Complete Guide to Basic Programming Constructs in C#

1. Conditional Statements

Concept Explanation

Conditional statements allow your program to make decisions and execute different code paths based on certain conditions. They are fundamental to creating dynamic and responsive programs.

Types of Conditional Statements:

- **if statement**: Executes code if a condition is true
- **if-else statement**: Executes one block if condition is true, another if false
- **if-else if-else**: Allows multiple conditions to be checked in sequence
- **switch statement**: Compares a variable against multiple possible values
- **Ternary operator**: Shorthand for simple if-else statements

Key Points:

- Conditions must evaluate to boolean (true/false)
- Use comparison operators: ==, !=, <, >, <=, >=
- Use logical operators: && (AND), || (OR), ! (NOT)
- Switch statements are efficient for multiple discrete value comparisons

Sample Program:

csharp

```
using System;

class ConditionalStatementsDemo
{
    static void Main()
    {
        Console.WriteLine("=== CONDITIONAL STATEMENTS DEMO ===\n");

        // 1. Basic if statement
        int temperature = 25;
        if (temperature > 30)
        {
            Console.WriteLine("It's hot outside!");
        }

        // 2. if-else statement
        int age = 17;
        if (age >= 18)
        {
            Console.WriteLine("You can vote!");
        }
        else
        {
            Console.WriteLine("You cannot vote yet.");
        }

        // 3. if-else if-else chain
        int score = 85;
        if (score >= 90)
        {
            Console.WriteLine("Grade: A");
        }
        else if (score >= 80)
        {
            Console.WriteLine("Grade: B");
        }
        else if (score >= 70)
        {
            Console.WriteLine("Grade: C");
        }
        else if (score >= 60)
        {
            Console.WriteLine("Grade: D");
```

```
}
else
{
    Console.WriteLine("Grade: F");
}

// 4. Switch statement
char grade = 'B';
switch (grade)
{
    case 'A':
        Console.WriteLine("Excellent work!");
        break;
    case 'B':
        Console.WriteLine("Good job!");
        break;
    case 'C':
        Console.WriteLine("Average performance.");
        break;
    case 'D':
        Console.WriteLine("Below average.");
        break;
    case 'F':
        Console.WriteLine("Failing grade.");
        break;
    default:
        Console.WriteLine("Invalid grade.");
        break;
}

// 5. Ternary operator
int number = 15;
string result = (number % 2 == 0) ? "Even" : "Odd";
Console.WriteLine($"The number {number} is {result}");

// 6. Complex conditions with logical operators
int hour = 14;
bool isWeekend = false;

if (hour >= 9 && hour <= 17 && !isWeekend)
{
    Console.WriteLine("Office is open");
}
else
```

```
        {
            Console.WriteLine("Office is closed");
        }
    }
}
```

---

2. Loops

Concept Explanation

Loops allow you to execute a block of code repeatedly based on a condition. They are essential for processing collections, performing repetitive tasks, and implementing algorithms.

Types of Loops:

- **for loop**: Used when you know the exact number of iterations
- **while loop**: Continues while a condition is true (pre-test loop)
- **do-while loop**: Executes at least once, then continues while condition is true (post-test loop)
- **foreach loop**: Iterates through collections and arrays

Key Points:

- Always ensure loop conditions will eventually become false to avoid infinite loops
- Use `break` to exit a loop early
- Use `continue` to skip the current iteration
- Choose the right loop type based on your specific needs

