Exploring C# Methods

From Basics to Advanced

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

What are Methods?

Definition and Purpose

Methods are fundamental building blocks in C# programming, functioning as reusable blocks of code that perform specific tasks. They encapsulate code into a single unit that can be called or invoked from different parts of a program, promoting organization and reducing redundancy.

Importance of Methods in Programming

Methods help to break down complex problems into smaller, more manageable pieces. This modular approach makes code easier to understand, test, and maintain. By defining methods, programmers can create a clear structure for their applications, facilitating better problem-solving and enhancing code readability.

Benefits of Using Methods

Code Reusability

One of the primary benefits of methods is code reusability. Once a method is written, it can be called multiple times from different parts of the program or even from other programs. This eliminates the need to write the same code repeatedly, saving time and reducing the potential for errors.

Modularity

Methods contribute to the modularity of a program. By dividing a program into separate methods, each responsible for a specific task, developers can focus on individual components without being overwhelmed by the complexity of the entire program. This modular design makes it easier to update and debug the code.

Ease of Maintenance

Maintaining a program becomes more straightforward when it is structured with methods. Since methods encapsulate specific functionality, changes to a method's implementation can be made without affecting the rest of the program. This isolation simplifies debugging and enhances the program's adaptability to future requirements or modifications.

Syntax of a Method

In C#, a method is defined using a specific syntax that includes the return type, method name, and optional parameters. Here is the basic structure:

```
returnType MethodName(parameterList)
{
    // Method body
}
```

- **returnType:** Specifies the type of value the method will return. If no value is returned, use `void`.

- **MethodName:** The name given to the method, following standard naming conventions.
- **parameterList:** A comma-separated list of parameters (if any) that the method takes. If there are no parameters, this can be left empty.

Anatomy of a Method

Method Signature

Includes the method's name and its parameter list.

Method Body

Enclosed in curly braces {}, this contains the code that defines what the method does.

Example

```
void Greet()
{
    Console.WriteLine("Hello, World!");
}
```

Calling Methods

How to Invoke a Method

To execute the code within a method, you call or invoke the method by using its name followed by parentheses. If the method takes parameters, you include them within the parentheses.

Syntax for Calling a Method

```
MethodName();
```

If the method has parameters, include the arguments within the parentheses:

```
MethodName(argument1, argument2);
```

Example

```
Greet(); // This calls the Greet method, which prints "Hello, World!"
```

Methods with No Parameters and No Return Types

Methods with no parameters and no return types are often referred to as void methods. These methods perform an action but do not require any input (parameters) from the caller and do not return any value.

Use Cases

- Setting up or initializing values at the start of a program or a certain process.
- Displaying information to the console or a user interface.
- Executing a series of statements where the result does not need to be returned or used elsewhere in the code.

Creating and Using Void Methods

The syntax for creating a void method in C# involves specifying the `void` keyword, followed by the method name and an empty parameter list. The method body is enclosed within curly braces {}.

```
void MethodName()
{
    // Method body
}
```

Examples

A Simple Method to Print a Message

```
void PrintWelcomeMessage()
{
    Console.WriteLine("Welcome to the C# programming tutorial!");
}
```

In this example, `PrintWelcomeMessage` is a method that prints a welcome message to the console. It does not take any parameters or return any value.

Method to Perform a Simple Operation

```
void DisplayCurrentTime()
{
    Console.WriteLine("The current time is: " + DateTime.Now.ToString("
   hh:mm:ss tt"));
}
```

This method, `DisplayCurrentTime`, displays the current time. It performs an operation but does not require any input or return any value.

Practical Examples

Defining and Calling a Void Method

```
void ShowInstructions()
{
    Console.WriteLine("1. Enter your name.");
    Console.WriteLine("2. Enter your age.");
    Console.WriteLine("3. Press Enter to submit.");
}
```

```
// Calling the method
ShowInstructions();
```

Here, `ShowInstructions` method provides instructions to the user. The method is called using its name followed by parentheses.

Void Method for Simple Calculation

```
void CalculateSum()
{
    int number1 = 5;
    int number2 = 10;
    int sum = number1 + number2;
    Console.WriteLine("The sum is: " + sum);
}

// Calling the method
CalculateSum();
```

The `CalculateSum` method performs a simple calculation of adding two numbers and prints the result. No parameters are required, and it does not return any value.

