Logo

Description automatically generated

Day 3

Object Oriented Programming with .Net

Scope

.Net Memory Management

Loops in Programming

The String Class

Formatting Numeric Data

The DateTime Class

File Reading with StreamReader

Regular Expression

Contents

[Scope 2](#_Toc177676518)

[Class Level Scope 2](#_Toc177676519)

[Method Level Scope 2](#_Toc177676520)

[Block Level Scope 3](#_Toc177676521)

[.NET Memory Management 4](#_Toc177676522)

[Understanding Loops in Programming 5](#_Toc177676523)

[For Loop 5](#_Toc177676524)

[While Loop 6](#_Toc177676525)

[Do While Loop 7](#_Toc177676526)

[Foreach Loop 8](#_Toc177676527)

[Break 9](#_Toc177676528)

[Continue 10](#_Toc177676529)

[The String Class 10](#_Toc177676530)

[Commonly Used String Methods 11](#_Toc177676531)

[Escape Characters 13](#_Toc177676532)

[String Manipulation with the String Class 14](#_Toc177676533)

[String Interpolation 15](#_Toc177676534)

[Formatting Numeric Data 16](#_Toc177676535)

[The DateTime Class 17](#_Toc177676536)

[Manipulating Dates 18](#_Toc177676537)

[File Reading with StreamReader 20](#_Toc177676538)

[Using Statement 20](#_Toc177676539)

[Regular Expression 21](#_Toc177676540)

[Regular Expression Operators. 22](#_Toc177676541)

# Scope

In programming, the term "scope" refers to the region of code where a particular variable can be seen or modified. In C#, the scope of a variable is determined by the location where it's declared.

## Class Level Scope

A variable declared at the class level can be accessed by any method within that class. Such variables are often used when multiple methods need to reference or manipulate the same piece of data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Example 1:**  Class scope | |  | | --- | | using System;  namespace ConsoleApp1  {  class Program  { // Class level scope starts here  static int classLevelVar = 10;  public static void Main()  {  // accessing class level variable  Console.WriteLine(classLevelVar);  Console.ReadLine();  }  } // Class level scope ends here  } | | Variable declaration with class level scope  The class scoped variable is visible anywhere within the class   |  | | --- | |  | |

**Note:** The classLevelVar is visible and accessible anywhere within the Program{} class.

## Method Level Scope

A variable declared inside a method can only be accessed within that method. Once the method completes execution, the variable is terminated, and its memory is released.

|  |  |  |  |
| --- | --- | --- | --- |
| **Example 2:**  Method scope | |  | | --- | | using System;  namespace ConsoleApp1  {  class Program  {  static void SendToConsole()  { // method level scope start  int methodLevelVar = 47;  Console.WriteLine(methodLevelVar);  } // method level scope end  public static void Main()  {  SendToConsole();  Console.WriteLine(methodLevelVar);  Console.ReadLine();  }  }  } | | Variable declaration with method level scope |

## Block Level Scope

Variables declared within a specific block of code, like within an if-else statement, for-loop, catch block, are only accessible within that block.

|  |  |
| --- | --- |
| A yellow bell with a white background  AI-generated content may be incorrect. | The block level scope code block is defined by curly Braces { }. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Example 3:**  Block level scope | | |  | | --- | | using System;  namespace ConsoleApp1  {  class Program  {  public static void Main()  {  int m = 47;  if (m == 0)  { // block level scope start  string blockLevelVar = "m = 47";  } // block level scope end  else  {  string blockLevelVar = "m != 47";  }  Console.WriteLine(blockLevelVar);  Console.ReadLine();  }  }  } | | Block level scope variable declaration |
| A yellow bell with a white background  AI-generated content may be incorrect. | When declaring a variable, it's best to limit its scope as much as possible. Broadening the scope increases the risk of unintended changes throughout the object's lifecycle, potentially leading to errors. | | | |

|  |  |
| --- | --- |
| Now you Try:  Correct the error in Example 3 so the output returns the following result:   |  | | --- | |  |   Exercise 1 and 2 |

# .NET Memory Management

In the .NET framework, when your program instantiates variables and objects, the platform uses a specialized mechanism to manage the allocation and deallocation of memory. This process is handled by the Garbage Collector (GC), which automatically monitors and frees memory that is no longer referenced, ensuring efficient memory usage.

The GC class provides access to the Garbage Collector functionality.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor and set a breakpoint on line 9:   |  |  |  |  | | --- | --- | --- | --- | |  | |  |  | | --- | --- | | 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | using System;  namespace ConsoleApp1  {  internal class Program  {  static void Main()  {  long memoryAllocation1 = GC.GetTotalMemory(true);  try  {  AllocateMemory();  long memoryAllocation3 = GC.GetTotalMemory(true);  }  catch (Exception e)  {  throw;  }  Console.ReadLine();  }  static void AllocateMemory()  {  //Create an array with a million elements  byte[] myArray = new byte[10000000];  long memoryAllocation2 = GC.GetTotalMemory(true);  }  }  } | |   The method GC.GetTotalMemory(true); returns the current memory consumed by the application.  While debugging, step through you code starting at line 9 and observe the fluctuation in memory consumption across the AllocateMemory() variables:   * At the outset, memory consumption hovers around 35,000. * Upon entering the AllocateMemory() method, the memory allocation spikes to over ten million.   Once control reverts to the Main() method, it stabilizes back to approximately 35,000. | | |
| 💡 | | You usually don’t need to manually trigger memory collection in .NET. Garbage collection happens automatically. The example above is to showcase the efficiency of .NET’s memory management system. |

# Understanding Loops in Programming

A loop is a fundamental programming construct designed to execute a section of code repeatedly, as long as a certain condition remains true. In C#, you have this suite of loops at your disposal:

for: Repeats code a fixed number of times, using initialization, condition, and increment.

while: Executes code as long as a condition remains true.

do while: Runs code at least once, then continues if the condition is true.

foreach: Iterates through elements in a collection, simplifying access.