Sample Program:

```csharp
```

```
using System;
using System.Collections.Generic;

class LoopsDemo
{
    static void Main()
    {
        Console.WriteLine("=== LOOPS DEMO ===\n");

        // 1. For loop - when you know the number of iterations
        Console.WriteLine("1. For loop (counting 1 to 5):");
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine($"Count: {i}");
        }

        // For loop with different increments
        Console.WriteLine("\nFor loop (even numbers 0 to 10):");
        for (int i = 0; i <= 10; i += 2)
        {
            Console.WriteLine($"Even number: {i}");
        }

        // 2. While loop - when condition-based repetition is needed
        Console.WriteLine("\n2. While loop (countdown from 5):");
        int countdown = 5;
        while (countdown > 0)
        {
            Console.WriteLine($"Countdown: {countdown}");
            countdown--;
        }

        // 3. Do-while loop - executes at least once
        Console.WriteLine("\n3. Do-while loop (user input simulation):");
        string userInput;
        int attempts = 0;
        do
        {
            attempts++;
            Console.WriteLine($"Attempt {attempts}: Enter 'quit' to exit");
            // Simulating user input
            userInput = (attempts == 3) ? "quit" : "continue";
            Console.WriteLine($"User entered: {userInput}");
```

```
}
while (userInput != "quit" && attempts < 5);

// 4. Foreach loop - for collections and arrays
Console.WriteLine("\n4. Foreach loop with array:");
string[] colors = {"Red", "Green", "Blue", "Yellow"};
foreach (string color in colors)
{
    Console.WriteLine($"Color: {color}");
}

// Foreach with List
Console.WriteLine("\nForeach loop with List:");
List<int> numbers = new List<int> {10, 20, 30, 40, 50};
foreach (int number in numbers)
{
    Console.WriteLine($"Number: {number}");
}

// 5. Nested loops - loops inside loops
Console.WriteLine("\n5. Nested loops (multiplication table):");
for (int i = 1; i <= 3; i++)
{
    for (int j = 1; j <= 3; j++)
    {
        Console.Write($"{i * j:D2} ");
    }
    Console.WriteLine();
}

// 6. Loop control statements
Console.WriteLine("\n6. Loop control (break and continue):");
for (int i = 1; i <= 10; i++)
{
    if (i == 5)
    {
        Console.WriteLine("Skipping 5");
        continue; // Skip this iteration
    }
    if (i == 8)
    {
        Console.WriteLine("Breaking at 8");
        break; // Exit the loop
    }
```

```
            Console.WriteLine($"Processing: {i}");
        }
    }
}
```

---

3. Arrays

Concept Explanation

Arrays are data structures that store multiple elements of the same type in a contiguous memory location. They provide indexed access to elements and are fundamental for organizing related data.

Types of Arrays:

- **Single-dimensional arrays**: Linear sequence of elements
- **Multi-dimensional arrays**: Arrays with multiple dimensions (2D, 3D, etc.)
- **Jagged arrays**: Arrays of arrays where each sub-array can have different lengths

Key Points:

- Array indices start from 0
- Array size is fixed once declared
- Use `.Length` property to get array size
- Arrays are reference types
- Multi-dimensional arrays store elements in a rectangular structure

Sample Program:

```csharp
```

```
using System;

class ArraysDemo
{
    static void Main()
    {
        Console.WriteLine("=== ARRAYS DEMO ===\n");

        // 1. Single-dimensional arrays
        Console.WriteLine("1. Single-dimensional arrays:");

        // Declaration and initialization - Method 1
        int[] numbers1 = new int[5]; // Creates array of 5 integers (all 0)
        numbers1[0] = 10;
        numbers1[1] = 20;
        numbers1[2] = 30;
        numbers1[3] = 40;
        numbers1[4] = 50;

        // Declaration and initialization - Method 2
        int[] numbers2 = new int[] {100, 200, 300, 400, 500};

        // Declaration and initialization - Method 3 (most common)
        int[] numbers3 = {1000, 2000, 3000, 4000, 5000};

        // Displaying array elements
        Console.WriteLine("Numbers1 array:");
        for (int i = 0; i < numbers1.Length; i++)
        {
            Console.WriteLine($"numbers1[{i}] = {numbers1[i]}");
        }

        Console.WriteLine("\nNumbers2 array (using foreach):");
        foreach (int num in numbers2)
        {
            Console.Write($"{num} ");
        }
        Console.WriteLine();

        // 2. Multi-dimensional arrays (2D)
        Console.WriteLine("\n2. Multi-dimensional arrays (2D Matrix):");

        // 3x4 matrix
```

```
int[,] matrix = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

Console.WriteLine("Matrix contents:");
for (int row = 0; row < matrix.GetLength(0); row++) // GetLength(0) = rows
{
    for (int col = 0; col < matrix.GetLength(1); col++) // GetLength(1) = columns
    {
        Console.Write($"{matrix[row, col]:D3} ");
    }
    Console.WriteLine();
}

// 3. Three-dimensional array
Console.WriteLine("\n3. Three-dimensional array:");
int[,,] cube = new int[2, 2, 2] {
    { {1, 2}, {3, 4} },
    { {5, 6}, {7, 8} }
};

for (int x = 0; x < cube.GetLength(0); x++)
{
    Console.WriteLine($"Layer {x}:");
    for (int y = 0; y < cube.GetLength(1); y++)
    {
        for (int z = 0; z < cube.GetLength(2); z++)
        {
            Console.Write($"{cube[x, y, z]} ");
        }
        Console.WriteLine();
    }
    Console.WriteLine();
}

// 4. Jagged arrays (arrays of arrays)
Console.WriteLine("4. Jagged arrays:");

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