Using Void Method for Initialization

```
void InitializeSettings()
{
    Console.WriteLine("Initializing settings...");
    // Code to initialize settings
    Console.WriteLine("Settings initialized.");
}

// Calling the method
InitializeSettings();
```

In this example, `InitializeSettings` simulates the initialization of settings. It prints messages before and after the initialization process.

Methods with No Parameters and Return Types

What are Return Types?

Return types specify the type of value a method will return to the caller after it has completed its execution. They are an essential aspect of method definitions, enabling methods to produce and provide results that can be used elsewhere in the program.

Common Return Types in C#

- **Primitive Types:** `int`, `float`, `double`, `char`, `bool`
- **String Type:** `string`
- **Complex Types:** Arrays, Lists, Custom Objects (Note: Custom objects will be covered later)

Creating Methods with Return Types

The syntax for creating a method with a return type involves specifying the return type before the method name. The method must include a return statement within its body to return a value of the specified type.

```
returnType MethodName()
{
    // Method body
    return value;
}
```

- **returnType:** Specifies the type of value the method will return (e.g., `int`, `string`).
- **MethodName:** The name of the method.
- **return value:** The value returned by the method, which must match the specified return type.

Examples

Method Returning an Integer

```
int GetNumber()
{
    return 42;
}
```

This method, `GetNumber`, returns an integer value of 42.

Method Returning a String

```
string GetGreeting()
{
    return "Hello, World!";
}
```

The `GetGreeting` method returns a string with the message "Hello, World!".

Practical Examples

Defining and Calling a Method that Returns an Integer

```
int CalculateSum()
{
    int num1 = 10;
    int num2 = 15;
    return num1 + num2;
}

// Calling the method and using the returned value
int result = CalculateSum();
Console.WriteLine("The sum is: " + result);
```

In this example, the `CalculateSum` method returns the sum of two integers. The returned value is stored in a variable and printed to the console.

Defining and Calling a Method that Returns a String

```
string GetWelcomeMessage()
{
    return "Welcome to C# programming!";
}

// Calling the method and using the returned value
string message = GetWelcomeMessage();
Console.WriteLine(message);
```

The `GetWelcomeMessage` method returns a welcome message. The returned string is stored in a variable and printed to the console.

Method Returning a Boolean Value

```
bool IsEven(int number)
{
    return number % 2 == 0;
}

// Calling the method and using the returned value
bool isEven = IsEven(4);
Console.WriteLine("Is the number even? " + isEven);
```

Here, the `IsEven` method checks if a given number is even and returns a boolean value (`true` or `false`). The returned value is used to determine if the number is even and is printed to the console.

Methods with Parameters and No Return Types

Defining Parameters

What are Parameters?

Parameters are variables that are passed into methods to provide the method with input data. They allow methods to perform tasks with different input values, making the methods more flexible and reusable.

Types of Parameters

- **Value Parameters:** These are the most common type of parameters, where the value is passed to the method. The method works with a copy of the value, meaning changes to the parameter within the method do not affect the original value.
- **Reference Parameters:** Although we won't go into `ref` or `out` parameters here, it's worth noting that there are ways to pass parameters by reference, allowing the method to modify the original value.

Creating Methods with Parameters

Syntax

To create a method with parameters, include the parameter list within the parentheses in the method declaration. Each parameter must have a type and a name.

```
void MethodName(parameterType parameterName)
{
    // Method body
}
```

- **parameterType:** The data type of the parameter (e.g., `int`, `string`).
- **parameterName:** The name of the parameter used within the method.

Examples

Method with One Parameter

```
void PrintNumber(int number)
{
    Console.WriteLine("The number is: " + number);
}
```

In this example, `PrintNumber` takes an integer parameter and prints it.

Method with Multiple Parameters

```
void PrintSum(int num1, int num2)
{
    int sum = num1 + num2;
    Console.WriteLine("The sum is: " + sum);
}
```

The `PrintSum` method takes two integer parameters, calculates their sum, and prints the result.

Practical Examples

Defining and Calling a Method with One Parameter

```
void GreetUser(string name)
{
    Console.WriteLine("Hello, " + name + "!");
}

// Calling the method with a parameter
GreetUser("Alice");
```

In this example, the `GreetUser` method takes a string parameter called `name` and prints a personalized greeting. The method is called with the argument "Alice", resulting in the output: "Hello, Alice!".