## For Loop

The for loop is used when you know exactly how many times you want the loop body to execute. The example below demonstrates a simple for loop:

The declaration and assignment Loop counter incremented

of the loop counter. with every loop iteration.

|  |  |
| --- | --- |
| for (int i = 0; i < 4; i++)  {  Console.WriteLine(i);  } | This condition is checked at the start of every loop iteration. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor and execute it. Then step through the for loop code.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Loop body | |  | | --- | | using System;  namespace ConsoleApp1  {  internal class Program  {  static void Main()  {  for (int i = 0; i < 4; i++)  {  Console.WriteLine(i);  }  Console.ReadLine();  }  }  } | | This code produces the following result.   |  | | --- | | 0  1  2  3 | | |

In this example; during the for loop's first iteration, the counter value is '0'. The loop body is executed when the condition evaluates to true. After each execution of the loop body, the Loop counter is incremented.

|  |  |
| --- | --- |
| 💡 | To save typing, in Visual Studio’s code panel type in the for keyword and hit *Tab* twice. |

|  |  |  |  |
| --- | --- | --- | --- |
| Now you try:  When you load the code below into Visual Studio there are no syntax errors, but when you execute the code there is a problem.   |  |  |  | | --- | --- | --- | | static void Main()  {  for (int i = 0; i < 4;)  {  Console.WriteLine("Value of i: {0}", i);  }  Console.ReadLine();  } | Fix the problem code so you get the following output:   |  | | --- | | Value of i: 0  Value of i: 1  Value of i: 2  Value of i: 3 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Now you try:  Create a program that takes in a number from the console and uses a for loop to produce the exact output as shown here: | |  | | --- | | Enter any number: 12  12  11  10  9  8  7  6  5  4  3  2  1  0 | |   Exercises 3 and 4 |

## While Loop

The while loop repeatedly executes a block of code as long as its specified condition evaluates to true.

In this example, the loop condition is assessed first. If the condition is true, the loop body is executed. The increment operation is performed at the end of the loop body.

Hence, during the loop's initial iteration, the number '0' is displayed.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor and execute it. Then step through the while loop code.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Declaration of the  loop counter  Loop body | |  | | --- | | using System;  namespace ConsoleApp1  {  internal class Program  {  static void Main()  {  int i = 0;  while (i < 7)  {  Console.WriteLine(i);  i++;  }  Console.ReadLine();  }  }  } | | |  |  | | --- | --- | | The result: | 0  1  2  3  4  5  6 |   While loop condition  Counter increment | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | Now you try:  Create a program that takes in a number from the console and uses a while loop to produce the exact output shown here: | |  | | --- | | Enter any number: 12  12  11  10  9  8  7  6  5  4  3  2  1  0 | | Exercises 5 and 6 | | |
| 💡 | | To save typing, in Visual Studio’s code panel type in the while keyword and hit *Tab* twice. | |

## Do While Loop

The do while loop is similar to the while loop with a key distinction: it guarantees at least one execution of its code block before the loop condition is assessed. After this initial execution, the loop will continue iterating as long as the condition remains true.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor and execute it. Then step through the do while loop.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Declaration of the  loop counter  Loop body  While loop condition | |  | | --- | | using System;  namespace ConsoleApp1  {  internal class Program  {  static void Main()  {  int i = 0;  do  {  Console.WriteLine(i);  i++;  }  while (i < 7);  Console.ReadLine();  }  }  } | | |  | | --- | | 0  1  2  3  4  5  6 |   Counter increment | |

In the previous do while loop: The loop body is executed initially, before checking the loop's condition. After the initial execution, the loop condition is then evaluated. If the condition is true, the loop continues to iterate. The increment operation takes place within the loop body. Thus, during the first pass of the loop, the number '0' is displayed.

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Now you try:  Create a program that takes in a number from the console and uses a do while loop to produce the exact output as below: | |  | | --- | | Enter any number: 12  12  11  10  9  8  7  6  5  4  3  2  1  0 | |   Exercise 7 |

## Foreach Loop

The foreach loop is tailored for iterating over collections, such as arrays, lists, and other enumerable types. It provides a more concise way to go through each element without the need for an index.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor and execute it. Then step through the foreach loop.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | The loop body is executed for each element in the array | |  | | --- | | static void Main()  {  string[] wildCats =  {  "Tiger",  "Cougar",  "Panther",  "Lion",  "Bobcat"  };  foreach (string cat in wildCats)  {  Console.WriteLine(cat);  }  Console.ReadLine();  } | | |  | | --- | | Tiger  Cougar  Panther  Lion  Bobcat |   Array contains the element list to loop through  With each loop iteration, the cat variable is populated with the next element in the array | |

In the previous foreach loop: The array wildCats is initialized with a list of wild 5 cat names. For every iteration of the foreach loop, the variable cat is populated with the next element in the wildCats array, and its value is printed to the console.