```
    for (int i = 0; i < jaggedArray.Length; i++)
    {
        Console.Write($"Row {i}: ");
        for (int j = 0; j < jaggedArray[i].Length; j++)
        {
            Console.Write($"{jaggedArray[i][j]} ");
        }
        Console.WriteLine();
    }

    // 5. Array operations
    Console.WriteLine("\n5. Array operations:");

    int[] operationArray = {5, 2, 8, 1, 9, 3};

    Console.WriteLine("Original array:");
    PrintArray(operationArray);

    // Sorting
    Array.Sort(operationArray);
    Console.WriteLine("After sorting:");
    PrintArray(operationArray);

    // Reversing
    Array.Reverse(operationArray);
    Console.WriteLine("After reversing:");
    PrintArray(operationArray);

    // Finding elements
    int searchValue = 8;
    int index = Array.IndexOf(operationArray, searchValue);
    Console.WriteLine($"Index of {searchValue}: {index}");

    // Array statistics
    Console.WriteLine($"Array length: {operationArray.Length}");
    Console.WriteLine($"First element: {operationArray[0]}");
    Console.WriteLine($"Last element: {operationArray[operationArray.Length - 1]}");
}

// Helper method to print array
static void PrintArray(int[] arr)
{
    foreach (int element in arr)
```

```
        {
            Console.Write($"{element} ");
        }
        Console.WriteLine();
    }
}
```

---

4. Methods and Functions

Concept Explanation

Methods (also called functions) are reusable blocks of code that perform specific tasks. They help organize code, reduce repetition, and make programs more modular and maintainable.

Method Components:

- **Access modifier**: public, private, protected, internal
- **Static keyword**: Method belongs to the class rather than an instance
- **Return type**: The type of value the method returns (void if no return)
- **Method name**: Identifier for the method
- **Parameters**: Input values the method accepts
- **Method body**: The code that executes when method is called

Key Points:

- Methods should have a single responsibility
- Use descriptive names for methods
- Methods can call other methods
- Return statement exits the method and optionally returns a value
- Methods promote code reusability and organization

Sample Program:

csharp

```
using System;

class MethodsDemo
{
    static void Main()
    {
        Console.WriteLine("=== METHODS AND FUNCTIONS DEMO ===\n");

        // 1. Calling methods without parameters
        PrintWelcomeMessage();
        PrintSeparator();

        // 2. Calling methods with parameters
        GreetUser("Alice");
        GreetUser("Bob");

        // 3. Calling methods with return values
        int sum = AddTwoNumbers(15, 25);
        Console.WriteLine($"Sum: {sum}");

        double average = CalculateAverage(85, 92, 78, 96);
        Console.WriteLine($"Average: {average:F2}");

        // 4. Using methods for validation
        string email1 = "user@example.com";
        string email2 = "invalid-email";

        Console.WriteLine($"Is '{email1}' valid? {IsValidEmail(email1)}");
        Console.WriteLine($"Is '{email2}' valid? {IsValidEmail(email2)}");

        // 5. Methods working with arrays
        int[] numbers = {1, 2, 3, 4, 5};
        DisplayArray(numbers);

        int max = FindMaximum(numbers);
        Console.WriteLine($"Maximum value: {max}");

        // 6. Mathematical operations
        double radius = 5.0;
        double area = CalculateCircleArea(radius);
        Console.WriteLine($"Circle area (radius {radius}): {area:F2}");

        // 7. String operations
```

```
    string text = "Hello World";
    int vowelCount = CountVowels(text);
    Console.WriteLine($"Vowels in '{text}': {vowelCount}");

    string reversed = ReverseString(text);
    Console.WriteLine($"Reversed: {reversed}");

    // 8. Complex operations
    bool isPrime17 = IsPrimeNumber(17);
    bool isPrime20 = IsPrimeNumber(20);
    Console.WriteLine($"Is 17 prime? {isPrime17}");
    Console.WriteLine($"Is 20 prime? {isPrime20}");

    int factorial5 = CalculateFactorial(5);
    Console.WriteLine($"Factorial of 5: {factorial5}");
}