Method with Multiple Parameters

```
void MultiplyNumbers(int a, int b)
{
    int product = a * b;
    Console.WriteLine("The product is: " + product);
}

// Calling the method with parameters
MultiplyNumbers(3, 4);
```

Here, the `MultiplyNumbers` method takes two integer parameters, multiplies them, and prints the product. Calling the method with 3 and 4 results in the output: "The product is: 12".

Using Parameters to Control Method Behavior

```
void DisplayMessage(string message, int times)
{
    for (int i = 0; i < times; i++)
    {
        Console.WriteLine(message);
    }
}

// Calling the method with parameters
DisplayMessage("Welcome to the course!", 3);
```

The `DisplayMessage` method takes a string parameter `message` and an int parameter `times`. It prints the message the specified number of times. Calling the method with "Welcome to the course!" and 3 prints the message three times.

Methods with Parameters and Return Types

Combining Parameters and Return Types

Methods with both parameters and return types are designed to perform operations based on input values and return a result. Parameters allow you to pass values into the method, and the return type specifies what type of value the method will send back to the caller.

Defining Such Methods

When defining such methods, you need to:

- Specify the return type.
- Define the method name.
- List the parameters within parentheses, separating multiple parameters with commas.
- Include a return statement in the method body that returns a value matching the specified return type.

Syntax

The syntax for creating a method with parameters and a return type involves declaring the return type, the method name, and the parameters, followed by the method body containing the logic and the return statement.

```
returnType MethodName(parameterType1 parameterName1, parameterType2
    parameterName2)
{
    // Method body
    return value;
}
```

- **returnType:** The type of value the method will return (e.g., `int`, `string`).
- **MethodName:** The name of the method.
- **parameterType:** The type of the parameter (e.g., `int`, `string`).
- **parameterName:** The name of the parameter.
- **return value:** The value returned by the method, which must match the specified return type.

Examples

Method with Integer Parameters and Return Type

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

This method, `AddNumbers`, takes two integer parameters and returns their sum.

Method with String Parameters and Return Type

```
string ConcatenateStrings(string str1, string str2)
{
    return str1 + " " + str2;
}
```

The `ConcatenateStrings` method takes two string parameters and returns their concatenation.

Practical Examples

Defining and Calling a Method that Adds Two Numbers

```
int MultiplyNumbers(int x, int y)
{
    return x * y;
}
```

```
// Calling the method and using the returned value
int product = MultiplyNumbers(3, 4);
Console.WriteLine("The product is: " + product);
```

In this example, the `MultiplyNumbers` method takes two integer parameters, multiplies them, and returns the product. The returned value is stored in a variable and printed to the console.

Defining and Calling a Method that Returns a Formatted String

```
string FormatName(string firstName, string lastName)
{
    return "Full Name: " + firstName + " " + lastName;
}

// Calling the method and using the returned value
string formattedName = FormatName("John", "Doe");
Console.WriteLine(formattedName);
```

The `FormatName` method takes two string parameters, formats them into a full name, and returns the result. The returned string is stored in a variable and printed to the console.

Method with Boolean Return Type and Integer Parameters

```
bool IsGreater(int num1, int num2)
{
    return num1 > num2;
}

// Calling the method and using the returned value
bool result = IsGreater(10, 5);
Console.WriteLine("Is the first number greater? " + result);
```

In this example, the `IsGreater` method takes two integer parameters, compares them, and returns a boolean value indicating whether the first number is greater than the second. The returned value is used to determine and print the result.

Understanding Optional Parameters

Optional parameters in C# are parameters that are not required to be provided by the caller of the method. If the caller does not provide a value for an optional parameter, the method uses a predefined default value. This feature allows for more flexible method calls and can simplify method overloading by reducing the need to define multiple methods with different parameter sets.

Benefits

- Reduces the number of method overloads by allowing default values.
- Provides flexibility to callers, enabling them to specify only the parameters that are relevant to their specific needs.
- Makes the method signatures clearer and easier to understand.

How to Set Default Values

To define an optional parameter, you assign a default value to the parameter in the method signature. If no argument is passed for that parameter when the method is called, the default value is used.

