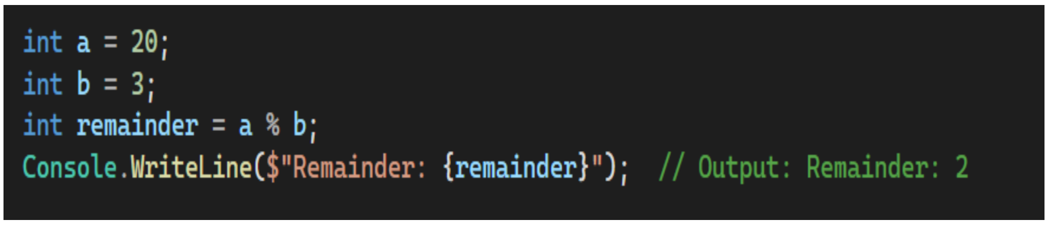


```
int a = 20;
int b = 4;
int quotient = a / b;
Console.WriteLine($"Quotient: {quotient}");  // Output: Quotient: 5
```

5. Modulus

the operation: %

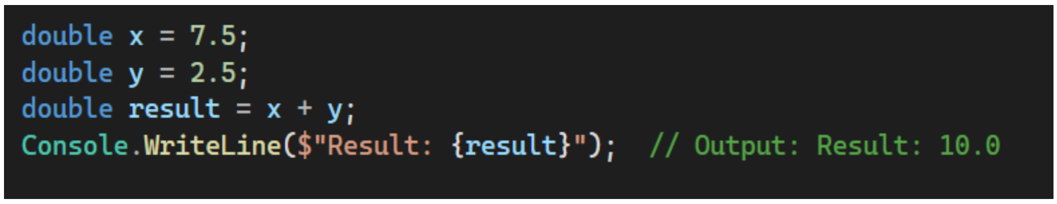
Description: The remainder is given when one value is divided by another.



```
int a = 20;
int b = 3;
int remainder = a % b;
Console.WriteLine($"Remainder: {remainder}");  // Output: Remainder: 2
```

6. Operations on decimal numbers

The same operations can be used on decimal numbers (float, double, decimal).

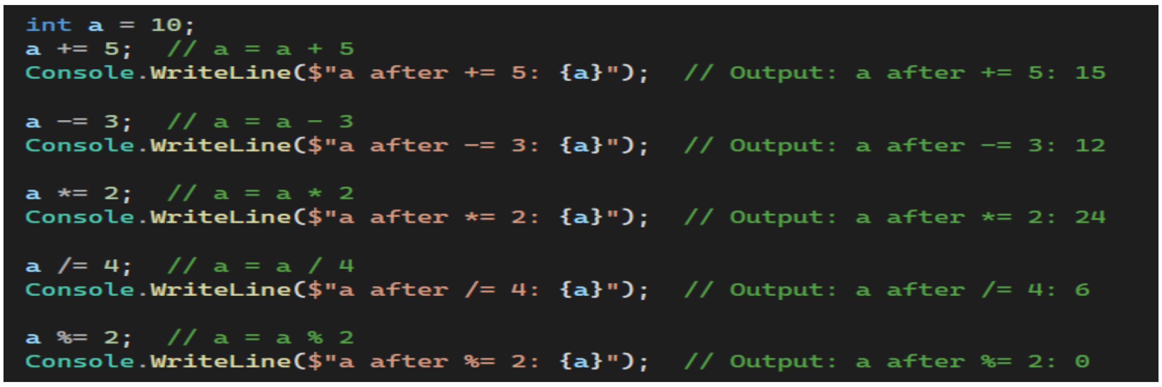


```
double x = 7.5;
double y = 2.5;
double result = x + y;
Console.WriteLine($"Result: {result}");  // Output: Result: 10.0
```

7. Compound Assignment Operators

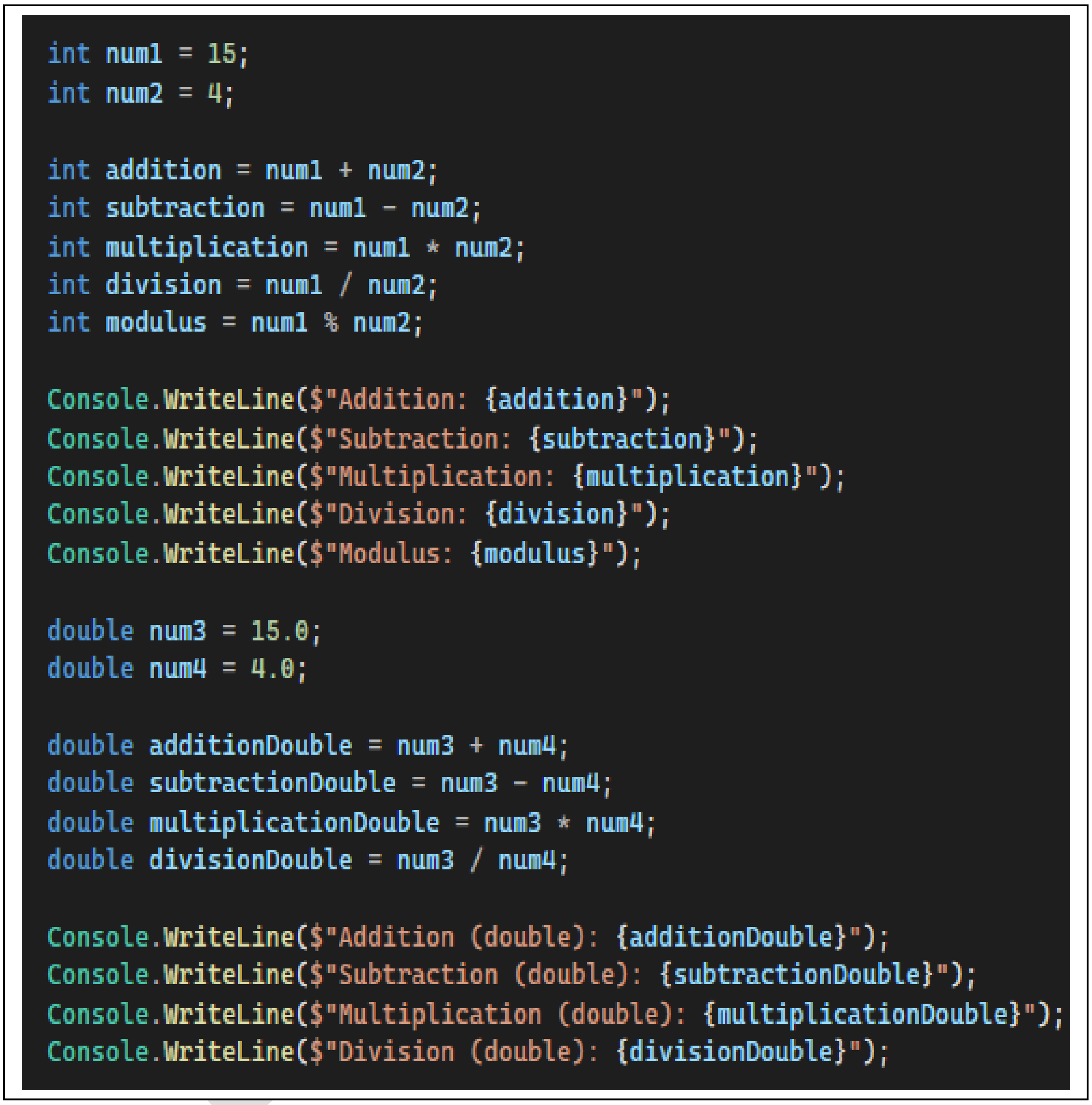
Operation: +=, -=, *=, /=, %=

Description: It performs the operation and stores the result in the original variable



```
int a = 10;
a += 5;  // a = a + 5
Console.WriteLine($"a after += 5: {a}");  // Output: a after += 5: 15

a -= 3;  // a = a - 3
Console.WriteLine($"a after -= 3: {a}");  // Output: a after -= 3: 12

a *= 2;  // a = a * 2
Console.WriteLine($"a after *= 2: {a}");  // Output: a after *= 2: 24

a /= 4;  // a = a / 4
Console.WriteLine($"a after /= 4: {a}");  // Output: a after /= 4: 6

a %= 2;  // a = a % 2
Console.WriteLine($"a after %= 2: {a}");  // Output: a after %= 2: 0
```

A practical example of all operations:



```
int num1 = 15;
int num2 = 4;

int addition = num1 + num2;
int subtraction = num1 - num2;
int multiplication = num1 * num2;
int division = num1 / num2;
int modulus = num1 % num2;

Console.WriteLine($"Addition: {addition}");
Console.WriteLine($"Subtraction: {subtraction}");
Console.WriteLine($"Multiplication: {multiplication}");
Console.WriteLine($"Division: {division}");
Console.WriteLine($"Modulus: {modulus}");

double num3 = 15.0;
double num4 = 4.0;

double additionDouble = num3 + num4;
double subtractionDouble = num3 - num4;
double multiplicationDouble = num3 * num4;
double divisionDouble = num3 / num4;

Console.WriteLine($"Addition (double): {additionDouble}");
Console.WriteLine($"Subtraction (double): {subtractionDouble}");
Console.WriteLine($"Multiplication (double): {multiplicationDouble}");
Console.WriteLine($"Division (double): {divisionDouble}");
```

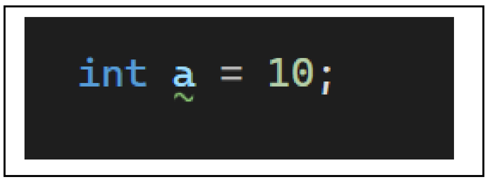
Assignment operators:

# To assign a value to a variable. In addition to the basic operators (=), there are a group of operators specific to the mnemonic and mnemonic compounds. Below is an explanation of these instructions with examples of each:

1. Basic operator

Primary assignment operator (=)

Description: Assigns a value to a variable.

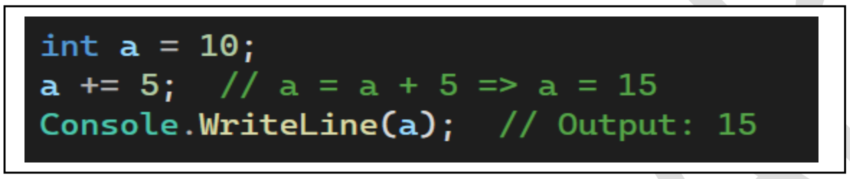


```
int a = 10;
```

2. Compound operators

Plural assignment operator (+=)

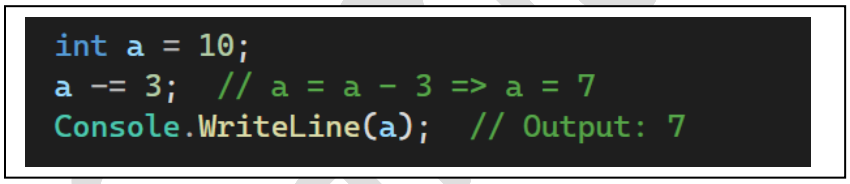
Description: Combines the value of a variable with a given value and then assigns the result to the variable.



```
int a = 10;
a += 5;  // a = a + 5 => a = 15
Console.WriteLine(a);  // Output: 15
```

3. Assignment operator with subtraction (-=)

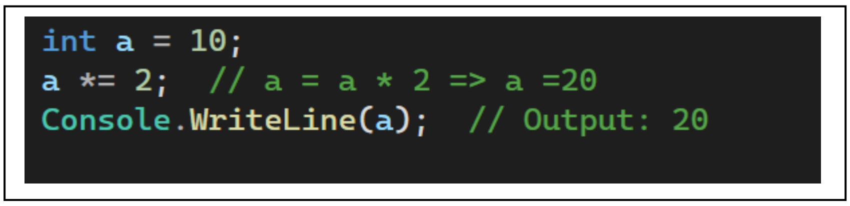
Description: Subtracts a given value from a variable and then assigns the result to the variable.



```
int a = 10;
a -= 3;  // a = a - 3 => a = 7
Console.WriteLine(a);  // Output: 7
```

4. Assignment operator with multiplication (*=)

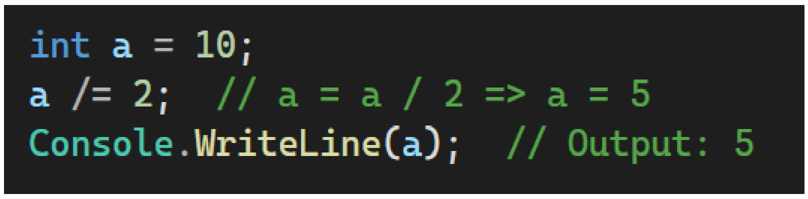
Description: Multiplying the variable by a certain value and then assigning the result to the variable.



```
int a = 10;
a *= 2;  // a = a * 2 => a =20
Console.WriteLine(a);  // Output: 20
```

5. Assignment operator with division (/=)

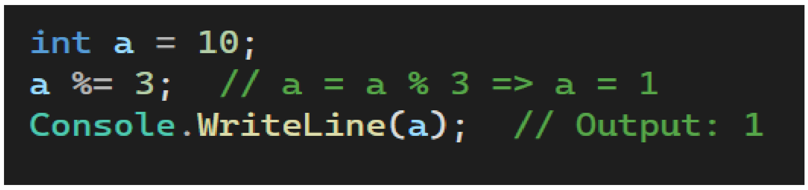
Description: Dividing the variable by a specific value and then assigning the result to the variable.



```
int a = 10;
a /= 2;  // a = a / 2 => a = 5
Console.WriteLine(a);  // Output: 5
```

6. Assignment operator with remainder (%=)

Description: Calculates the remainder of a variable with a given value and then assigns the result to the variable.