// 1. Method with no parameters and no return value
static void PrintWelcomeMessage()
{
    Console.WriteLine("Welcome to the Methods Demo!");
}

static void PrintSeparator()
{
    Console.WriteLine("".PadRight(40, '-'));
}

// 2. Method with parameters but no return value
static void GreetUser(string name)
{
    Console.WriteLine($"Hello, {name}! Nice to meet you.");
}

// 3. Methods with parameters and return values
static int AddTwoNumbers(int a, int b)
{
    return a + b;
}

static double CalculateAverage(params double[] numbers)
{
    if (numbers.Length == 0) return 0;
```

```
    double sum = 0;
    foreach (double num in numbers)
    {
        sum += num;
    }
    return sum / numbers.Length;
}

// 4. Boolean returning methods (validation)
static bool IsValidEmail(string email)
{
    return email.Contains("@") && email.Contains(".");
}

// 5. Methods working with arrays
static void DisplayArray(int[] array)
{
    Console.Write("Array elements: ");
    foreach (int element in array)
    {
        Console.Write($"{element} ");
    }
    Console.WriteLine();
}

static int FindMaximum(int[] array)
{
    if (array.Length == 0) return 0;

    int max = array[0];
    for (int i = 1; i < array.Length; i++)
    {
        if (array[i] > max)
        {
            max = array[i];
        }
    }
    return max;
}

// 6. Mathematical methods
static double CalculateCircleArea(double radius)
{
    return Math.PI * radius * radius;
```

```
}

// 7. String manipulation methods
static int CountVowels(string text)
{
    int count = 0;
    string vowels = "aeiouAEIOU";

    foreach (char c in text)
    {
        if (vowels.Contains(c))
        {
            count++;
        }
    }
    return count;
}

static string ReverseString(string input)
{
    char[] charArray = input.ToCharArray();
    Array.Reverse(charArray);
    return new string(charArray);
}

// 8. Complex algorithmic methods
static bool IsPrimeNumber(int number)
{
    if (number < 2) return false;
    if (number == 2) return true;
    if (number % 2 == 0) return false;

    for (int i = 3; i * i <= number; i += 2)
    {
        if (number % i == 0) return false;
    }
    return true;
}

static int CalculateFactorial(int n)
{
    if (n <= 1) return 1;
    return n * CalculateFactorial(n - 1); // Recursive approach
```

```
    }
}
```

---

5. Method Parameters

Concept Explanation

Method parameters define how data is passed between methods. Understanding different parameter types is crucial for controlling how data flows through your program and managing memory efficiently.

Types of Method Parameters:

- **Value parameters**: Default type, passes a copy of the value
- **Reference parameters (ref)**: Passes a reference to the original variable
- **Output parameters (out)**: Used to return multiple values from a method
- **Parameter arrays (params)**: Allows variable number of arguments

Key Points:

- Value parameters don't affect the original variable
- Reference parameters require initialization before passing
- Output parameters don't need initialization before passing
- Params must be the last parameter in the method signature
- Choose the right parameter type based on your needs

Sample Program:

csharp

```
using System;

class MethodParametersDemo
{
    static void Main()
    {
        Console.WriteLine("=== METHOD PARAMETERS DEMO ===\n");

        // 1. Value Parameters (default behavior)
        Console.WriteLine("1. Value Parameters:");
        int originalNumber = 100;
        Console.WriteLine($"Before calling ModifyValue: {originalNumber}");
        ModifyValue(originalNumber);
        Console.WriteLine($"After calling ModifyValue: {originalNumber}");
        Console.WriteLine("Value parameters don't change the original variable\n");

        // 2. Reference Parameters (ref keyword)
        Console.WriteLine("2. Reference Parameters:");
        int refNumber = 100;
        Console.WriteLine($"Before calling ModifyByReference: {refNumber}");
        ModifyByReference(ref refNumber);
        Console.WriteLine($"After calling ModifyByReference: {refNumber}");
        Console.WriteLine("Reference parameters DO change the original variable\n");

        // 3. Output Parameters (out keyword)
        Console.WriteLine("3. Output Parameters:");