Syntax

```
void MethodName(parameterType parameterName = defaultValue)
{
    // Method body
}
```

- **parameterType:** The type of the parameter (e.g., `int`, `string`).
- **parameterName:** The name of the parameter.
- **defaultValue:** The value assigned to the parameter if no argument is provided.

Examples

```
void DisplayNumber(int number = 10)
{
    Console.WriteLine("The number is: " + number);
}

void Greet(string name = "Guest")
{
    Console.WriteLine("Hello, " + name + "!");
}
```

Creating Methods with Optional Parameters

When defining a method with optional parameters, specify the default values in the parameter list. Optional parameters must come after all required parameters.

Syntax

```
void MethodName(parameterType1 requiredParameter, parameterType2
   optionalParameter = defaultValue)
{
    // Method body
}
```

Examples

```
void PrintMessage(string message, int repeatCount = 1)
{
    for (int i = 0; i < repeatCount; i++)
    {
        Console.WriteLine(message);
```

```
    }
}

void DisplayInfo(string name, int age = 0, string country = "Unknown")
{
    Console.WriteLine("Name: " + name);
    Console.WriteLine("Age: " + age);
    Console.WriteLine("Country: " + country);
}
```

Practical Examples

Defining and Calling a Method with an Optional Parameter

```
void ShowAlert(string message, bool isUrgent = false)
{
    if (isUrgent)
    {
        Console.WriteLine("URGENT: " + message);
    }
    else
    {
        Console.WriteLine("Alert: " + message);
    }
}

// Calling the method with both parameters
ShowAlert("System maintenance at midnight", true);

// Calling the method with only the required parameter
ShowAlert("System maintenance at midnight");
```

In this example, the `ShowAlert` method has an optional parameter `isUrgent` with a default value of `false`. The method behavior changes based on whether the `isUrgent` parameter is provided.

Method with Multiple Optional Parameters

```
void SendEmail(string recipient, string subject = "No Subject", string
    body = "No Content")
{
    Console.WriteLine("Sending Email:");
    Console.WriteLine("To: " + recipient);
    Console.WriteLine("Subject: " + subject);
    Console.WriteLine("Body: " + body);
}

// Calling the method with all parameters
SendEmail("example@example.com", "Meeting Reminder", "Don't forget the
    meeting at 3 PM");

// Calling the method with one optional parameter
SendEmail("example@example.com", "Greetings");

// Calling the method with only the required parameter
```

```
SendEmail("example@example.com");
```

The `SendEmail` method demonstrates multiple optional parameters. Depending on the provided arguments, it adjusts the subject and body of the email.

Using Out Parameters and Ref Keyword

Out Parameters

Out parameters in C# allow methods to return multiple values. They are useful when a method needs to output more than one value but cannot do so directly with the return statement alone.

Use Cases

- Returning multiple values from a method.
- Methods that need to modify multiple values or variables.
- Situations where you need to pass an uninitialized variable to a method and have it initialized within the method.

Syntax

To use out parameters, you must specify the `out` keyword in both the method definition and the method call.

```
void MethodName(out dataType parameterName)
{
    // Method body
    parameterName = value; // Assigning a value to the out parameter
}
```

Examples

```
void GetCoordinates(out int x, out int y)
{
    x = 10; // Assigning a value to x
    y = 20; // Assigning a value to y
}
```

In this example, `GetCoordinates` assigns values to the out parameters `x` and `y`.

Ref Keyword

The `ref` keyword is used to pass arguments by reference, allowing the method to modify the value of the passed argument. Unlike out parameters, `ref` parameters need to be initialized before being passed to a method.

Use Cases

- When you need a method to modify the value of an argument.
- When the initial value of the variable is required within the method.
- Useful for performance optimization by avoiding copying large objects.

Syntax

To use `ref` parameters, you must specify the `ref` keyword in both the method definition and the method call.

```
void MethodName(ref dataType parameterName)
{
    // Method body
    parameterName = newValue; // Modifying the value of the ref
   parameter
}
```

Examples

```
void DoubleValue(ref int number)
{
    number *= 2; // Doubling the value of the ref parameter
}
```

In this example, `DoubleValue` modifies the value of the `number` parameter by doubling it.