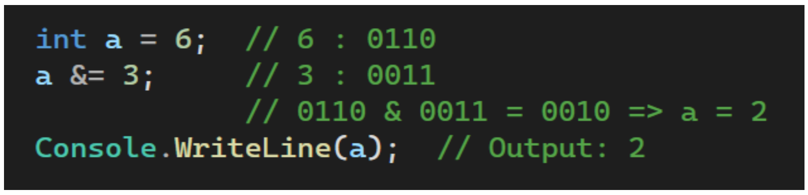


```
int a = 10;
a %= 3;  // a = a % 3 => a = 1
Console.WriteLine(a);  // Output: 1
```

Compound logical operators

7. Assignment operator with bit AND (&=)

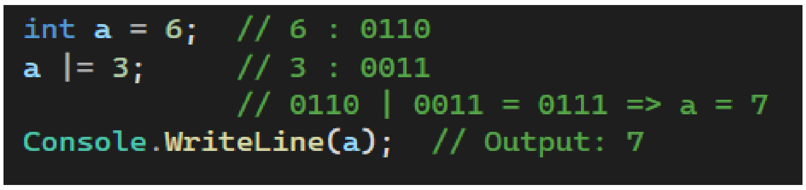
Description: Perform a bit-level AND operation between a variable and a value and then assign the result to the variable.



```
int a = 6;  // 6 : 0110
a &= 3;     // 3 : 0011
            // 0110 & 0011 = 0010 => a = 2
Console.WriteLine(a);  // Output: 2
```

8. Assignment operator with OR bit (|=)

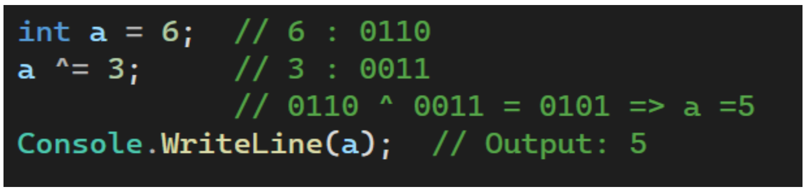
Description: Perform a bit-level OR operation between a variable and a value and then assign the result to the variable.



```
int a = 6;  // 6 : 0110
a |= 3;     // 3 : 0011
            // 0110 | 0011 = 0111 => a = 7
Console.WriteLine(a);  // Output: 7
```

9. Assignment operator with XOR bit (^=)

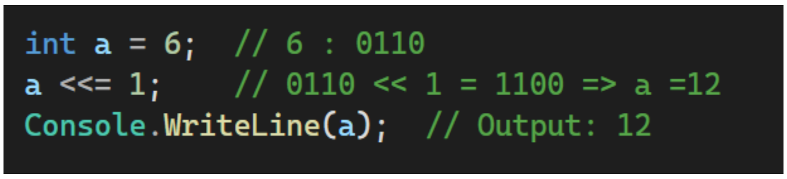
Description: Perform a bit-level XOR operation between a variable and a value and then assign the result to the variable.



```
int a = 6;   // 6 : 0110
a ^= 3;      // 3 : 0011
             // 0110 ^ 0011 = 0101 => a =5
Console.WriteLine(a);  // Output: 5
```

10. Assignment operator with left shift (<<=)

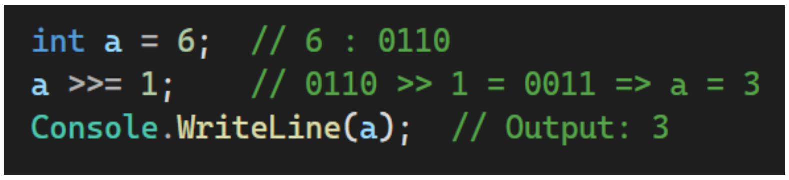
Description: Perform a bit-level shift to the left on a variable with a given value and then assign the result to the variable.



```
int a = 6;   // 6 : 0110
a <<= 1;     // 0110 << 1 = 1100 => a =12
Console.WriteLine(a);  // Output: 12
```

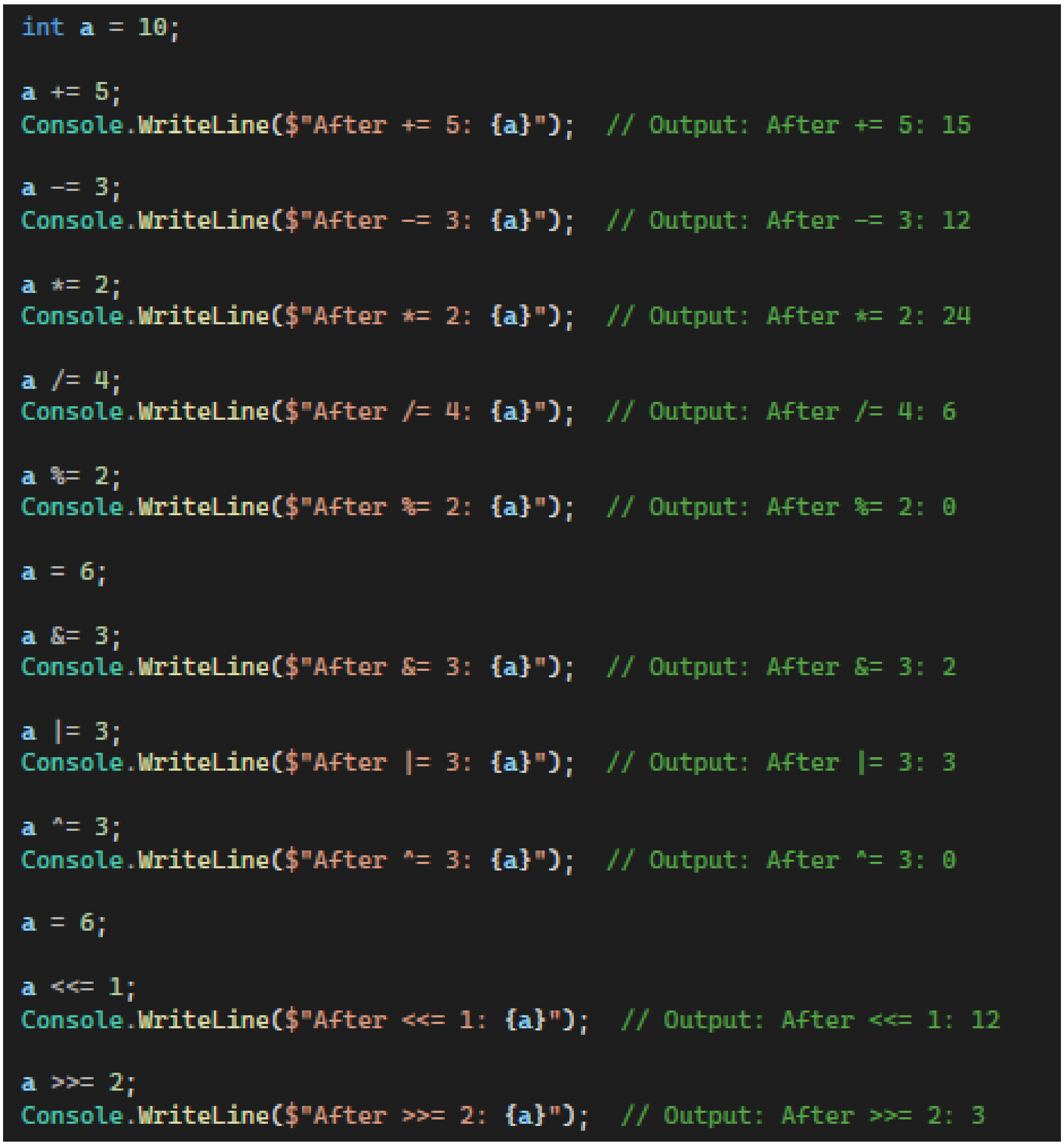
11. Operator with right shift (>>=)

Description: Perform a right bit shift on a variable with a given value and then assign the result to the variable.



```
int a = 6;   // 6 : 0110
a >>= 1;     // 0110 >> 1 = 0011 => a = 3
Console.WriteLine(a);  // Output: 3
```

A practical example illustrating all the complex operations:



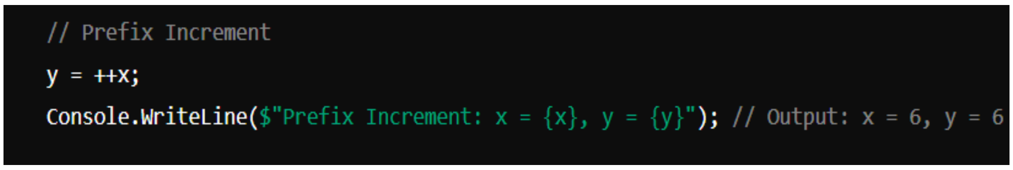
```
int a = 10;

a += 5;
Console.WriteLine($"After += 5: {a}");  // Output: After += 5: 15

a -= 3;
Console.WriteLine($"After -= 3: {a}");  // Output: After -= 3: 12

a *= 2;
Console.WriteLine($"After *= 2: {a}");  // Output: After *= 2: 24

a /= 4;
Console.WriteLine($"After /= 4: {a}");  // Output: After /= 4: 6

a %= 2;
Console.WriteLine($"After %= 2: {a}");  // Output: After %= 2: 0

a = 6;

a &= 3;
Console.WriteLine($"After &= 3: {a}");  // Output: After &= 3: 2

a |= 3;
Console.WriteLine($"After |= 3: {a}");  // Output: After |= 3: 3

a ^= 3;
Console.WriteLine($"After ^= 3: {a}");  // Output: After ^= 3: 0

a = 6;

a <<= 1;
Console.WriteLine($"After <<= 1: {a}");  // Output: After <<= 1: 12

a >>= 2;
Console.WriteLine($"After >>= 2: {a}");  // Output: After >>= 2: 3
```

Increment & Decrement operators :

increment (++) and decrement (--) operators are used to increase or decrease the value of a variable by 1. These operators can be used in two ways: **prefix** and **postfix**.

1. Prefix Increment/Decrement

- **Prefix Increment (++x)**: Increases the value of x by 1 and then returns the new value.



```
// Prefix Increment
y = ++x;
Console.WriteLine($"Prefix Increment: x = {x}, y = {y}"); // Output: x = 6, y = 6
```

- **Prefix Decrement (--x)**: Decreases the value of x by 1 and then returns the new value.

```
// Prefix Decrement
y = --x;
Console.WriteLine($"Prefix Decrement: x = {x}, y = {y}"); // Output: x = 6, y = 6
```

2. Postfix Increment/Decrement

- **Postfix Increment (x++)**: Returns the current value of x, then increases the value of x by 1.

```
// Postfix Increment
y = x++;
Console.WriteLine($"Postfix Increment: x = {x}, y = {y}"); // Output: x = 7, y = 6
```

- **Postfix Decrement (x--)**: Returns the current value of x, then decreases the value of x by 1.

```
// Postfix Decrement
y = x--;
Console.WriteLine($"Postfix Decrement: x = {x}, y = {y}"); // Output: x = 5, y = 6
```

Explanation:

- **Prefix Increment (++x)**: x is increased first, and then its value is assigned to y.
- **Postfix Increment (x++)**: The current value of x is assigned to y, and then x is increased.
- **Prefix Decrement (--x)**: x is decreased first, and then its value is assigned to y.
- **Postfix Decrement (x--)**: The current value of x is assigned to y, and then x is decreased.

Comparison Operators:

Comparison operators in C# are used to compare two values. The result of a comparison is always a Boolean value (true or false). Here are the main comparison operators in C#:

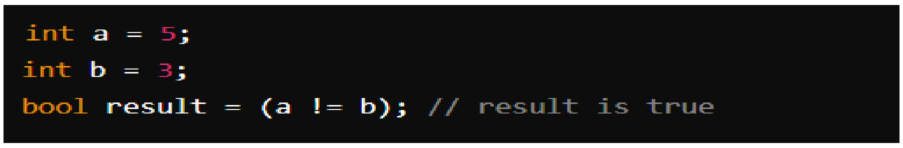
1. Equal To (==)

- Compares two values for equality.
- **Example**

```
int a = 5;
int b = 5;
bool result = (a == b); // result is true
```

2. Not Equal To (!=)

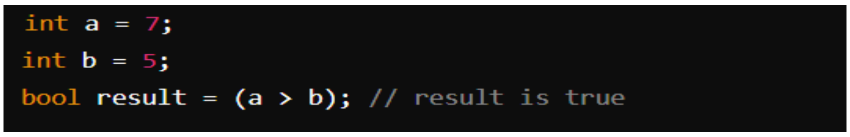
- Compares two values to check if they are not equal.
- **Example:**



```csharp
int a = 5;
int b = 3;
bool result = (a != b); // result is true
```

3. Greater Than (>)

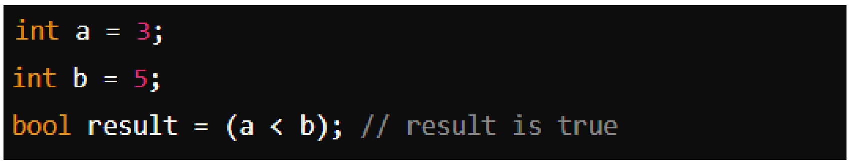
- Checks if the left-hand operand is greater than the right-hand operand.
- **Example:**



```csharp
int a = 7;
int b = 5;
bool result = (a > b); // result is true
```

4. Less Than (<)

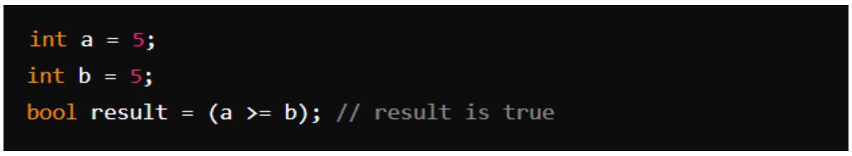
- Checks if the left-hand operand is less than the right-hand operand.
- **Example:**



```csharp
int a = 3;
int b = 5;
bool result = (a < b); // result is true
```

5. Greater Than or Equal To (>=)

- Checks if the left-hand operand is greater than or equal to the right-hand operand.
- **Example:**



```csharp
int a = 5;
int b = 5;
bool result = (a >= b); // result is true
```

6. Less Than or Equal To (<=)

- Checks if the left-hand operand is less than or equal to the right-hand operand.
- **Example:**

```
int a = 3;
int b = 5;
bool result = (a <= b); // result is true
```

Example Code:

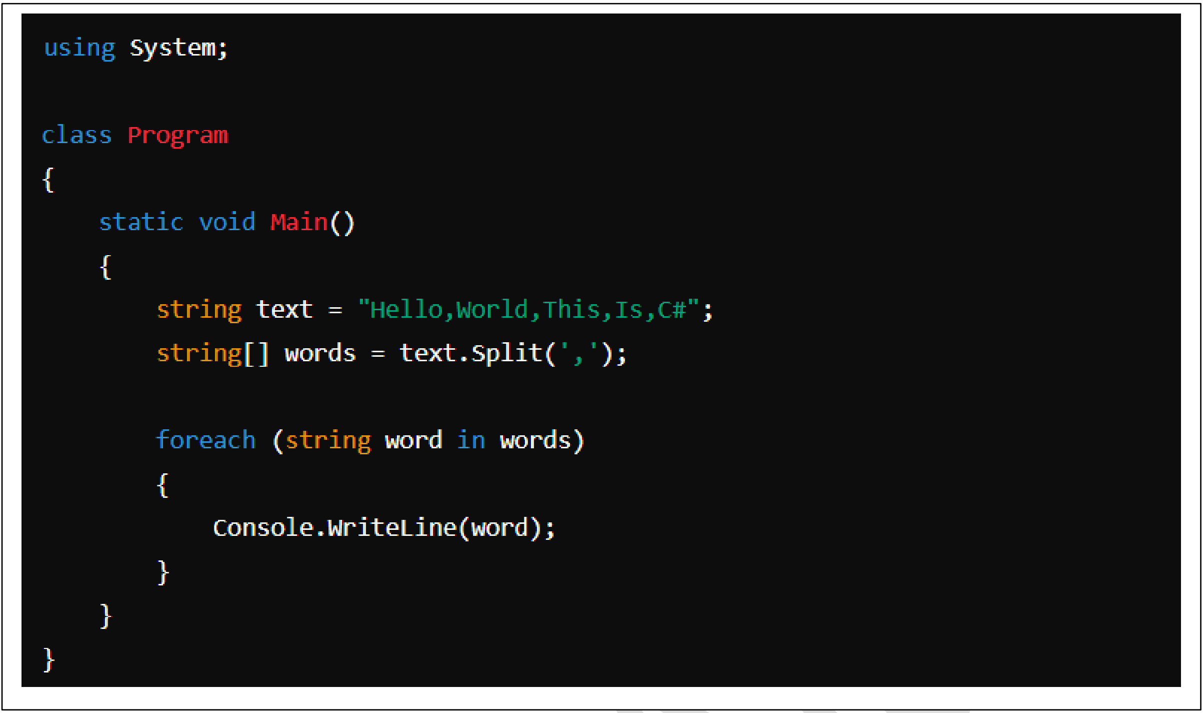
```
using System;

class Program
{
    static void Main()
    {
        int x = 10;
        int y = 20;

        Console.WriteLine(x == y);  // Output: False
        Console.WriteLine(x != y);  // Output: True
        Console.WriteLine(x > y);   // Output: False
        Console.WriteLine(x < y);   // Output: True
        Console.WriteLine(x >= 10); // Output: True
        Console.WriteLine(x <= 10); // Output: True
    }
}
```

Text Analysis and Manipulation:

In C#, text analysis and manipulation can be achieved using various tools and functions provided by the language. These tools can be used for tasks such as splitting text, replacing words, converting text to uppercase or lowercase, and many other operations. Below are some common examples of text analysis and manipulation in C#.

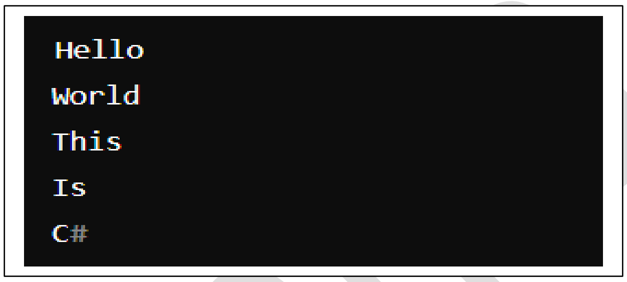
1. Splitting Strings

You can split a string into parts using the Split method, which relies on a character or a set of characters as a delimiter.



```
using System;

class Program
{
    static void Main()
    {
        string text = "Hello,World,This,Is,C#";
        string[] words = text.Split(',');

        foreach (string word in words)
        {
            Console.WriteLine(word);
        }
    }
}
```

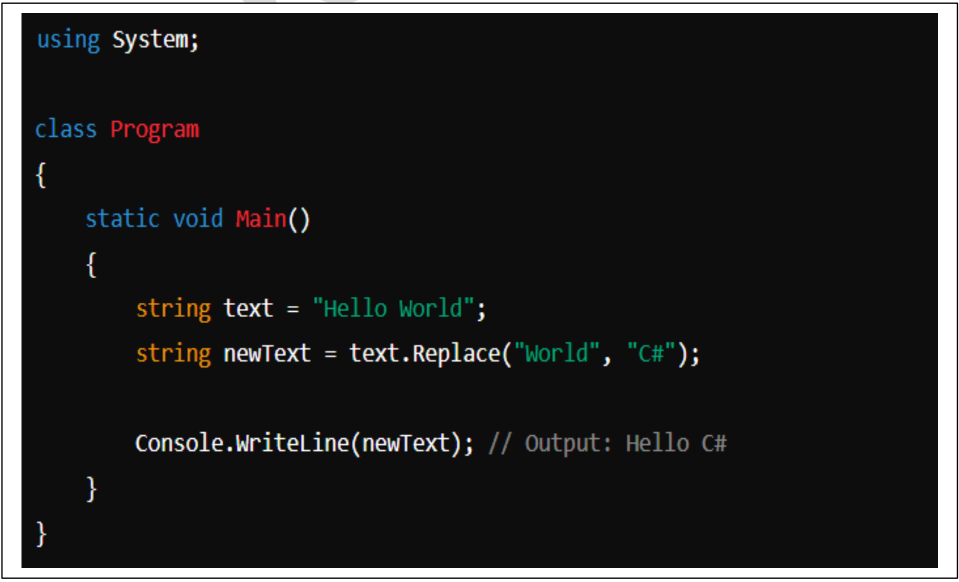
The output:



```
Hello
World
This
Is
C#
```

2. Replacing Strings

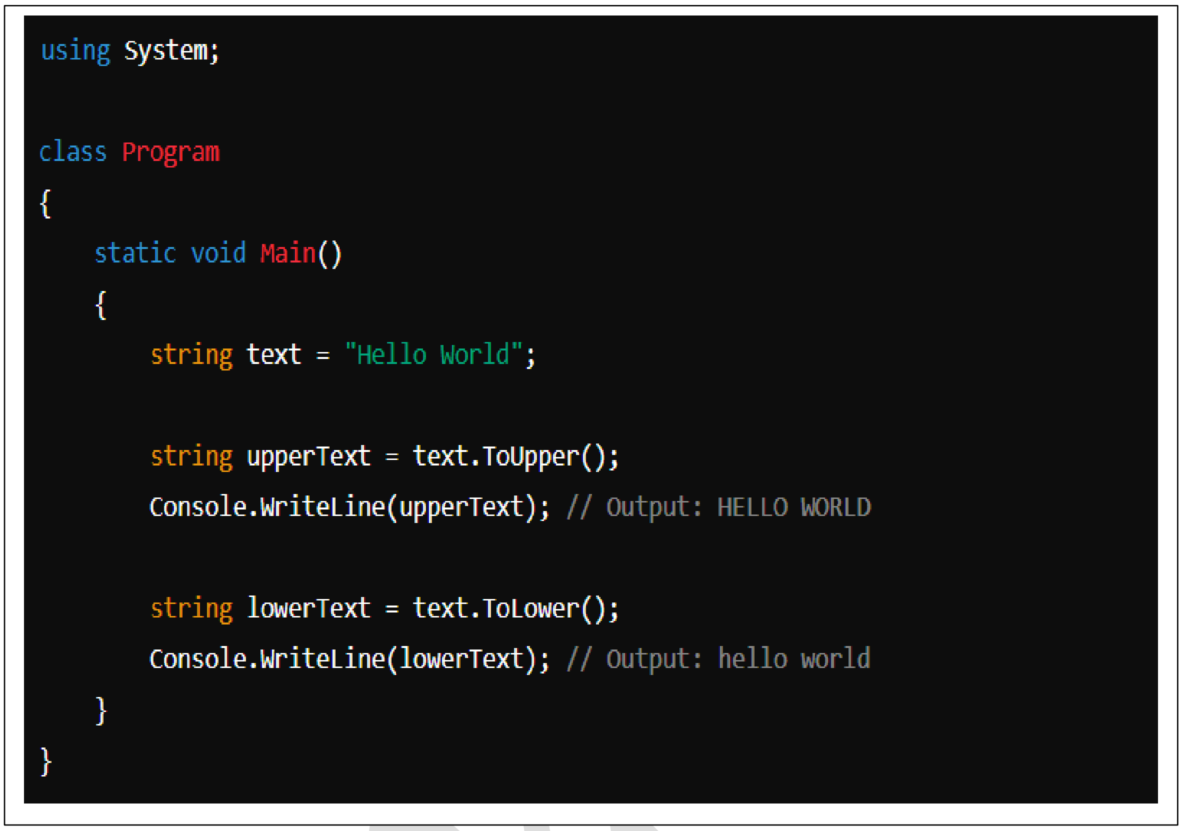
You can replace a specific part of the text using the Replace method.



```
using System;

class Program
{
    static void Main()
    {
        string text = "Hello World";
        string newText = text.Replace("World", "C#");

        Console.WriteLine(newText); // Output: Hello C#
    }
}
```

3. Changing Case

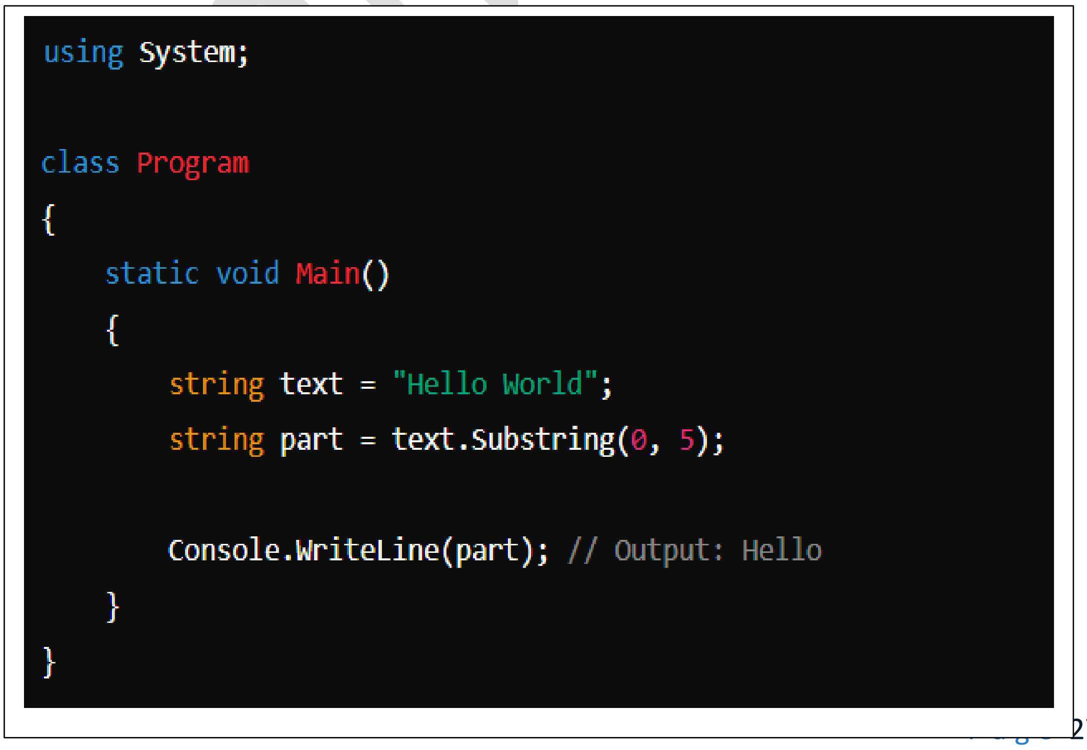
You can convert text to uppercase using ToUpper or to lowercase using ToLower.



```csharp
using System;

class Program
{
    static void Main()
    {
        string text = "Hello World";

        string upperText = text.ToUpper();
        Console.WriteLine(upperText); // Output: HELLO WORLD

        string lowerText = text.ToLower();
        Console.WriteLine(lowerText); // Output: hello world
    }
}
```

4. Extracting Substrings

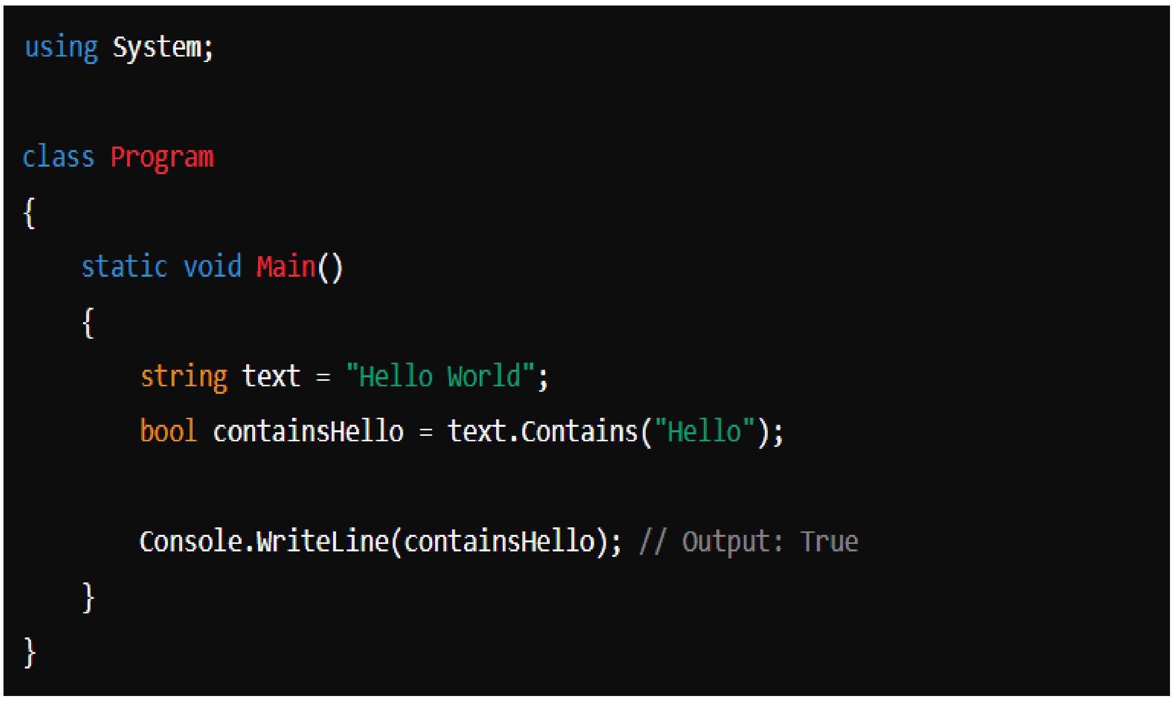
You can extract a specific part of the text using the Substring method.



```csharp
using System;

class Program
{
    static void Main()
    {
        string text = "Hello World";
        string part = text.Substring(0, 5);

        Console.WriteLine(part); // Output: Hello
    }
}
```