        // Example 1: Mathematical operations returning multiple results
        int dividend = 17, divisor = 5;
        int quotient, remainder;
        DivideNumbers(dividend, divisor, out quotient, out remainder);
        Console.WriteLine($"{dividend} ÷ {divisor} = {quotient} remainder {remainder}");

        // Example 2: Parsing with validation
        string numberString = "123";
        int parsedNumber;
        bool isValid = TryParseInteger(numberString, out parsedNumber);
        Console.WriteLine($"Parsing '{numberString}': Valid = {isValid}, Value = {parsedNumber}");

        string invalidString = "abc";
        bool isValid2 = TryParseInteger(invalidString, out parsedNumber);
        Console.WriteLine($"Parsing '{invalidString}': Valid = {isValid2}, Value = {parsedNumber}");
```

```
// Example 3: Getting circle measurements
double radius = 3.0;
double area, circumference;
CalculateCircleProperties(radius, out area, out circumference);
Console.WriteLine($"Circle (radius {radius}): Area = {area:F2}, Circumference = {circumference:F2}\n");

// 4. Parameter Arrays (params keyword)
Console.WriteLine("4. Parameter Arrays (params):");

// Can call with different numbers of arguments
int sum1 = SumNumbers(1, 2, 3);
Console.WriteLine($"Sum of 1, 2, 3 = {sum1}");

int sum2 = SumNumbers(10, 20, 30, 40, 50);
Console.WriteLine($"Sum of 10, 20, 30, 40, 50 = {sum2}");

int sum3 = SumNumbers(); // No arguments
Console.WriteLine($"Sum of no numbers = {sum3}");

// Can also pass an array
int[] numberArray = {100, 200, 300};
int sum4 = SumNumbers(numberArray);
Console.WriteLine($"Sum of array elements = {sum4}");

// Example with different data types
DisplayInfo("Student Records:", "John Doe", "Jane Smith", "Bob Johnson");
Console.WriteLine();

// 5. Combining different parameter types
Console.WriteLine("5. Combining different parameter types:");

string operation = "multiply";
double num1 = 12.5, num2 = 4.0;
double result;
bool success;

success = PerformCalculation(operation, num1, num2, out result);
Console.WriteLine($"Operation: {operation}, Numbers: {num1}, {num2}");
Console.WriteLine($"Success: {success}, Result: {result}");

// 6. Advanced example: Statistical analysis
Console.WriteLine("\n6. Statistical Analysis Example:");
double[] data = {85.5, 92.0, 78.5, 96.0, 88.5, 91.0, 87.5};
```

```
    double mean, median, standardDeviation;
    AnalyzeData(data, out mean, out median, out standardDeviation);

    Console.WriteLine($"Data Analysis Results:");
    Console.WriteLine($"Mean: {mean:F2}");
    Console.WriteLine($"Median: {median:F2}");
    Console.WriteLine($"Standard Deviation: {standardDeviation:F2}");
}

// 1. Value parameter methods
static void ModifyValue(int value)
{
    value = 999; // This change won't affect the original variable
    Console.WriteLine($"Inside ModifyValue: {value}");
}

// 2. Reference parameter methods
static void ModifyByReference(ref int value)
{
    value = 999; // This change WILL affect the original variable
    Console.WriteLine($"Inside ModifyByReference: {value}");
}

// 3. Output parameter methods
static void DivideNumbers(int dividend, int divisor, out int quotient, out int remainder)
{
    quotient = dividend / divisor;
    remainder = dividend % divisor;
}

static bool TryParseInteger(string input, out int result)
{
    return int.TryParse(input, out result);
}

static void CalculateCircleProperties(double radius, out double area, out double circumference)
{
    area = Math.PI * radius * radius;
    circumference = 2 * Math.PI * radius;
}

// 4. Parameter array methods
static int SumNumbers(params int[] numbers)
{
```

```
    int sum = 0;
    Console.Write("Summing: ");
    foreach (int number in numbers)
    {
        Console.Write($"{number} ");
        sum += number;
    }
    Console.WriteLine();
    return sum;
}

static void DisplayInfo(string title, params string[] items)
{
    Console.WriteLine(title);
    for (int i = 0; i < items.Length; i++)
    {
        Console.WriteLine($"  {i + 1}. {items[i]}");
    }
}