|  |  |  |  |
| --- | --- | --- | --- |
| Now you try:   |  |  |  | | --- | --- | --- | | Develop a program that displays a list of bird species you might spot in Australia.   1. Create a list that contains five bird species native to Australia. 2. Use a foreach loop to iterate over the list. 3. For each iteration, display the bird species on the console, in the same format as below. | |  | | --- | | Birds to spot in Australia  --------------------------  1. Magpie  2. Emu  3. Rainbow Lorikeet  4. Galah  5. Kookaburra | |   Exercise 8 |

## Break

In C#, the break statement serves as an immediate exit point from loops. Once the break statement is encountered within a loop, no subsequent iterations of that loop will be executed, and control resumes after the loop's closing bracket.

|  |  |  |  |
| --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor and execute it. Then step through the foreach loop.   |  |  | | --- | --- | | |  | | --- | | static void Main()  {  string[] wildCats =  {  "Tiger",  "Cougar",  "Panther",  "Lion",  "Bobcat"  };  foreach (string cat in wildCats)  {  if (cat == "Cougar")  {  Console.WriteLine("It's a " + cat + ", now " +  "let's get out of here!");  break;  }  Console.WriteLine(cat);  }  Console.ReadLine();  } | |  |  | | --- | | Tiger  It's a Cougar, now let's get out of here! | |

|  |  |
| --- | --- |
| A yellow bell with a white background  AI-generated content may be incorrect. | The break statement only exits the **nearest enclosing loop** (such as for, while, or foreach). It does **not** break out of the if statement block. |

## Continue

In C#, the continue statement causes the loop to skip the rest of the current iteration and move directly to the next iteration. Unlike the break statement which exits the loop entirely, continue just bypasses the current loop iteration and continues with the next one.

|  |  |  |  |
| --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor and execute it. Then step through the foreach loop.   |  |  | | --- | --- | | |  | | --- | | static void Main()  {  string[] wildCats =  {  "Tiger",  "Cougar",  "Panther",  "Lion",  "Bobcat"  };  foreach (string cat in wildCats)  {  if (cat == "Cougar" || cat == "Lion")  {  continue;  }  Console.WriteLine(cat);  }  Console.ReadLine();  } | |  |  | | --- | | Tiger  Panther  Bobcat | |

Exercises 9 and 10

# The String Class

The string class isn't merely a data type that holds a sequence of characters. When you declare a variable using the string keyword, you're actually creating an instance of the System.String class. This provides you with a rich library of methods for versatile string operations, including concatenation, comparison, extraction, searching, and trimming.

## Commonly Used String Methods

Below are some of the commonly used *String* methods.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy each code example into your code editor and test it out:  Substring(*startIndex*,*length*)  Extracts a part of the string beginning at *startIndex* and spanning the specified *length*.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var text = "Hello World";  var sub\_text = text.Substring(6, 5);  Console.Write(sub\_text); | | |  | | --- | | World | |   ToUpper()  Converts all characters of the string to uppercase.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var greeting = "Hello World";  Console.Write(greeting.ToUpper()); | | |  | | --- | | HELLO WORLD | |   ToLower()  Converts all characters of the string to lowercase.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var greeting = "Hello World";  Console.Write(greeting.ToLower()); | | |  | | --- | | hello world | |   Trim()  Removes whitespace from the beginning and end of the string.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var greeting = " Hello World ";  Console.Write(greeting.Trim()); | | |  | | --- | | Hello World | |   Split(*delimiter*)  Splits the string at each occurrence of the *delimiter*, returning an array of substrings.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var csv = "Yan,Jenny,John";  string[] names = csv.Split(',');  Console.WriteLine(names.Length);  Console.Write(names[2]); | | |  | | --- | | 3  John | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Join(*delimiter, array*)  Combines the elements of *array* into a single string, separated by the specified *delimiter*.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | string[] nameList = { "Yan","Jenny","John" };  string names = string.Join(", ", nameList);  Console.Write(names); | | |  | | --- | | Yan, Jenny, John | |   Contains(substring)  Checks if the string contains the specified substring, returning true or false.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var greeting = "Hello World";  if (greeting.Contains("Hello")) {  Console.Write("Does contain Hello");  } else {  Console.Write("Does not contain Hello");  } | | |  | | --- | | Does contain Hello | |   Replace(oldValue, newValue)  Replaces occurrences of oldValue in the string with newValue.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var greeting = "Hello World";  var newGreeting = greeting.Replace("World", "Ishana");  Console.Write(newGreeting); | | |  | | --- | | Hello Ishana | |   IndexOf(value)  Finds the first occurrence of value in the string and returns its index.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var greeting = "Hello Aarav!";  var index = greeting.IndexOf("Aa");  Console.Write(index); | | |  | | --- | | 6 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Now you try:  In many software applications, there's a need to process user input. One common task is breaking down strings into components. For example, splitting full names to get first and last names separately.   1. Create a console application that prompts the user for their first and last name.  |  |  |  | | --- | --- | --- | | 1. Split the entered name into its individual components. 2. Use a foreach loop to display each component on a separate line, preceded by a header that says **Your name list is**.   Your output should look like this: | |  | | --- | | Please enter your full name: Craig Watson  Your name list is:  Craig  Watson | |   Exercises 11, 12 and 13 |

## Escape Characters

String literals are enclosed by double quotes (""). However, there are occasions when you need to include special characters like a double quote within a string. To handle these scenarios, escape sequences are used.