5. Checking for Substring Presence

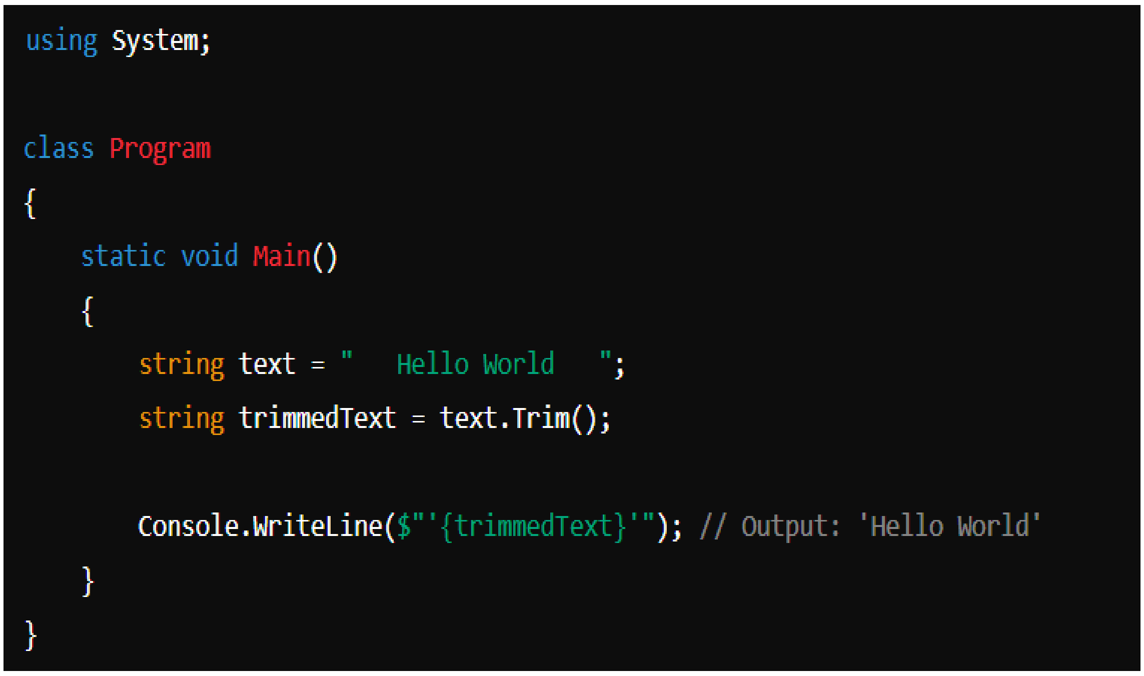
You can check for the presence of a specific text within another text using the Contains method.



```
using System;

class Program
{
    static void Main()
    {
        string text = "Hello World";
        bool containsHello = text.Contains("Hello");

        Console.WriteLine(containsHello); // Output: True
    }
}
```

6. Trimming Whitespaces

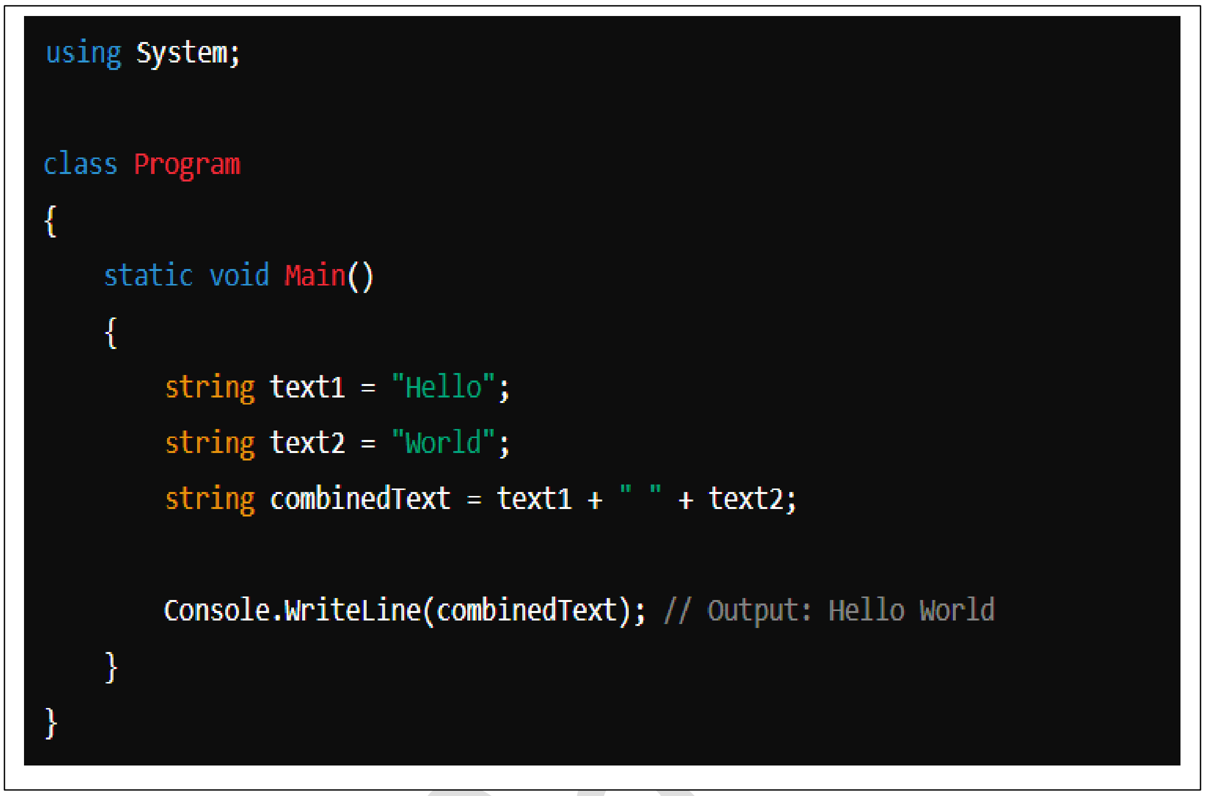
You can remove excess whitespaces at the beginning or end of the text using Trim, TrimStart, or TrimEnd.



```
using System;

class Program
{
    static void Main()
    {
        string text = "   Hello World   ";
        string trimmedText = text.Trim();

        Console.WriteLine($"'{trimmedText}'"); // Output: 'Hello World'
    }
}
```

7. Concatenating Strings

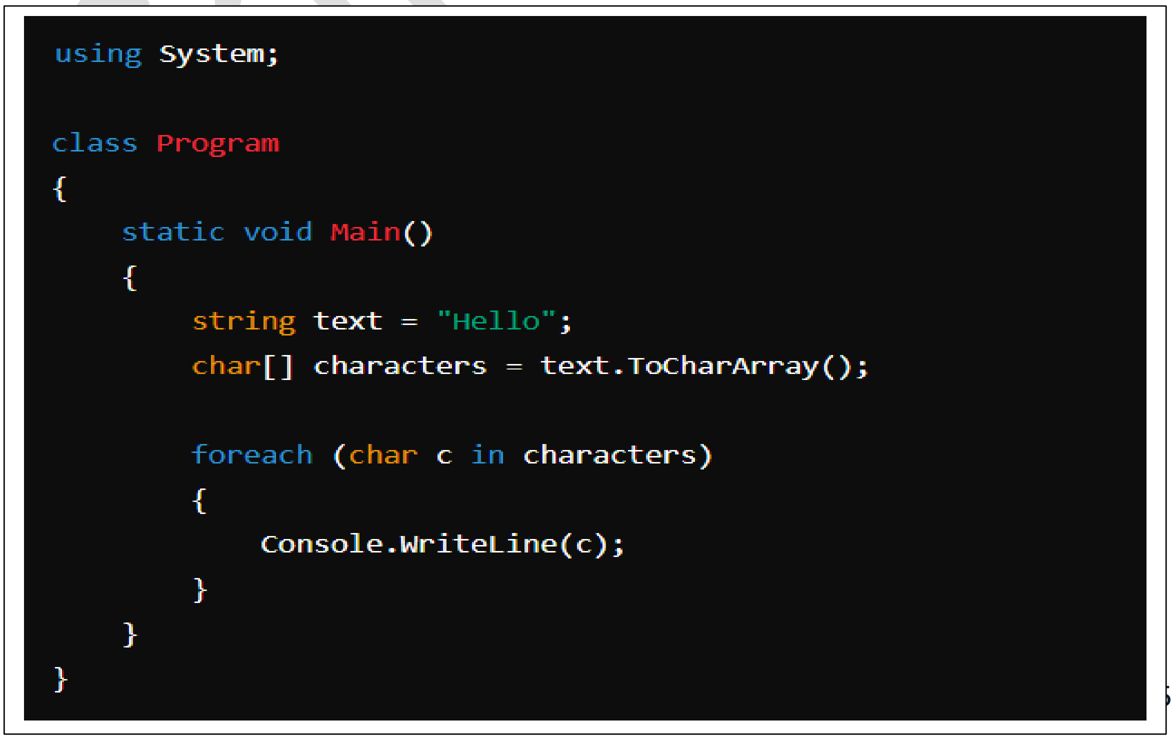
You can concatenate multiple strings using the + operator or using String.Concat.



```
using System;

class Program
{
    static void Main()
    {
        string text1 = "Hello";
        string text2 = "World";
        string combinedText = text1 + " " + text2;

        Console.WriteLine(combinedText); // Output: Hello World
    }
}
```

8. Converting Text to Character Array

You can convert a string to a character array using the ToCharArray method.



```
using System;

class Program
{
    static void Main()
    {
        string text = "Hello";
        char[] characters = text.ToCharArray();

        foreach (char c in characters)
        {
            Console.WriteLine(c);
        }
    }
}
```

Summary:

Text analysis and manipulation in C# provides a wide range of powerful functions that facilitate the handling of strings in various ways. With these functions, you can write programs that handle text in a flexible and efficient manner.

Control Flow / If Statement

Control flow, particularly using if statements, is a fundamental concept in programming that allows you to execute certain pieces of code based on conditions. Here's a basic overview and example in C#:

1. Basic if Statement

The if statement evaluates a condition, and if that condition is true, the code block inside the if statement is executed. If the condition is false, the code block is skipped.

Syntax:

```
if (condition)
{
    // Code to execute if condition is true
}
```

Example:

```
int number = 10;

if (number > 5)
{
    Console.WriteLine("The number is greater than 5.");
}
```

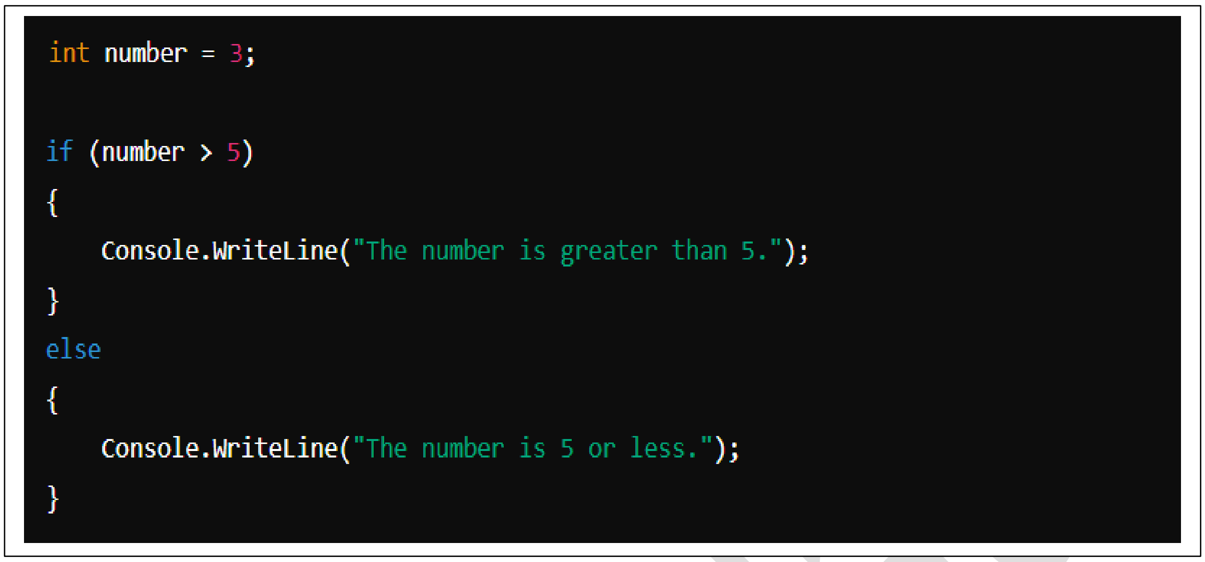
2. if-else Statement

Sometimes you want to execute one block of code if the condition is true, and another block if it's false.