// 5. Combined parameter types
static bool PerformCalculation(string operation, double a, double b, out double result)
{
    result = 0;

    switch (operation.ToLower())
    {
        case "add":
            result = a + b;
            return true;
        case "subtract":
            result = a - b;
            return true;
        case "multiply":
            result = a * b;
            return true;
        case "divide":
            if (b != 0)
            {
                result = a / b;
                return true;
            }
            return false;
        default:
```

```
            return false;
        }
    }

    // 6. Advanced statistical analysis method
    static void AnalyzeData(double[] data, out double mean, out double median, out double standardDeviation)
    {
        // Calculate mean
        double sum = 0;
        foreach (double value in data)
        {
            sum += value;
        }
        mean = sum / data.Length;

        // Calculate median
        double[] sortedData = new double[data.Length];
        Array.Copy(data, sortedData, data.Length);
        Array.Sort(sortedData);

        int midpoint = sortedData.Length / 2;
        if (sortedData.Length % 2 == 0)
        {
            median = (sortedData[midpoint - 1] + sortedData[midpoint]) / 2;
        }
        else
        {
            median = sortedData[midpoint];
        }

        // Calculate standard deviation
        double sumOfSquaredDifferences = 0;
        foreach (double value in data)
        {
            sumOfSquaredDifferences += Math.Pow(value - mean, 2);
        }
        standardDeviation = Math.Sqrt(sumOfSquaredDifferences / data.Length);
    }
}
```

---

6. Method Overloading

Concept Explanation

Method overloading allows you to create multiple methods with the same name but different parameter signatures. This provides flexibility and makes your code more intuitive by allowing the same operation to work with different types or numbers of parameters.

Overloading Rules:

- Methods must have the same name
- Methods must have different parameter signatures (number, type, or order of parameters)
- Return type alone cannot distinguish overloaded methods
- Parameter names don't matter for overloading

Benefits:

- Provides multiple ways to call the same logical operation
- Makes code more readable and intuitive
- Reduces the need to remember different method names for similar operations

Key Points:

- Compiler determines which method to call based on arguments provided
- Most specific match is chosen when multiple overloads could apply
- Overloading is resolved at compile time
- Can overload constructors, operators, and regular methods

Sample Program:

csharp

```
using System;

class MethodOverloadingDemo
{
    static void Main()
    {
        Console.WriteLine("=== METHOD OVERLOADING DEMO ===\n");

        // 1. Basic overloading - different number of parameters
        Console.WriteLine("1. Basic Overloading (different parameter counts):");

        // Calling different versions of CalculateArea
        Console.WriteLine($"Square area (side 5): {CalculateArea(5)}");
        Console.WriteLine($"Rectangle area (5x8): {CalculateArea(5, 8)}");
        Console.WriteLine($"Triangle area (base 6, height 4): {CalculateArea(6, 4, "triangle")}");
        Console.WriteLine();

        // 2. Overloading with different parameter types
        Console.WriteLine("2. Overloading with different parameter types:");

        DisplayValue(42);              // int version
        DisplayValue(3.14159);         // double version
        DisplayValue("Hello World");   // string version
        DisplayValue(true);            // bool version
        DisplayValue('A');             // char version
        Console.WriteLine();

        // 3. Mathematical operations overloading
        Console.WriteLine("3. Mathematical operations overloading:");

        // Add method with different parameter types
        Console.WriteLine($"Add integers (5 + 3): {Add(5, 3)}");
        Console.WriteLine($"Add doubles (2.5 + 3.7): {Add(2.5, 3.7):F2}");
        Console.WriteLine($"Add three integers (1 + 2 + 3): {Add(1, 2, 3)}");
        Console.WriteLine($"Add strings ('Hello' + 'World'): {Add("Hello", "World")}");
        Console.WriteLine();

        // 4. Print methods with different formatting
        Console.WriteLine("4. Print methods with different formatting:");

        Print("Simple message");
        Print("Formatted message", true);
        Print("Centered message", 50, true);
```

```
Print("Custom message", ConsoleColor.Green, true);
Console.WriteLine();

// 5. Search methods in different data structures
Console.WriteLine("5. Search methods with different data types:");

// Array search
int[] intArray = {1, 3, 5, 7, 9, 11};
Console.WriteLine($"Search 7 in int array: Index {Search(intArray, 7)}");

// String array search
string[] stringArray = {"apple", "banana", "cherry", "date"};
Console.WriteLine($"Search 'banana'
```