Escape sequences begin with a backslash (\) followed by a character to represent a specific character or behavior in a string.

Here are some commonly used escape sequences:

\": Inserts a double quote.

\\: Inserts a backslash.

\n: Inserts a newline.

\t: Inserts a tab.

For instance, if you wanted to write the string He said, "Hello!" in your code, you would write it as:

"He said, \"Hello!\"" using the escape sequence for a double quote.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy each code example into your code editor and test it out:  Double Quote:To embed a double quote within a string, use the \" escape sequence.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var message = "Hello \"World\"";  Console.WriteLine(message); | | |  | | --- | | Hello "World" | |   New Line:To introduce a new line within a string, use the \n escape sequence.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var message = "Hello\nWorld";  Console.WriteLine(message); | | |  | | --- | | Hello  World | |   Back Slash:To include a backslash character within a string, use the \\ escape sequence.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var message = "Hello \\World\\";  Console.WriteLine(message); | | |  | | --- | | Hello \World\ | |   Tab:To insert a tab space within a string, use the \t escape sequence.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | |  | | --- | | string columns = " Column 1\tColumn 2";  Console.WriteLine(columns); | | |  | | --- | | Column 1 Column 2 | |  | |

|  |  |  |  |
| --- | --- | --- | --- |
| Now you try:  You are tasked with creating a C# program that will display a specific phrase to the console. The catch is, you must use a single Console.WriteLine() statement, and after the message is displayed, the console should produce a **bell sound**.   |  |  |  | | --- | --- | --- | | **Hints:** 1. Use the Console.WriteLine()function  combined with string formatting to produce the desired output.  2. Incorporate the bell sound using an appropriate escape character.  3. Google it!  Your output should look like this and produce a bell sound: | |  | | --- | | Hello, welcome to SSD! |   Exercises 14 and 15 | |

## String Manipulation with the String Class

The String class in C# provides robust mechanisms to effectively manipulate textual data. Below are some examples illustrating different facets of string manipulation:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy each code example into your code editor and test it out:  Verbatim Strings  The verbatim identifier, denoted as @, allows for the creation of string literals where backslashes and newlines are treated as normal characters, without needing escape sequences.   * Backslashes are treated as regular characters. * Newlines in the source are preserved.  |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var message =  @"Hi Sally,  you will find the invoice here:  c:\documents\files\invoice.txt";  Console.WriteLine(message); | | |  | | --- | | Hi Sally,  you will find the invoice here:  c:\documents\files\invoice.txt | |   Concatenation  As you have seen previously, the "+" operator is used to concatenate two strings. This operator also allows you the breakup a string over multiple lines. The code below shows an example of string concatenation.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | var message = "Hello "  + "World";  Console.WriteLine(message); | | |  | | --- | | Hello World | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Hands-on Practice Continued:  string.Format()  Another way to format a string is to use the string.Format(*string*) method. It takes a string literal as its first argument. The remaining arguments can pass values and variables which replace specific positions in the string, defined by parameter-based placeholders - {*n*}.   |  |  | | --- | --- | | |  | | --- | | var name = "Asmee";  var greeting = string.Format("Hello {0}, {1}!", name, "good morning");  Console.WriteLine(greeting); | |  |  | | --- | | Hello Asmee, good morning! | |

|  |  |  |  |
| --- | --- | --- | --- |
| Now you try:  Write a C# program that uses the string.Format() method to display this temperature value on the console but formatted to two decimal places.  You're given a temperature value as a decimal: decimal temp = 20.423m;  **Tips:** 1. The string.Format() method allows for formatting strings using placeholders.  You might want to research how to format numbers to a specific number of decimal places with this method.  2. Concatenate this "°C".   |  |  |  | | --- | --- | --- | | Your output should look like this: | |  | | --- | | The temperature is: 20.42°C | |   Exercises 16, 17 and 18 |

## String Interpolation

String interpolation simplifies the embedding of expressions, variables, and even functions directly within string literals. This is achieved by prefixing the string with the "$" symbol.

In the example below, the result of the calculation 2 \* 3 and the method .ToUpper() are directly interpolated into the string.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | var itemType = "bucket List";  var message = $"I have {2 \* 3} items " +  $"on my {itemType.ToUpper()}!";  Console.WriteLine(message); | | |  | | --- | | I have 6 items on my BUCKET LIST! | |

## Formatting Numeric Data

Formatting specifiers can be used alongside string interpolation to customize how numerical values are displayed.

The examples below adjust precision for the integer and decimal values. In your code editor, set up the two numeric variables like below, then copy in the different string interpolation statements :

|  |  |
| --- | --- |
| |  | | --- | | decimal decValue = 1234.5678m;  Console.WriteLine($"{decValue:F}, {decValue:N}, {decValue:C}"); |   Floating-Point Number Currency |

|  |  |  |
| --- | --- | --- |
| The output shows the variable displayed in different ways, based upon the specifier. | |  | | --- | | 1234.57, 1,234.57, $1,234.57 | |

Some of the most common format codes include:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Code | Description | Example: 1234.5678 | | F | Floating-Point | 1234.57 | | N | Number | 1,234.57 | | C | Currency | $1,234.57 | | $ is determined by the system’s regional setting.  More about that later. |