Syntax:

```
if (condition)
{
    // Code to execute if condition is true
}
else
{
    // Code to execute if condition is false
}
```

Example:

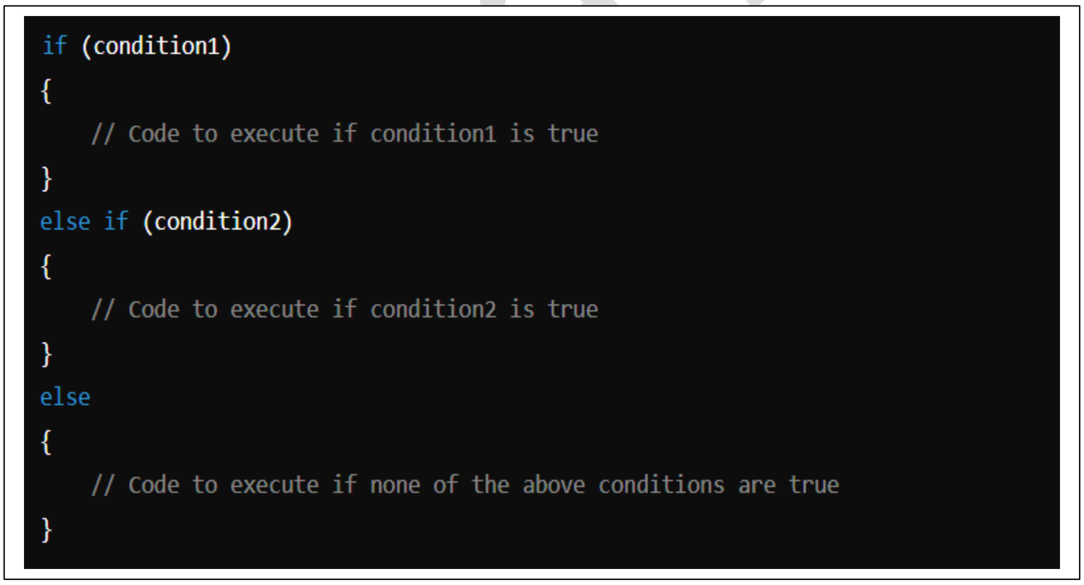


```csharp
int number = 3;

if (number > 5)
{
    Console.WriteLine("The number is greater than 5.");
}
else
{
    Console.WriteLine("The number is 5 or less.");
}
```

2. if-else if-else Statement

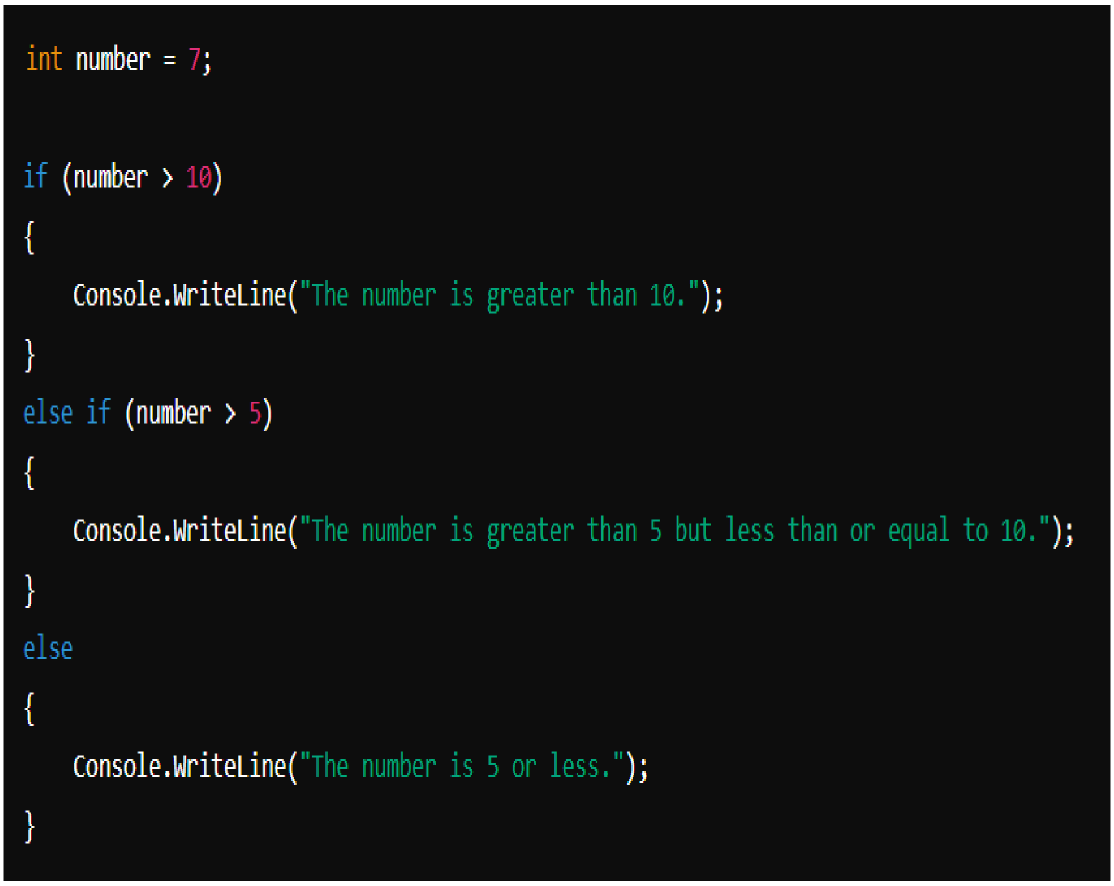
For multiple conditions, you can chain if and else statements together.

Syntax:



```csharp
if (condition1)
{
    // Code to execute if condition1 is true
}
else if (condition2)
{
    // Code to execute if condition2 is true
}
else
{
    // Code to execute if none of the above conditions are true
}
```

Example:

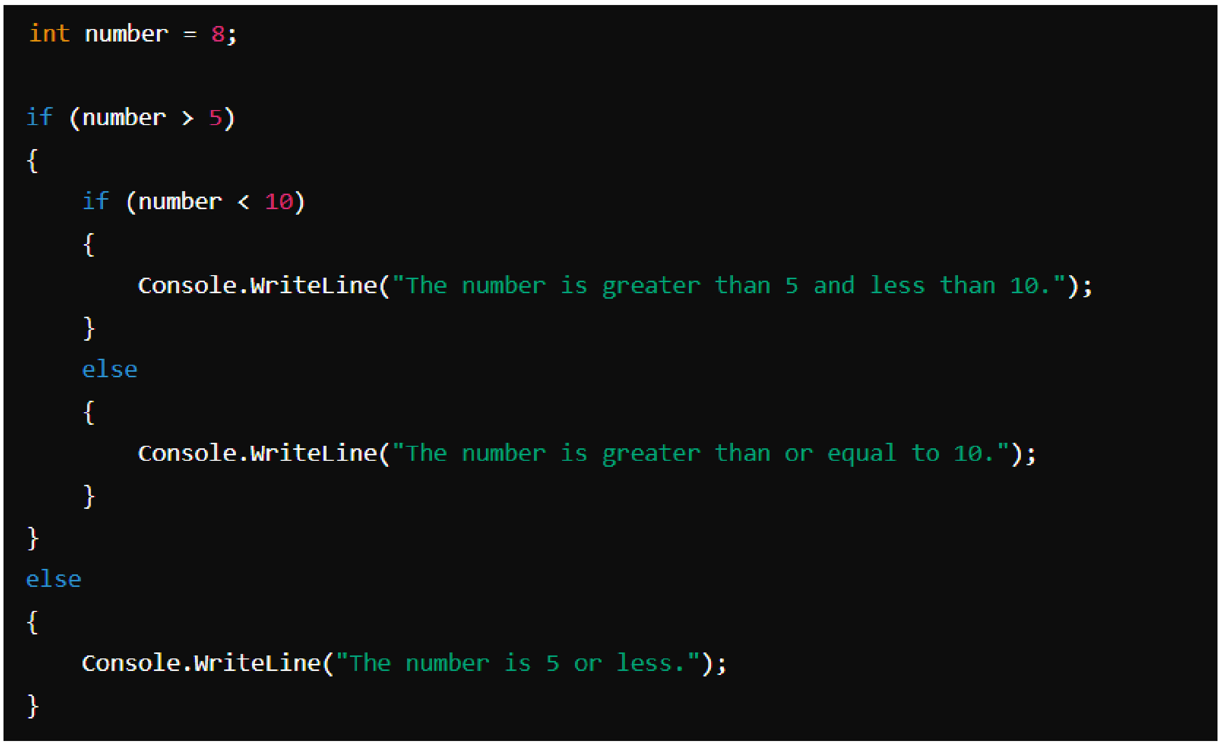


```
int number = 7;

if (number > 10)
{
    Console.WriteLine("The number is greater than 10.");
}
else if (number > 5)
{
    Console.WriteLine("The number is greater than 5 but less than or equal to 10.");
}
else
{
    Console.WriteLine("The number is 5 or less.");
}
```

4. Nested if Statements

You can also nest if statements within each other.

Example:



```
int number = 8;

if (number > 5)
{
    if (number < 10)
    {
        Console.WriteLine("The number is greater than 5 and less than 10.");
    }
    else
    {
        Console.WriteLine("The number is greater than or equal to 10.");
    }
}
else
{
    Console.WriteLine("The number is 5 or less.");
}
```

Task 4 :

If you'd like, you can try writing a simple C# program using if statements to check whether a number is positive, negative, or zero.

Debugger Basics

Debugging is an essential part of programming that helps you find and fix errors or bugs in your code. A debugger is a tool that allows you to execute your program step by step, inspect variables, and understand the flow of your program. Here's a basic overview of how to use a debugger in C#, typically within Visual Studio.

1. Setting Breakpoints

- A **breakpoint** is a marker that you set on a line of code. When the debugger runs your program, it will pause execution when it reaches a breakpoint, allowing you to inspect the current state of the program.
- To set a breakpoint, click on the left margin next to the line number in your code, or place the cursor on a line and press F9. A red dot will appear, indicating the breakpoint.

2. Starting the Debugger

- You can start debugging by pressing F5 or selecting **Start Debugging** from the Debug menu. The program will run normally until it hits a breakpoint.
- Alternatively, you can use Ctrl + F5 to start the program without debugging.

3. Stepping Through Code

- **Step Over (F10):** Executes the current line of code. If the line is a method call, it will execute the entire method without stepping into it.
- **Step Into (F11):** Steps into the method, allowing you to debug inside that method.
- **Step Out (Shift + F11):** If you're inside a method, this command will complete the method execution and return to the calling method.
- **Continue (F5):** Resumes the execution of the program until the next breakpoint or the end of the program.

4. Inspecting Variables

- When the debugger is paused (at a breakpoint), you can hover over variables to see their current values.
- The **Locals** window shows all local variables and their values at the current point in the execution.
- The **Watch** window allows you to add specific variables or expressions that you want to monitor closely.

5. Examining the Call Stack

- The **Call Stack** window shows the list of method calls that are currently active, allowing you to trace the flow of execution.
- This is particularly useful for understanding how you arrived at a particular point in the code.

6. Editing Code While Debugging

- Visual Studio allows you to edit your code while debugging (this is called **Edit and Continue**). After making changes, you can continue running the program without restarting the debugging session.

7. Conditional Breakpoints

- You can make a breakpoint conditional, so it only triggers when a certain condition is met. Right-click on a breakpoint and select **Conditions** to specify an expression that must be true for the breakpoint to be hit.

8. Using Immediate Window

- The **Immediate Window** allows you to execute commands or evaluate expressions at runtime. This is useful for testing changes or checking values on the fly.

9. Output Window

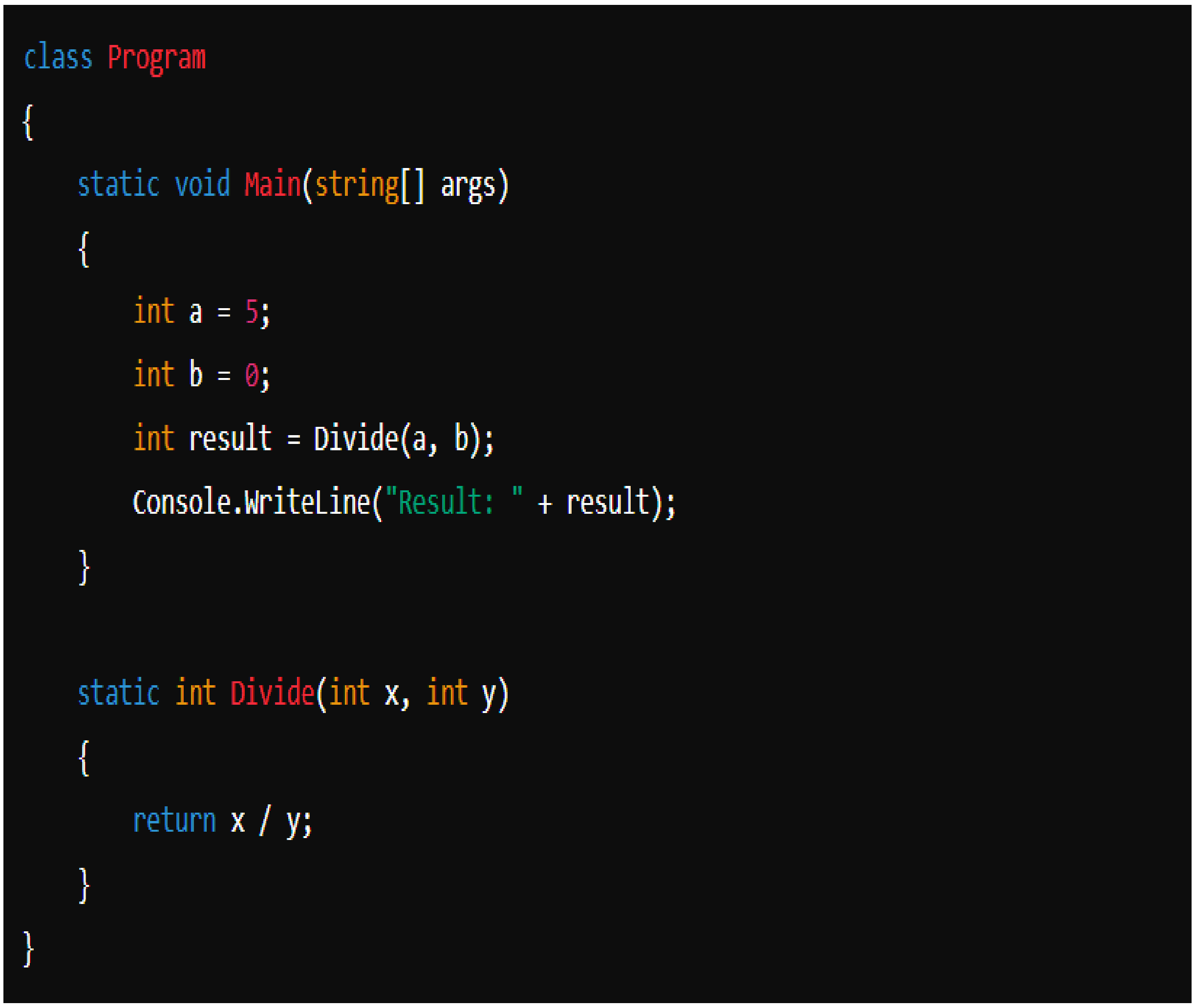
- The **Output Window** shows various debug messages, such as exceptions, output from Console.WriteLine, and more. It's helpful for understanding what's happening in your **program.**

10. Debugging Exceptions

- When an exception occurs, the debugger will pause execution at the line where the exception was thrown. You can inspect the exception details to understand what went wrong.
- You can configure the debugger to break on specific types of exceptions, even if they're caught in your code.