Practical Examples

Using Out Parameters

```
void Divide(int dividend, int divisor, out int quotient, out int
   remainder)
{
    quotient = dividend / divisor;
    remainder = dividend % divisor;
}

// Calling the method
int q, r;
Divide(10, 3, out q, out r);
Console.WriteLine("Quotient: " + q); // Output: Quotient: 3
Console.WriteLine("Remainder: " + r); // Output: Remainder: 1
```

In this example, the `Divide` method calculates the quotient and remainder of a division operation and returns them using out parameters.

Using Ref Keyword

```
void Increment(ref int number)
{
    number += 1; // Incrementing the value of the ref parameter
}

// Calling the method
int value = 5;
Increment(ref value);
Console.WriteLine("Incremented Value: " + value); // Output:
   Incremented Value: 6
```

In this example, the `Increment` method increases the value of the `number` parameter by 1.

Combining Out and Ref Parameters

```
void ProcessValues(int input, ref int refValue, out int outValue)
{
    refValue *= 2; // Modifying the ref parameter
    outValue = input + refValue; // Assigning a value to the out
   parameter
}

// Calling the method
int initialValue = 3;
int result;
ProcessValues(4, ref initialValue, out result);
Console.WriteLine("Ref Value: " + initialValue); // Output: Ref Value:
   6
Console.WriteLine("Out Value: " + result); // Output: Out Value: 10
```

In this example, `ProcessValues` uses both `ref` and `out` parameters to modify and return multiple values.

Overloading Methods

Definition

Method overloading is a feature in C# that allows multiple methods in the same class to have the same name but different parameters. These differences can be in the number of parameters, the types of parameters, or the order of parameters.

Benefits

- Methods with the same name can perform similar but slightly different tasks, making the code easier to read and understand.
- It allows for more flexible method calls, accommodating different input scenarios without needing unique method names.
- Overloaded methods can share the same logic but operate on different data types or numbers of inputs, reducing code duplication.

Creating Overloaded Methods

To create overloaded methods, define multiple methods with the same name but different parameter lists. The method signature (name and parameter types) must be unique for each overloaded method.

Syntax

```
void MethodName()
{
    // Method body
}

void MethodName(int parameter1)
{
    // Method body
}

void MethodName(int parameter1, string parameter2)
{
    // Method body
}
```

Examples

```
void DisplayMessage()
{
    Console.WriteLine("Hello, World!");
}

void DisplayMessage(string message)
{
    Console.WriteLine(message);
}

void DisplayMessage(string message, int count)
{
    for (int i = 0; i < count; i++)
    {
        Console.WriteLine(message);
    }
}
```

In this example, `DisplayMessage` is overloaded three times:

- The first method has no parameters and prints a default message.
- The second method takes a single string parameter and prints the provided message.
- The third method takes a string and an integer, printing the message multiple times.

Practical Examples

Overloaded Methods for Addition

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

double Add(double a, double b)
{
    return a + b;
}

int Add(int a, int b, int c)
{
    return a + b + c;
}

// Calling the methods
int sum1 = Add(5, 10);          // Uses the first method
double sum2 = Add(5.5, 10.5);   // Uses the second method
int sum3 = Add(1, 2, 3);        // Uses the third method

Console.WriteLine("Sum1: " + sum1);
Console.WriteLine("Sum2: " + sum2);
Console.WriteLine("Sum3: " + sum3);
```

In this example:

- The first `Add` method takes two integers and returns their sum.
- The second `Add` method takes two doubles and returns their sum.
- The third `Add` method takes three integers and returns their sum.

Overloaded Methods for Area Calculation

```
double CalculateArea(double radius)
{
    return Math.PI * radius * radius; // Area of a circle
}

double CalculateArea(double length, double width)
{
    return length * width; // Area of a rectangle
}

// Calling the methods
double circleArea = CalculateArea(5.0);             // Uses the first
   method
double rectangleArea = CalculateArea(4.0, 6.0);     // Uses the second
   method

Console.WriteLine("Circle Area: " + circleArea);
Console.WriteLine("Rectangle Area: " + rectangleArea);
```

In this example:

- The first `CalculateArea` method calculates the area of a circle given its radius.

- The second `CalculateArea` method calculates the area of a rectangle given its length and width.

Overloaded Methods for Logging Messages