Precision Control

You can fine-tune the **number of decimal places** by adding a digit after the specifier.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Code | Description | Example: 1234.5678 | | F3 | Floating-Point 3 decimals | 1234.568 | | N3 | Number 4 decimals | 1,234.568 | |  |

|  |  |
| --- | --- |
| |  | | --- | | Console.WriteLine($"{decValue:F1},{ decValue:N1}"); |   F Precision 1 N Precision 1 |

|  |  |  |
| --- | --- | --- |
| Here is the output generated from the code above. | |  | | --- | | 1234.0, 1,234.5 | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor. Then execute it:   |  | | --- | | decimal decValue = 9876.5432m;  Console.WriteLine($"{decValue:F}"); // Floating-Point (defaults to 2 decimals)  Console.WriteLine($"{decValue:N}"); // Number with separators  Console.WriteLine($"{decValue:C}"); // Currency |  |  |  |  |  | | --- | --- | --- | --- | | Here is the result: | |  |  | | --- | --- | | 9876.54  9,876.54  $9,876.54 | Notice that the decimal places default to 2. | | | Try it again, but this time with explicit precision:   |  | | --- | | decimal decValue = 9876.5432m;  Console.WriteLine($"{decValue:F3}"); // Floating-Point: 3 decimals  Console.WriteLine($"{decValue:N3}"); // Number with separators: 3 decimals |  |  |  |  |  | | --- | --- | --- | --- | | Here is the result: | |  |  | | --- | --- | | 9876.543  9,876.543 | Notice that the decimal places are now 3. | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Now you try:  Create an application that does the following:   * Prompt the user to input the name of an item they wish to sell (e.g., milk, bread, etc.). * Prompt them to input the cost price of the specified item. * Read the cost into a decimal variable. * Increase the entered cost price by 40% to determine the retail price. * Use string interpolation to display a message indicating the retail price of the item. * Implement a try-catch block to handle potential errors from invalid cost price entries, such as  |  |  |  | | --- | --- | --- | | non-numeric values.   * Your output should look like this: | |  | | --- | | Enter the name of the item you wish to sell: eggs  Enter the cost price of the item: 4.23  The retail price of eggs is $5.92 | |   **Tips:** 1. Use the decimal data type for cost price and retail price calculations to maintain  precision.  2. The exception handling should catch any parsing errors and notify the user with an  appropriate message like "Invalid cost price entry. Please enter a valid number."  Exercises 20 and 21 |

## Working with CultureInfo in C#

Sometimes, you need your program to display **numbers, dates, or currency** in a way that matches a **specific country or culture**.

That’s where the CultureInfo class from System.Globalization comes in.

Using CultureInfo:

* Ensures **currency symbols** match the location (e.g., $, £, €, ¥).
* Handles **number formats** (comma vs. period for decimals).
* Adapts **date formats** (MM/dd/yyyy vs. dd/MM/yyyy).
* Makes applications more **global-friendly**.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor. Then execute it:   |  | | --- | | decimal price = 1234.56m;  // Default (system culture)  Console.WriteLine($"The default price:\t{price.ToString("C")}");  // US dollars  Console.WriteLine($"The USD price:\t\t{price.ToString("C", new CultureInfo("en-US"))}");  // British pounds  Console.WriteLine($"The GBP price:\t\t{price.ToString("C", new CultureInfo("en-GB"))}");  // Japan yen  Console.WriteLine($"The JPN price:\t\t{price.ToString("C", new CultureInfo("ja-JP"))}"); |  |  |  |  | | --- | --- | --- | | Here is the result: | |  | | --- | | The default price: $1,234.56  The USD price: $1,234.56  The GBP price: £1,234.56  The JPN price: ¥1,235 | | |  | | | |

## The ToString Method

In C#, ToString()is a method in the string class which is used to convert values (numbers, dates, etc.) into strings.

As you can see in the example above when the ToString()method is combined with **format specifiers** and CultureInfo, it becomes a powerful way to control how data is displayed.

# The DateTime Class

The DateTime class is essential when dealing with dates and times in C#.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy each code example into your code editor and test it out:  **Now Property:** Retrieves the system's current date and time.  Time component   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | DateTime now = DateTime.Now;  Console.WriteLine(now); | | |  | | --- | | 2025-09-19 8:14:35 PM |   Date component |   **Today Property:** If you need just the date without the time, you can utilize the Today property. This sets the time component to midnight.   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | DateTime today = DateTime.Today;  Console.WriteLine(today); | | |  | | --- | | 2025-09-19 12:00:00 AM | | |

## Manipulating Dates

A DateTime variable provides access to various properties and methods, allowing you to retrieve or modify specific parts of the date/time:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Extract day of the week or year:   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | DateTime now = DateTime.Now;  Console.WriteLine(now);  Console.WriteLine(now.DayOfWeek);  Console.WriteLine(now.DayOfYear); | | |  | | --- | | 2025-09-19 8:27:12 PM  Thursday  263 | |   Modify date and time:   |  |  |  |  | | --- | --- | --- | --- | | |  | | --- | | DateTime now = DateTime.Now;  Console.WriteLine(now.AddYears(1000));  Console.WriteLine(now.AddMonths(12));  Console.WriteLine(now.AddDays(5));  Console.WriteLine(now.AddHours(48));  Console.WriteLine(now.AddSeconds(11)); | | |  | | --- | | 3025-09-19 8:28:15 PM  2026-09-19 8:28:15 PM  2025-09-24 8:28:15 PM  2025-09-21 8:28:15 PM  2025-09-19 8:28:26 PM | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Hands-on Practice: Continued  Date Comparisons and Operations:   |  |  | | --- | --- | | |  | | --- | | DateTime now = DateTime.Now;  DateTime newDate = now.AddYears(10);  if (newDate < now)  {  Console.WriteLine("The new date is in the past");  }  else  {  Console.WriteLine("The new date is in the future");  }  Console.WriteLine($"The two dates difference is:");  Console.WriteLine($"{newDate - now}"); | |  |  | | --- | | The new date is in the future  The two dates difference is:  3652.00:00:00 |   days, minutes, secs, milli secs |