Example of a Simple Debugging Session

Consider the following simple C# code:



```
class Program
{
    static void Main(string[] args)
    {
        int a = 5;
        int b = 0;
        int result = Divide(a, b);
        Console.WriteLine("Result: " + result);
    }

    static int Divide(int x, int y)
    {
        return x / y;
    }
}
```

Steps to Debug:

1. **Set a Breakpoint:** Set a breakpoint on the line int result = Divide(a, b);.
2. **Start Debugging:** Press F5 to start the debugger.
3. **Step Into:** When the breakpoint is hit, press F11 to step into the Divide method.
4. **Inspect Variables:** Hover over x and y to see their values.
5. **Continue Execution:** Press F5 to let the program run. An exception will occur because of division by zero.
6. **Examine the Exception:** The debugger will pause at the exception, allowing you to inspect what went wrong.

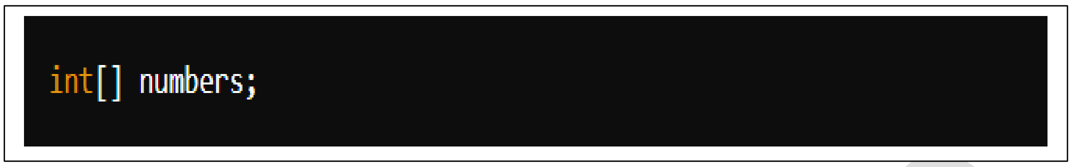
Arrays

Arrays in C# are a fundamental data structure that allow you to store a fixed-size collection of elements of the same type. They are useful for managing collections of data, like numbers, strings, or objects, in a structured way.

1.Declaring and Initializing Arrays

1. Declaring an Array

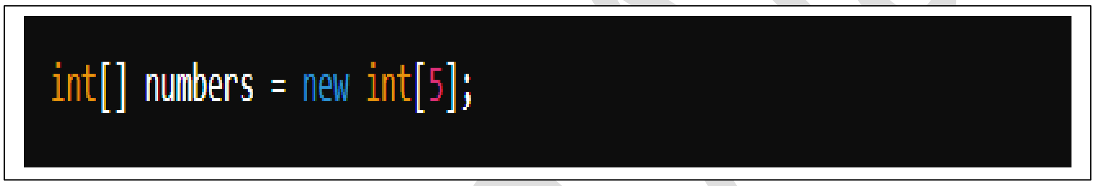
- To declare an array in C#, you specify the type of the elements followed by square brackets [].
- Example:



```
int[] numbers;
```

2. Initializing an Array

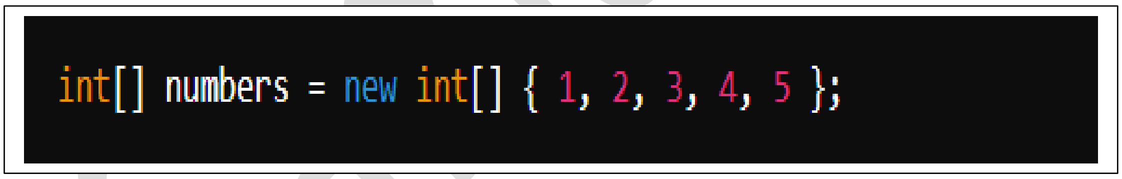
- You can initialize an array by specifying its size and optionally setting initial values.
- Example with size only:



```
int[] numbers = new int[5];
```

This creates an array of 5 integers, all initialized to 0.

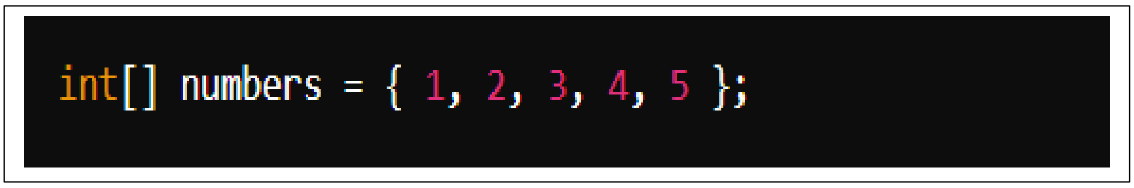
Example with initial values:



```
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
```

Here, the array is initialized with 5 elements.

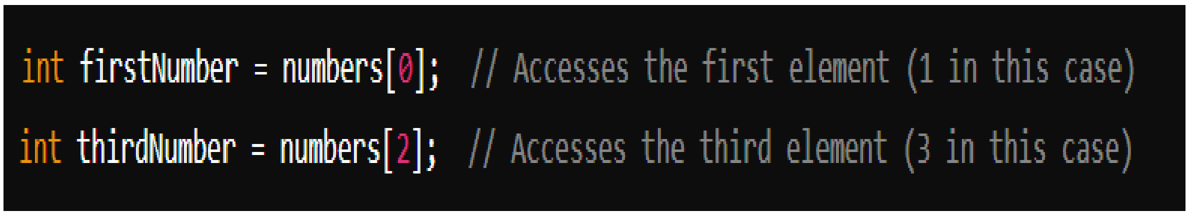
You can also omit the new int [] part when directly initializing with values:



```
int[] numbers = { 1, 2, 3, 4, 5 };
```

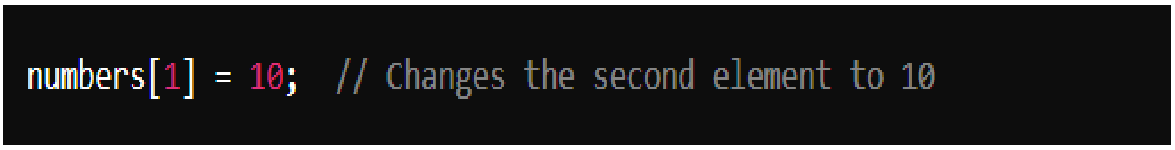
2.Accessing Array Elements

- Array elements are accessed using their index, which starts at 0.
- Example:



```
int firstNumber = numbers[0];  // Accesses the first element (1 in this case)
int thirdNumber = numbers[2];  // Accesses the third element (3 in this case)
```

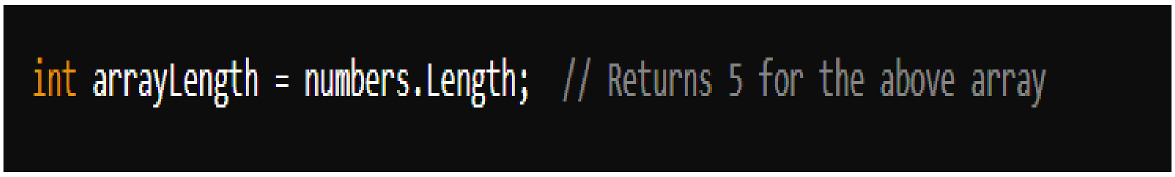
You can also modify elements by assigning new values:



```
numbers[1] = 10;  // Changes the second element to 10
```

3.Array Properties and Methods

- **Length:** The `Length` property gives the total number of elements in the array.



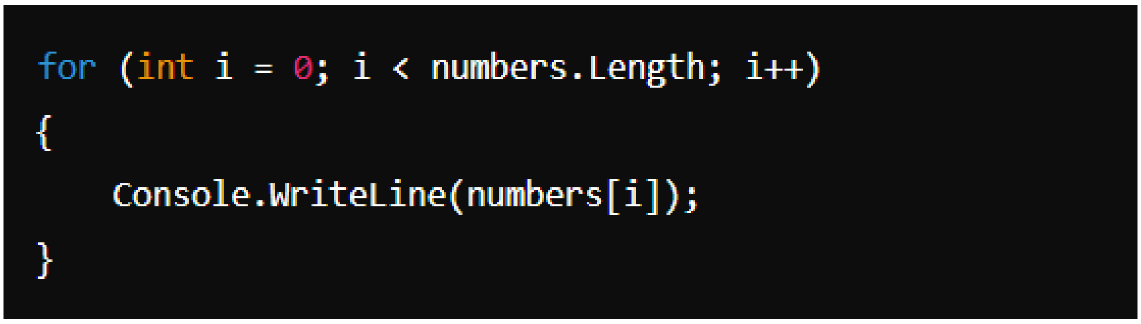
```
int arrayLength = numbers.Length;  // Returns 5 for the above array
```

- **IndexOutOfRangeException:** If you try to access an index that is outside the bounds of the array, an `IndexOutOfRangeException` will be thrown.

4.Looping Through Arrays

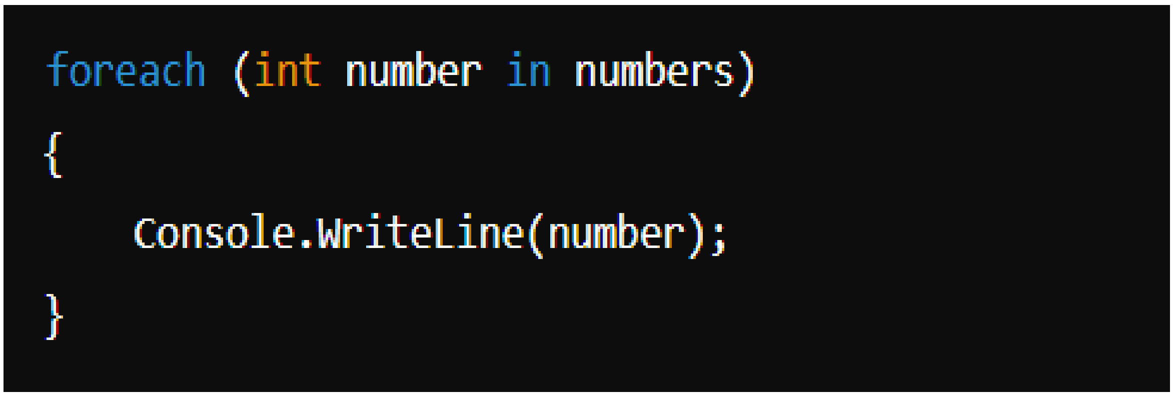
- Arrays are often looped through using `for` or `foreach` loops.

Using `for` Loop:



```
for (int i = 0; i < numbers.Length; i++)
{
    Console.WriteLine(numbers[i]);
}
```

Using foreach Loop:

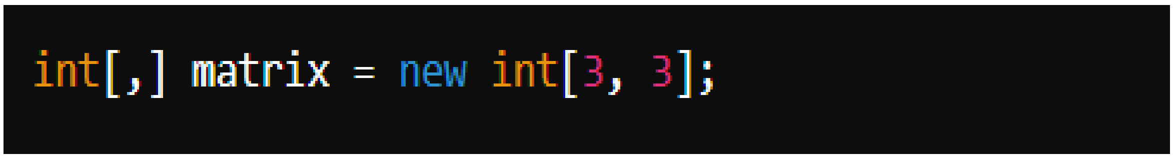


```
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

5.Multidimensional Arrays

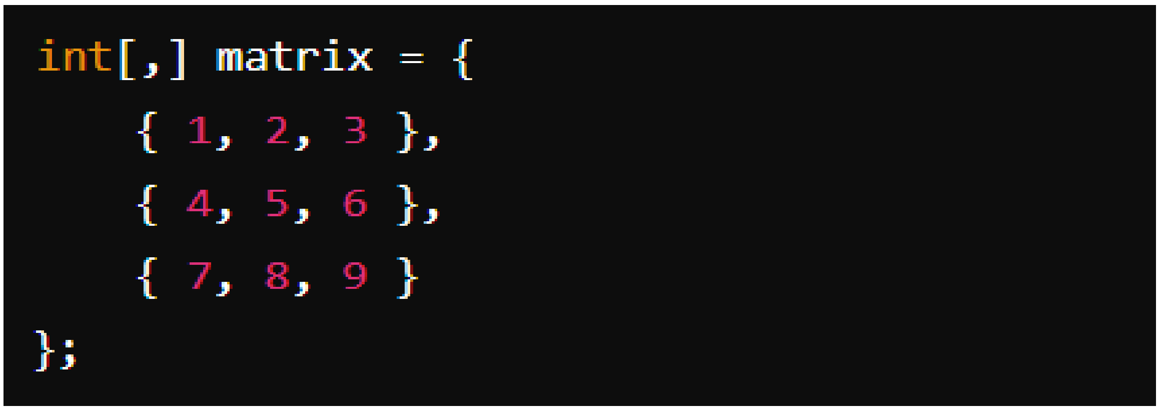
- C# supports multidimensional arrays, like 2D arrays (often used for matrices).

Declaring and Initializing a 2D Array:



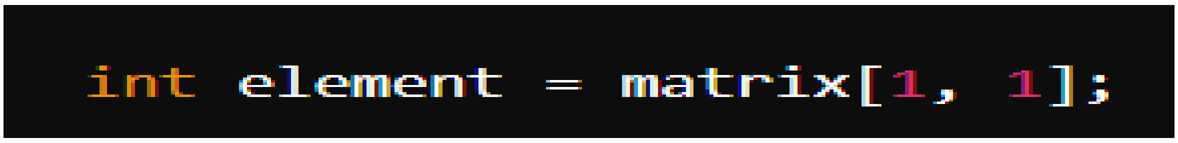
```
int[,] matrix = new int[3, 3];
```

- You can also initialize it with values:



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

Accessing Elements in a 2D Array:

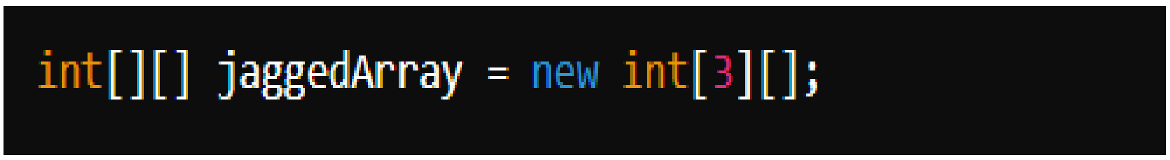