```
void LogMessage(string message)
{
    Console.WriteLine("INFO: " + message);
}

void LogMessage(string message, string level)
{
    Console.WriteLine(level.ToUpper() + ": " + message);
}

void LogMessage(string message, string level, DateTime timestamp)
{
    Console.WriteLine("[" + timestamp.ToString("yyyy-MM-dd HH:mm:ss") +
    "] " + level.ToUpper() + ": " + message);
}

// Calling the methods
LogMessage("System started");                          // Uses the first
    method
LogMessage("User login failed", "Warning");            // Uses the
   second method
LogMessage("System error", "Error", DateTime.Now);     // Uses the third
    method
```

In this example:

- The first `LogMessage` method logs an informational message.
- The second `LogMessage` method logs a message with a specified severity level.
- The third `LogMessage` method logs a message with a severity level and a timestamp.

Understanding Recursion

Definition

Recursion is a programming technique where a method calls itself in order to solve a problem. It involves breaking down a problem into smaller sub-problems of the same type. A recursive method typically has two main components:

- **Base Case:** A condition that stops the recursion to prevent infinite loops.
- **Recursive Case:** The part of the method that includes the recursive call to the method itself.

Common Use Cases

Recursion is particularly useful in scenarios where a problem can be divided into smaller, similar problems, and can be solved more naturally using a recursive approach. Common use cases include:

- Mathematical computations: Such as calculating factorials, Fibonacci series, etc.
- Tree and graph traversal: Navigating hierarchical data structures like binary trees or parsing complex data.
- Divide and conquer algorithms: Examples include quicksort and mergesort.

Creating Recursive Methods

The syntax for creating a recursive method is similar to any other method. However, within the method body, there must be a condition to call the method itself.

Syntax

```
returnType MethodName(parameters)
{
    if (base condition)
    {
        // Base case
        return base value;
    }
    else
    {
        // Recursive case
        return MethodName(modified parameters);
    }
}
```

Examples

```
int Factorial(int n)
{
    if (n <= 1)  // Base case
    {
        return 1;
    }
    else
    {
        return n * Factorial(n - 1);  // Recursive call
    }
}

int Fibonacci(int n)
{
    if (n <= 1)  // Base case
    {
        return n;
    }
    else
    {
        return Fibonacci(n - 1) + Fibonacci(n - 2);  // Recursive calls
    }
}
```

Practical Examples

Calculating the Greatest Common Divisor (GCD)

The GCD of two integers is the largest integer that divides both numbers without leaving a remainder. This can be calculated using the Euclidean algorithm.

```
int GCD(int a, int b)
{
    if (b == 0)  // Base case: when second number becomes 0
    {
        return a;
    }
    else
    {
        return GCD(b, a % b);  // Recursive call: GCD of b and
   remainder of a divided by b
    }
}

// Calling the method and using the returned value
int result = GCD(48, 18);
Console.WriteLine("GCD of 48 and 18 is: " + result);  // Output: 6
```

Binary Search

Binary search is a method for finding an element in a sorted array by repeatedly dividing the search interval in half.

```
int BinarySearch(int[] array, int target, int left, int right)
{
    if (left > right)  // Base case: target is not found
    {
        return -1;
    }

    int mid = (left + right) / 2;

    if (array[mid] == target)  // Base case: target is found
    {
        return mid;
    }
    else if (array[mid] > target)
    {
        return BinarySearch(array, target, left, mid - 1);  //
   Recursive call: search in the left half
    }
    else
    {
        return BinarySearch(array, target, mid + 1, right);  //
   Recursive call: search in the right half
    }
}

// Calling the method and using the returned value
int[] numbers = {1, 3, 5, 7, 9, 11, 13};
int target = 7;
int result = BinarySearch(numbers, target, 0, numbers.Length - 1);
```

```
Console.WriteLine("Index of 7 is: " + result);  // Output: 3
```

Reverse a String

Reversing a string can be done by swapping the first and last characters and then recursively reversing the substring that excludes those characters.

```
string ReverseString(string s)
{
    if (s.Length <= 1)  // Base case: string is empty or has one
   character
    {
        return s;
    }
    else
    {
        return s[s.Length - 1] + ReverseString(s.Substring(1, s.Length
   - 2)) + s[0];  // Recursive call
    }
}

// Calling the method and using the returned value
string original = "recursion";
string reversed = ReverseString(original);
Console.WriteLine("Reversed string is: " + reversed);  // Output: "
   noisrucer"
```