|  |  |
| --- | --- |
| Hands-on Practice:  Copy the code below into your code editor and test it out:   |  | | --- | | static void Main()  {  Console.Write("Enter a date (e.g., 09/25/2025 " +  "or September 25, 2025): ");  string input = Console.ReadLine();  DateTime enteredDate;  bool isDateValid = DateTime.TryParse(input, out enteredDate);  if (isDateValid)  {  Console.WriteLine($"You entered a valid date: {enteredDate}");  Console.WriteLine($"Day of week: {enteredDate.DayOfWeek}");  }  else  {  Console.WriteLine("Invalid input. " +  "Please enter a valid date format.");  }  Console.ReadLine();  } | |

|  |  |  |  |
| --- | --- | --- | --- |
| Hands-on Practice: Continued  **Parsing Dates:** The DateTime.TryParse() method attempts to convert a string representation of a date and time to its DateTime equivalent. It returns a boolean value indicating the success of the conversion.   |  |  | | --- | --- | | Declare the variable to hold the converted DateTime. | The method will populate the out variable with the converted DateTime |  |  | | --- | | DateTime enteredDate;  bool isDateValid = DateTime.TryParse(Console.ReadLine(), out enteredDate); |   The variable isDateValid will hold true if the conversion was successful, otherwise it will be false. |

|  |  |  |  |
| --- | --- | --- | --- |
| Now you try:  Write a program that does the following:   * Prompt the user to enter a specific date in the future (e.g., a wedding, graduation day, etc.). * Calculate the number of days remaining until that event from the current date. * Display the exact time (in days, hours, and minutes) remaining for that event. * Display a message if the date is before Now. * Display a message if the date is in incorrect format.   Your output should resemble this:   |  | | --- | |  |  |  | | --- | |  |  |  | | --- | |  |   **Tip:** To solve this exercise, you'll need to make use of DateTime.TryParse(),a date subtraction, the TimeSpan class and two if conditions (nested).  Exercises 22, 23, 24 and 25 |

# File Reading with StreamReader

While the StreamWriter class allows us to write data to a file, the StreamReader class is its counterpart for reading data from a file.

## Using Statement

The using statement in C# is a convenient syntax that ensures the correct disposal of objects, like files or database connections.

|  |  |  |
| --- | --- | --- |
| Hands-on Practice:  In the example below, we first create a file named TargetSales.csv and populate it with sales data. After writing the data, we use StreamReader to read the file and display its contents on the console. Copy the code below into your code editor and execute it.   |  | | --- | | static void Main()  {  const string FILE\_NAME = "TargetSales.csv";  const bool APPEND = false;  string filePath = "../../" + FILE\_NAME;  StreamWriter sw = new StreamWriter(filePath, APPEND);  // Create and write to the file.  sw.WriteLine("Seattle,05-02-2010,1643690.9,0");  sw.WriteLine("Portland,12-02-2010,1641957.44,1");  sw.WriteLine("San Francisco,19-02-2010,1611968.17,0");  sw.Close();  //Write file contents to the console  using (StreamReader readerObject = new StreamReader(filePath))  {  string line;  while ((line = readerObject.ReadLine()) != null)  {  string[] storeSales = line.Split(',');  Console.Write($"Location: {storeSales[0]}, ");  Console.Write($"Date: {storeSales[1]}, ");  Console.Write($"Sales: {storeSales[2]}, ");  Console.WriteLine($"Is Holiday: {storeSales[3]}");  }  }  Console.ReadLine();  } |   When the code block under the using statement is exited, the using‘s Dispose method is automatically called, ensuring that resources are cleaned up promptly.  Running the application produces the following result:   |  | | --- | | Location: Seattle, Date: 05-02-2010, Sales: 1643690.9, Is Holiday: 0  Location: Portland, Date: 12-02-2010, Sales: 1641957.44, Is Holiday: 1  Location: San Francisco, Date: 19-02-2010, Sales: 1611968.17, Is Holiday: 0 |   Exercise 26 |

# Regular Expression

Regular expressions, often called **regex**, are patterns that help us search, match, and manipulate text.

For example, Canadian postcodes alternate between letters and numbers, like "K1A 0B1". With regex, we can precisely validate this format using a comparison like this:

|  |
| --- |
| string postcode = "K1A 0B1";  Regex.IsMatch(postcode, @"^[A-Za-z]\d[A-Za-z][ -]?\d[A-Za-z]\d$"); |

The Regex.IsMatch() method returns true for any string that matches the Regex pattern.