```
int element = matrix[1, 1];
```

6.Jagged Arrays

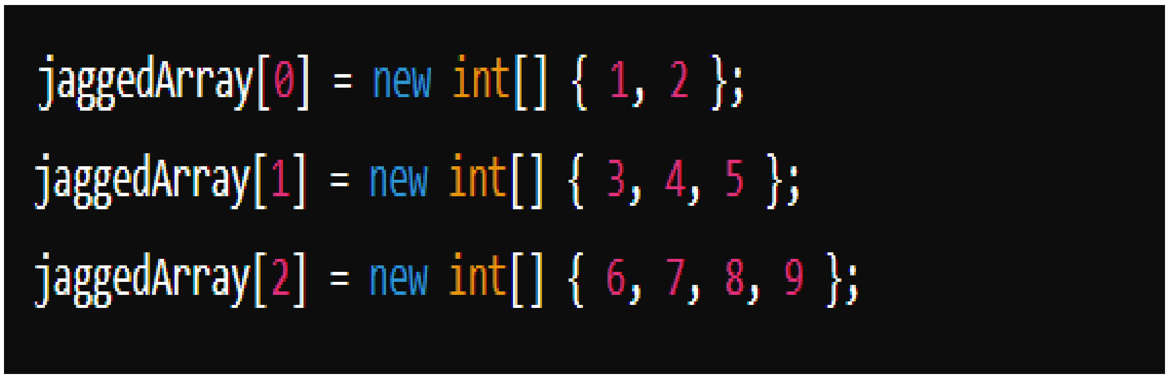
- A jagged array is an array of arrays, where each "sub-array" can have different lengths.

Declaring a Jagged Array:



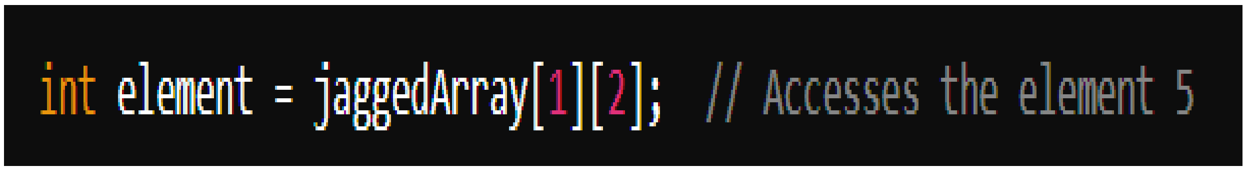
```
int[][] jaggedArray = new int[3][];
```

Initializing the Sub-Arrays:



```
jaggedArray[0] = new int[] { 1, 2 };
jaggedArray[1] = new int[] { 3, 4, 5 };
jaggedArray[2] = new int[] { 6, 7, 8, 9 };
```

Accessing Elements in a Jagged Array:

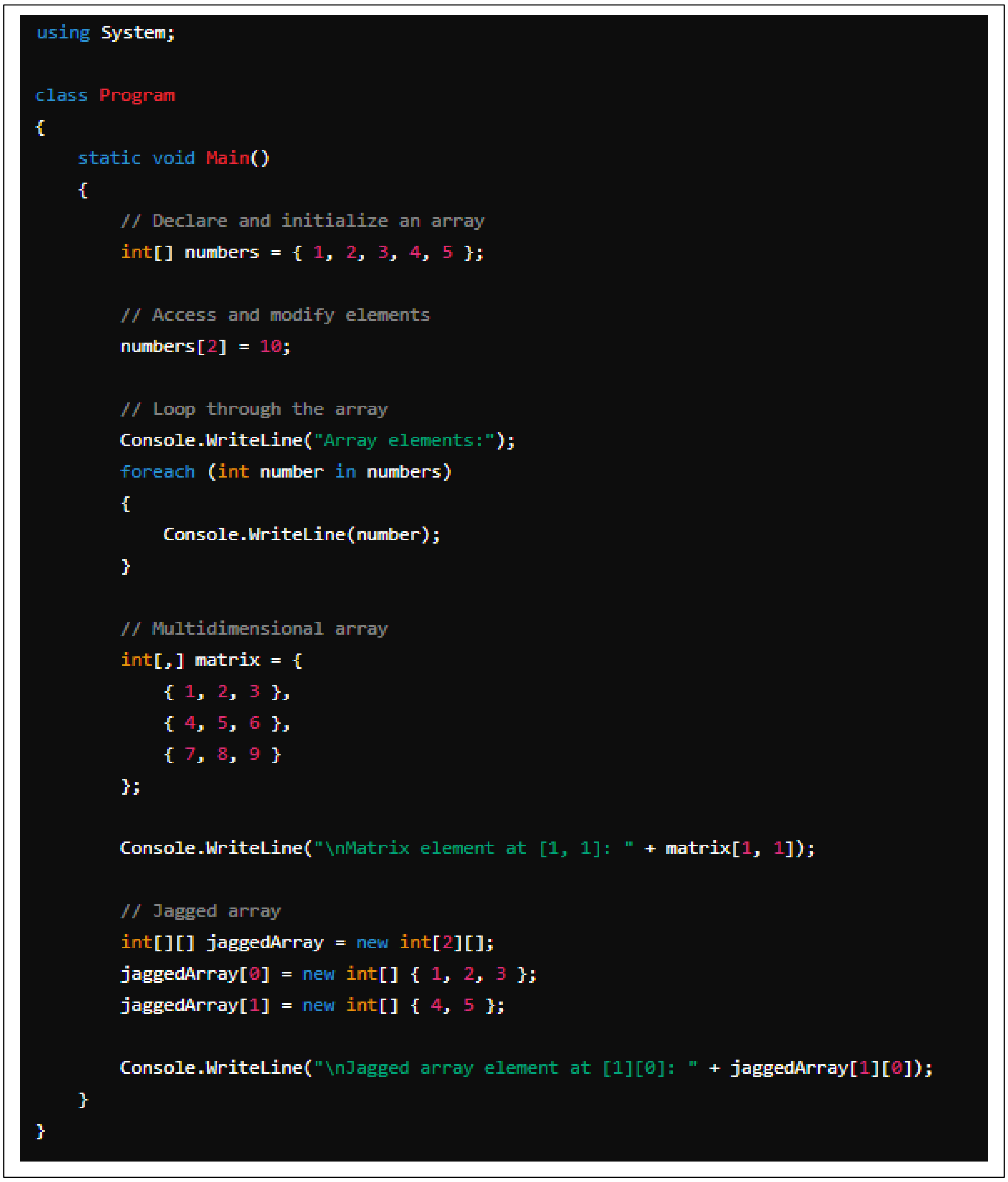


```
int element = jaggedArray[1][2];  // Accesses the element 5
```

Example Program:

Working with Arrays

Here's a simple C# program that demonstrates the use of arrays:



```
using System;

class Program
{
    static void Main()
    {
        // Declare and initialize an array
        int[] numbers = { 1, 2, 3, 4, 5 };

        // Access and modify elements
        numbers[2] = 10;

        // Loop through the array
        Console.WriteLine("Array elements:");
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }

        // Multidimensional array
        int[,] matrix = {
            { 1, 2, 3 },
            { 4, 5, 6 },
            { 7, 8, 9 }
        };

        Console.WriteLine("\nMatrix element at [1, 1]: " + matrix[1, 1]);

        // Jagged array
        int[][] jaggedArray = new int[2][];
        jaggedArray[0] = new int[] { 1, 2, 3 };
        jaggedArray[1] = new int[] { 4, 5 };

        Console.WriteLine("\nJagged array element at [1][0]: " + jaggedArray[1][0]);
    }
}
```

Key Points:

- Arrays in C# are zero-indexed.
- The size of an array is fixed once it is created.
- C# provides powerful features for working with arrays, including multidimensional and jagged arrays, to handle complex data structures.

OOP

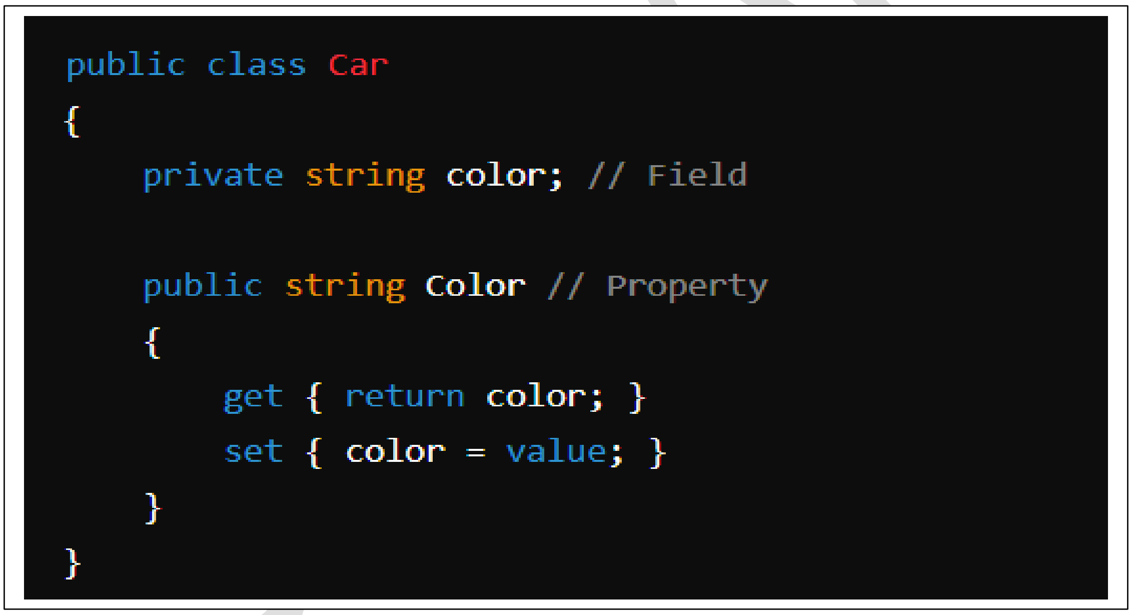
Object-Oriented Programming (OOP) is a programming paradigm that organizes code around "objects" which contain both data (fields) and methods (functions) that operate on that data. C# is a language that fully supports OOP.

Key Principles of OOP in C#:

There are four main principles in OOP:

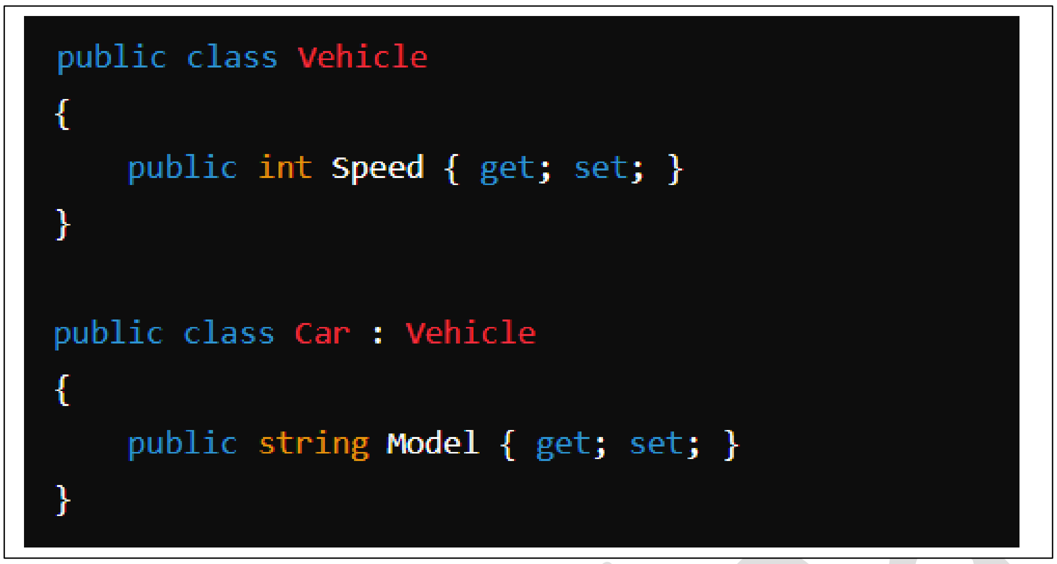
1. **Encapsulation:**

    - Encapsulation means bundling the data and methods that operate on that data into a single unit, known as an object. This helps protect the data from direct access from outside the object and controls how it is accessed or modified.
    - In C#, encapsulation is achieved using **fields** and **properties**, with access modifiers like **private** and **public** to control access levels.



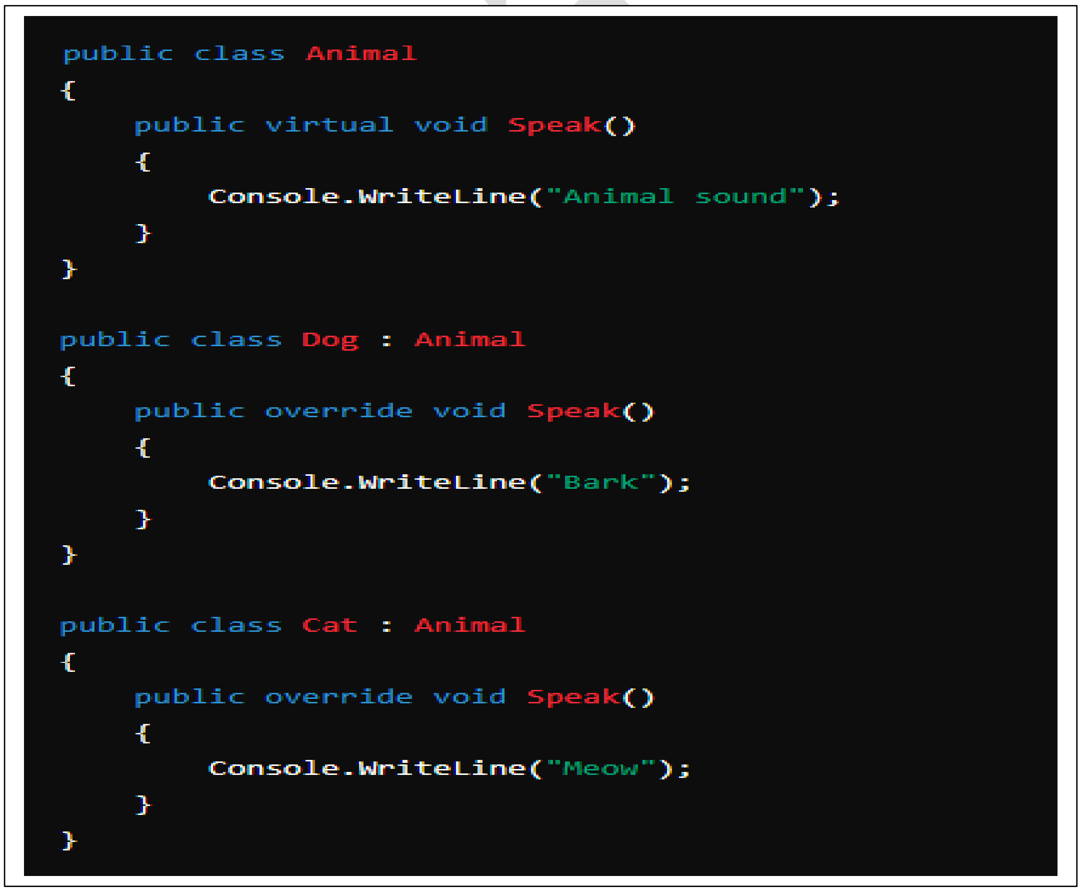
```
public class Car
{
    private string color; // Field

    public string Color // Property
    {
        get { return color; }
        set { color = value; }
    }
}
```

2. **Inheritance:**
    - Inheritance allows you to create a new class based on an existing class. The new class (derived class) inherits properties and methods from the existing class (base class), which enhances code reusability.
    - In C#, inheritance is implemented using the **:** syntax between the derived class name and the base class name.