Using traditional string manipulation to validate Canadian postcodes would be far more cumbersome and less precise.

|  |  |  |  |
| --- | --- | --- | --- |
| Hands-on Practice:  Consider the following example, which uses the Regex class and its IsMatch() method:   |  |  |  | | --- | --- | --- | | static void Main()  {  string[] names = { "Sam"  , "Sally"  , "Steve"  , "Saab"  , "Scott" };  foreach (var name in names)  {  if (Regex.IsMatch(name, "^Sa"))  {  Console.WriteLine($"{name} " +  $"matches the pattern!");  }  }  Console.ReadLine();  } | Regex pattern "^Sa" searches for strings that start with "S" followed by the letter "a".  This would match "S", "Sa", "Saa", and so forth.   |  | | --- | | Sam matches the pattern!  Sally matches the pattern!  Saab matches the pattern! | | |

|  |  |
| --- | --- |
| A yellow bell with a white background  AI-generated content may be incorrect. | To reference the Regex class in C#, include the System.Text.RegularExpressions namespace. |

Exercises 27 and 28

## Regular Expression Operators.

|  |  |
| --- | --- |
| Expression | Description |
| \* | Matches the preceding character 0 or more times. |
| + | Matches the preceding character 1 or more times. |
| ? | Matches the preceding character 0 or 1 times. |
| {n} | Matches the preceding character exactly *n* times. |
| {n,} | Matches the preceding character at least *n* times. |
| {n,m} | Matches the preceding character between *n* to *m* times. |
| ^ | Matches the start of a string. |
| $ | Matches the end of a string. |
| . | Matches any single character. |
| \d | Matches any single decimal digit [0-9]. |
| \D | Matches any non-decimal character. |
| \w | Matches any alphanumeric character or underscore. |
| \W | Matches any non-word character. |
| \s | Matches whitespace characters. |
| \S | Matches non-whitespace characters. |
| \n | Matches a newline character. |
| [0-9] | Matches any digit between 0-9. |
| [a-z] | Matches any lowercase character a-z. |
| [A-Z] | Matches any uppercase character A-Z. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Regex Operator examples with Regex.IsMatch  Copy this code example into your code editor and try it out for yourself:   |  |  | | --- | --- | | \* | 0 or more occurrences of the preceding character | |  | static void Main()  {  const string PATTERN = @"ca\*t";  string[] strings = {"at", "ca", "ct", "cat", "caat", "caaat"};  foreach (var s in strings)  {  if (Regex.IsMatch(s, PATTERN))  {  Console.WriteLine($"{s} matches!");  }  }  Console.ReadLine();  } |  |  | | --- | | ct matches!  cat matches!  caat matches!  caaat matches! | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice: Continued  Regex Operator examples with Regex.IsMatch  Copy the following code examples below into your code editor and try them out for yourself:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | + | | 1 or more occurrences of the preceding character | | | | |  | | --- | | const string PATTERN = @"ca+t";  string[] strings = { "at",  "ca",  "ct",  "cit",  "cat",  "caat",  "caaat"}; | | | |  | | --- | | cat matches!  caat matches!  caaat matches! | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | ? | | 0 or 1 occurrence of the preceding character. | | | | |  | | --- | | const string PATTERN = @"ca?t";  string[] strings = { "at",  "ca",  "ct",  "cit",  "cat",  "caat",  "caaat"}; | | | |  | | --- | | ct matches! cat matches! | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | {n} | | Exactly *n* occurrences of the preceding character. | | | | |  | | --- | | const string PATTERN = @"ca{2}t";  string[] strings = { "at",  "ca",  "ct",  "cit",  "cat",  "caat",  "caaat"}; | | | |  | | --- | | caat matches! | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | {n,} | | At least *n* times of the preceding character. | | | | |  | | --- | | const string PATTERN = @"ca{2,}t";  string[] strings = { "at",  "ca",  "ct",  "cit",  "cat",  "caat",  "caaat"}; | | | |  | | --- | | caat matches!  caaat matches! | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice Continued:  Regex Operator examples with Regex.IsMatch   |  |  |  |  |  | | --- | --- | --- | --- | --- | | ^ | | The word after this element matches the beginning of the string. | | | | |  | | --- | | const string PATTERN = @"^Buddy";  string[] strings = {  "My dog's name is Buddy",  "Buddy is my dog's name",  "My dog's name is Buddy Boy",  "Big Buddy is my dog's name" };  foreach (var s in strings)  {  if (Regex.IsMatch(s, PATTERN))  {  Console.WriteLine ($"{s}");  }  } | | | |  | | --- | | Buddy is my dog's name | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | . | | Is used to match any single character. | | | | |  | | --- | | const string PATTERN = @"c.t";  string[] strings = { "c9",  "ct",  "cat",  "cot",  "caat"}; | | | |  | | --- | | cat matches!  cot matches! | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | \d | | Is used to match any single decimal digit [0-9]. | | | | |  | | --- | | const string PATTERN = @"c\dt";  string[] strings = { "c9a",  "c21t",  "cat",  "c9t",  "caat"}; | | | |  | | --- | | c9t matches! | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | \D | | Is used to match any single non-decimal digit [^0-9]. | | | | |  | | --- | | const string PATTERN = @"c\Dt";  string[] strings = { "ca",  "ct",  "cat",  "cot",  "caat"}; | | | |  | | --- | | cat matches!  cot matches! | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice Continued:  Regex Operator Examples with Regex.IsMatch   |  |  |  |  |  | | --- | --- | --- | --- | --- | | \w | | Is used to match any single letter, number, or underscore character. | | | | |  | | --- | | const string PATTERN = @"c\wt";  string[] strings = { "c9a",  "c\_t",  "cat",  "c9t",  "caat"}; | | | |  | | --- | | c\_t matches!  cat matches!  c9t matches! | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | \W | | Is used to match any non-letter character. | | | | |  | | --- | | const string PATTERN = @"c\Wt";  string[] strings = { "ca",  "ct",  "cat",  "c9t",  "c@t",  "c&t"}; | | | |  | | --- | | c@t matches!  c&t matches! | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | [] | | Is used to match a range of characters. | | | | |  | | --- | | const string PATTERN = @"[ame]";  string[] strings = { "money",  "run",  "yes",  "Sunday",  "eye",  "gun"}; | | | |  | | --- | | money matches!  yes matches!  Sunday matches!  eye matches! | |   **Note:** [ame] means one of those letters (ame) must appear **anywhere** in the string.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | [0-9] | | Is used to match any numeric in the range of 0-9. | | | | |  | | --- | | const string PATTERN = @"[0-9]";  string[] strings = { "mo9ey",  "run",  "ye2s",  "Sunday",  "ey212e",  "gun77"}; | | | |  | | --- | | mo9ey matches!  ye2s matches!  ey212e matches!  gun77 matches! | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Regex Operator Examples with Regex.IsMatch   |  |  |  |  |  | | --- | --- | --- | --- | --- | | [a-z] | | Is used to match any lower-case character in the range of a-z. | | | | |  | | --- | | const string PATTERN = @"[a-z]";  string[] strings = { "890",  "30s",  "a098",  "490A8",  "4234F",  "CAN"}; | | | |  | | --- | | 30s matches!  a098 matches! | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | | [A-Z] | | Is used to match any upper-case character in the range of A-Z. | | | | |  | | --- | | const string PATTERN = @"[A-Z]";  string[] strings = { "890",  "30s",  "a098",  "490A8",  "4234F",  "CAN"}; | | | |  | | --- | | 490A8 matches!  4234F matches!  CAN matches! | | |

**Note:** Regex patterns vary in complexity. While you don't need to master them, a basic understanding can greatly improve text-processing tasks.

|  |  |  |
| --- | --- | --- |
| Hands-on Practice:  The Regex.Match() method checks a string against a specified regex pattern. It returns a Match object, which provides details about the match like its success, value, index, and length.  In the example below, the pattern ^Disneyland checks if strings begin with the word "Disneyland". The ^ operator in regex ensures that the match should be at the start of the string.   |  | | --- | | static void Main()  {  const string PATTERN = @"^Disneyland";  string[] strings =  {  "Disneyland is the happiest place on Earth",  "The happiest place on Earth is Disneyland"  };  Console.WriteLine($"Regex Pattern: {PATTERN}\n" +  $"--------------------------");  foreach (var thisString in strings)  {  Console.WriteLine($"String: {thisString}\n");  Match match = Regex.Match(thisString, PATTERN);  if (match.Success)  {  Console.WriteLine($" Match Found: {match.Success}");  Console.WriteLine($" Value: {match.Value}");  Console.WriteLine($" Index: {match.Index}");  Console.WriteLine($" Length: {match.Length}");  }  else  {  Console.WriteLine(" No match found!");  }  Console.WriteLine("-");  }  Console.ReadLine();  } |   The code above outputs the following detail.   |  | | --- | | Regex Pattern: ^Disneyland  --------------------------  String: Disneyland is the happiest place on Earth  Match Found: True  Value: Disneyland  Index: 0  Length: 10  -  String: The happiest place on Earth is Disneyland  No match found!  - | |

|  |  |  |  |
| --- | --- | --- | --- |
| Now you try:   |  |  |  | | --- | --- | --- | | In the code provided in the previous exercise, the Regex pattern is set to identify strings that begin with "Disneyland" using the ^ operator.  Alter the Regex pattern to employ the $ operator so that the Regex.Match() method identifies a match only when "Disneyland" appears at the end of a string.  Upon successful completion, your program's output should this. | |  | | --- | | Regex Pattern: Disneyland$  --------------------------  String: Disneyland is the happiest place on Earth  No match found!  -  String: The happiest place on Earth is Disneyland  Match Found: True  Value: Disneyland  Index: 31  Length: 10  - | | |

Regular expressions often combine multiple operators to craft intricate patterns. For instance, consider the task of validating a traditional first name:

It should start with a capital letter: ^[A-Z].

|  |  |
| --- | --- |
| ^[A-Z][a-z]+$ | It should end after these lowercase letters: $. |

It should be followed only by lowercase letters appearing one or more times: [a-z]+.

|  |  |
| --- | --- |
| Hands-on Practice:  Here's how this pattern functions:   |  | | --- | | static void Main()  {  const string PATTERN = @"^[A-Z][a-z]+$";  string[] names = { "Tom" , "Jenny0", "Tom Tom"  , "sally", "T" , "Tony" };  Console.WriteLine($"Checking names against" +  $" pattern: {PATTERN}");  foreach (var name in names)  {  bool isMatch = Regex.IsMatch(name, PATTERN);  Console.WriteLine($"Does '{name}' fit the " +  $"pattern? {(isMatch ? "YES" : "NO")}");  }  Console.ReadLine();  } | |

|  |  |
| --- | --- |
| The program will produce the following output.   |  | | --- | | Checking names against pattern: ^[A-Z][a-z]+$  Does 'Tom' fit the pattern? YES  Does 'Jenny0' fit the pattern? NO  Does 'Tom Tom' fit the pattern? NO  Does 'sally' fit the pattern? NO  Does 'T' fit the pattern? NO  Does 'Tony' fit the pattern? YES | |

Exercises 29 and 30