```
public class Vehicle
{
    public int Speed { get; set; }
}

public class Car : Vehicle
{
    public string Model { get; set; }
}
```

3. **Polymorphism:**
   - Polymorphism allows objects of different classes to be treated as objects of a common base class. Methods in the base class can be overridden in derived classes, allowing for flexibility in how methods are implemented.
   - In C#, polymorphism is achieved using **virtual methods**, **method overriding**, and **interface implementation**.

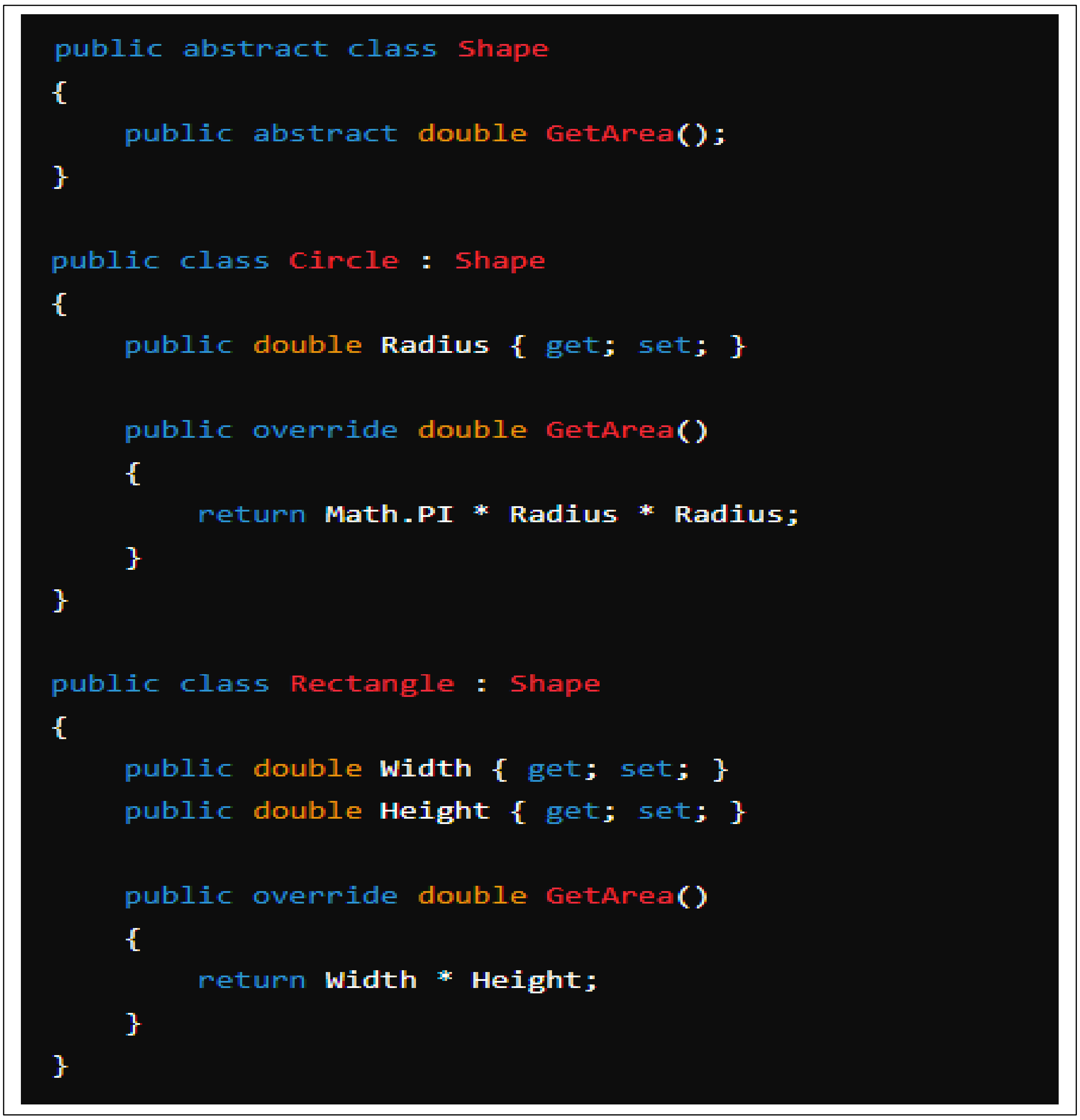


```
public class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal sound");
    }
}

public class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Bark");
    }
}

public class Cat : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Meow");
    }
}
```

- Here, you can use the `Speak()` method for both the `Dog` and `Cat` classes, even though the base class `Animal` doesn't know the actual type of the objects.

4. Abstraction:

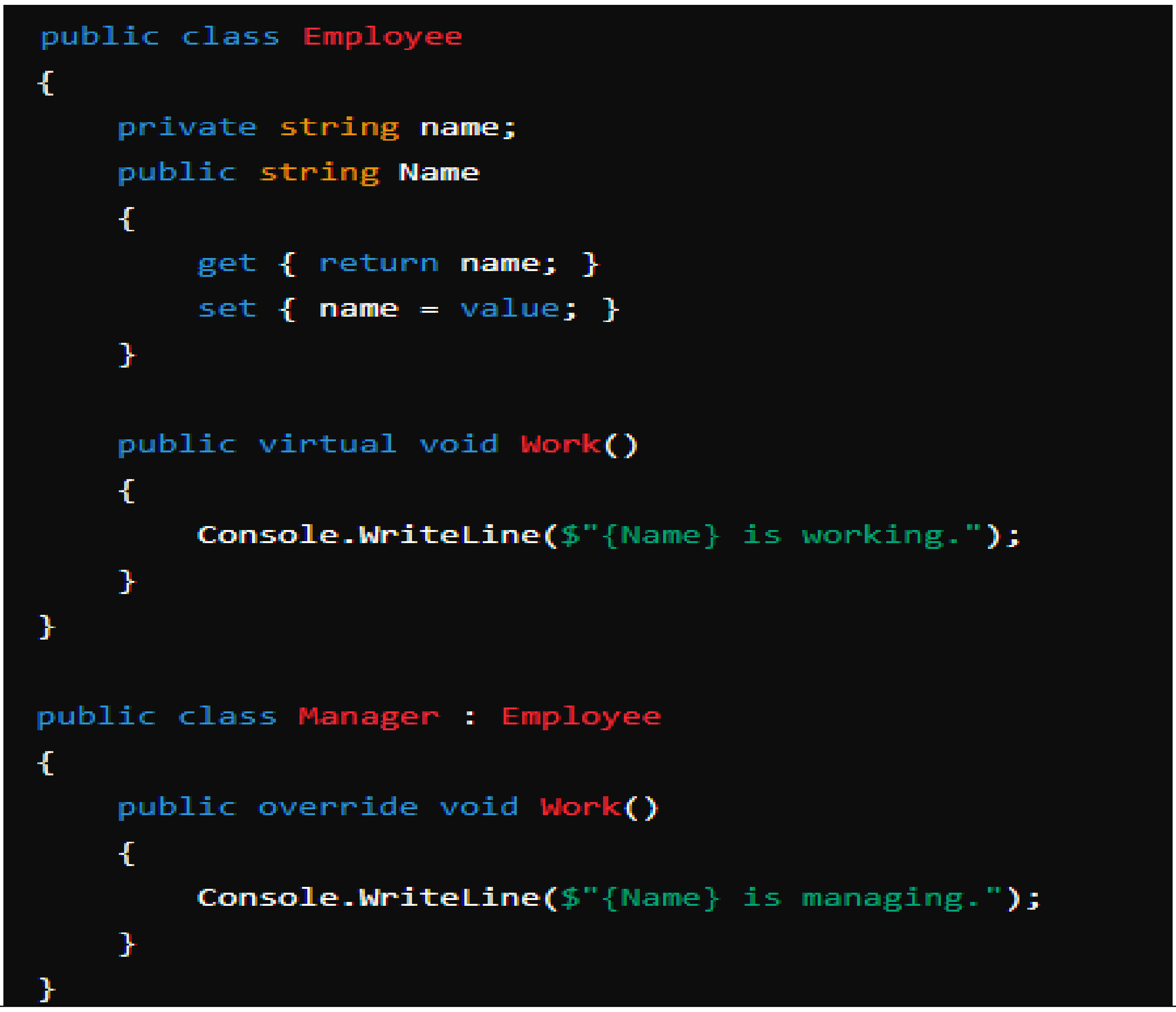
- Abstraction simplifies complexity by hiding unnecessary details and exposing only the essential features of an object. Abstraction defines "what an object does" without specifying "how it does it."
- In C#, abstraction is implemented using **abstract classes** and **interfaces**.



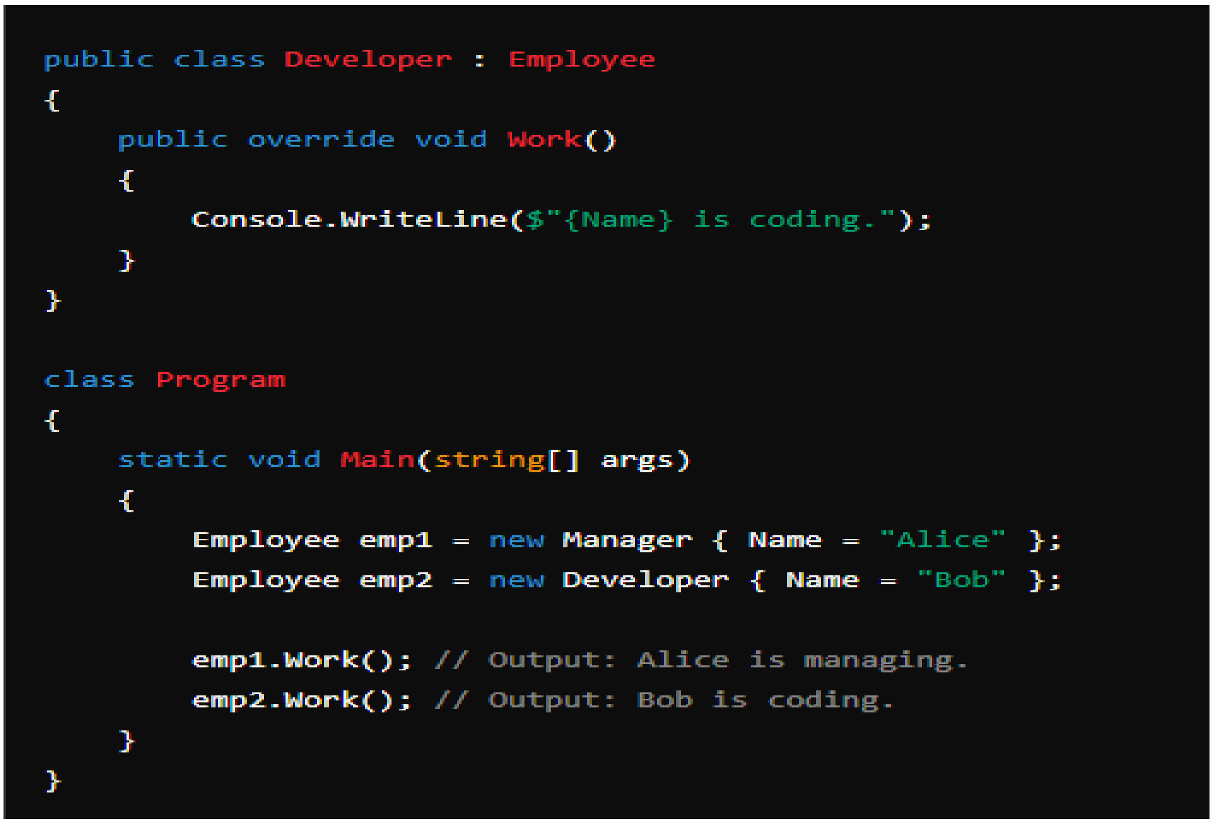
```
public abstract class Shape
{
    public abstract double GetArea();
}

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

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

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public override double GetArea()
    {
        return Width * Height;
    }
}
```

Here, the Shape class is abstract and contains an abstract method GetArea(). Derived classes like Circle and Rectangle must implement this method.

Putting It All Together:



```
public class Employee
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public virtual void Work()
    {
        Console.WriteLine($"{Name} is working.");
    }
}

public class Manager : Employee
{
    public override void Work()
    {
        Console.WriteLine($"{Name} is managing.");
    }
}
```



```
public class Developer : Employee
{
    public override void Work()
    {
        Console.WriteLine($"{Name} is coding.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Employee emp1 = new Manager { Name = "Alice" };
        Employee emp2 = new Developer { Name = "Bob" };

        emp1.Work(); // Output: Alice is managing.
        emp2.Work(); // Output: Bob is coding.
    }
}
```

Conclusion:

OOP in C# allows you to organize your code in a more logical and understandable way, making it easier to maintain and extend. By using encapsulation, inheritance, polymorphism, and abstraction, you can write flexible, reusable, and easy-to-maintain code.

Task 5:

Question 1:

The library

library is present in the simple email system. The system should contain the following categories: Book: contains information such as title (title), author (author), year (year), and book number (ISBN).

Library: contains a list of all books with functions of new book, remove book, and explanation of book using book number (ISBN).

Required: The class book exists.

Visit the class library and write the methods of deleting books.

Add a method to search for a book using book number (ISBN).

Write the codes of these categories.

Task 6:

Question 2: Managing Company Employees

Create a program to manage company employees. The program should contain the following classes:

Employee: Contains fields Name, ID, and Sal.

Manager: Inherits from Employee, and adds a field for Department.

Developer: Inherits from Employee, and adds a field for ProgrammingLanguage.

Required:

Create an Employee class with basic fields and methods.

Create Manager and Developer classes that inherit from Employee.

Add a DisplayInfo method in each class to display employee details.

Write code to create a list of employees containing managers and developers, and display their information using the DisplayInfo method.