Logo

Description automatically generated

Day 2

Object Oriented Development with .Net

Casting

Implicite Conversion

Explicit Conversion

Enumeration

Arrays

Multidimensional Arrays

C# Operators

Conditional Statements

Methods

Flow Of Execution

Method Parameters

Contents

[Casting 2](#_Toc207131295)

[Implicit Casting 2](#_Toc207131296)

[Explicit Casting 3](#_Toc207131297)

[Examples of Explicit Casting 3](#_Toc207131298)

[Casting Between Non-Integral Types 3](#_Toc207131299)

[Casting with Error Handling 5](#_Toc207131300)

[Enumeration 6](#_Toc207131301)

[Arrays 7](#_Toc207131302)

[Array Length 8](#_Toc207131303)

[Array Sort 9](#_Toc207131304)

[Multidimensional Arrays 9](#_Toc207131305)

[Multidimensional Array Length 11](#_Toc207131306)

[C# Operators 11](#_Toc207131307)

[Arithmetic Operators 11](#_Toc207131308)

[Comparison Operators 13](#_Toc207131309)

[Logical Operators 14](#_Toc207131310)

[Conditional Statements 15](#_Toc207131311)

[If, Else if, Else Statement 15](#_Toc207131312)

[Ternary Operator (Shorthand If...Else) 16](#_Toc207131313)

[Switches 17](#_Toc207131314)

[Methods 19](#_Toc207131315)

[Why Use Methods 19](#_Toc207131316)

[Method Syntax 20](#_Toc207131317)

[Flow Of Execution 20](#_Toc207131318)

[Visual Studio Colour-Coding 22](#_Toc207131319)

[Method Overloading 23](#_Toc207131320)

# Casting

In object-oriented programming, data can exist in different types. The action of converting one type to another is known as casting. C# provides two main types of casting: Implicit Conversion and Explicit Conversion.

## Implicit Casting

When converting a smaller integral type into a larger integral type there is no risk of data loss. The compiler automatically converts one type of data to another without the need for additional syntax.

|  |  |  |
| --- | --- | --- |
|  | Minimum Value | Maximum Value |
| int | -2,147,483,648 | 2,147,483,647 |
| long | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| Float | ±1.5 × 10^−45 | ±3.4 × 10^38 |
| double | ±5.0 × 10^−324 | ±1.7 × 10^308 |

Implicit Casting, small into larger is done automatically: int 🡪 long 🡪 float 🡪 double

In the example below the variable numAllowed, which is of type int, is assigned to displayValue, a long type. Since the range of long is larger than that of int, there's no risk of data loss, eliminating the need for additional coding.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hands-on Practice:  In your Visual Studio code editor, replace your main method with the one below.   |  | | --- | | static void Main()  {  int numAllowed = 635;  long displayValue;  displayValue = numAllowed;  Console.Write($"Total people allowed is: {displayValue}");  Console.ReadLine();  } |   The original int value was passed to a long with no data loss as indicated by the output below.   |  |  |  | | --- | --- | --- | | |  | | --- | | Total people allowed is: 635 | | **Note:** In the examples today, I use string interpolation ($"") which allows for inserting expressions directly into the string. This makes the code more readable and concise. | |

Quiz Questions 1 and 2

## Explicit Casting

This conversion requires the use of a specific casting operator. Since this conversion can result in data loss (for example, truncating values when converting from double to int), the compiler requires an explicit instruction from the programmer to proceed. This is called **explicit casting**.

Explicit Casting, large into smaller requires a cast: double 🡪 float 🡪 long 🡪 int

## Examples of Explicit Casting

|  |  |  |
| --- | --- | --- |
| int: | |  | | --- | | int num1 = (int)635.99; |   num1 is set to 635 and the decimal is truncated. |

In the example below, resulting in the value 123.

The decimal portion is lost in this conversion.

|  |  |  |
| --- | --- | --- |
| Hands-on Practice:  In your Visual Studio code editor, replace your main method with the one below.   |  | | --- | | static void Main()  {  double d = 123.45;    int i = (int)d;  Console.Write($"Number of attempts: {i}");  Console.ReadLine();  } |   The double value 123.45 is explicitly cast to an int, and the decimal portion was lost as indicated in the output below.   |  | | --- | | Number of attempts: 123 | |

## Casting Between Non-Integral Types

For non-integral types like strings, conversion methods like Int32.Parse() or Int32.TryParse() are used.

|  |
| --- |
| int num2 = Int32.Parse("635"); |

Num2 is set to 635 after it is converted to an int.

**Note:** You must be careful with explicit conversions. A wrong cast can lead to run time errors.

In the example below, the string value 123 is cast to an int, using the Int32.Parse() method.

|  |  |  |
| --- | --- | --- |
| Hands-on Practice:  In your Visual Studio code editor, replace your main method with the one below.   |  | | --- | | static void Main()  {  string s = "123";  int i;  i = Int32.Parse(s);  Console.Write($"Converted the string into an int: {i}");  Console.ReadLine();  } |   The original string value was passed to an int as indicated in the output below.   |  | | --- | | Converted the string into an int: 123 | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1. Now you try:   Modify the value of s so it is "123.45", Then run your application.   |  | | --- | | static void Main()  {  string s = "123.45";  int i;  i = Int32.Parse(s);  Console.Write($"Converted a string into an int: {i}");  Console.ReadLine();  } |   Place a screenshot of the result in the box below.   |  |  | | --- | --- | |  | | |  | | It is important to never allow your application to crash like this. When errors are not handled gracefully users quickly lose confidence in your application. | |   Quiz Questions 3, 4, 5 and 6 |

## Casting with Error Handling

|  |
| --- |
| bool isValid = Int32.TryParse("635", out int numOut); |

numOut is set to 635 after it is converted to an int. isValid will contain true or false depending on the success of the conversion.

|  |  |  |
| --- | --- | --- |
| Hands-on Practice:  In your Visual Studio code editor, replace your main method with the one below.   |  | | --- | | static void Main()  {  string s = "123";  int i;  bool isValid = Int32.TryParse(s, out i);  Console.WriteLine($"Converted the string into an int: {i}");  Console.Write($"Was the conversion valid: {isValid}");  Console.ReadLine();  } |   The original string value was passed to an int as indicated in the output below.   |  | | --- | | Converted the string into an int: 123  Was the conversion valid: True | |

|  |  |  |
| --- | --- | --- |
| 1. Now you try:   Modify the value of s so it is "123.45" then run your application.   |  | | --- | | static void Main()  {  string s = "123.45";  int i;  bool valid = Int32.TryParse(s, out i);  Console.WriteLine($"Converted a string into an int: {i}");  Console.Write($"Was the conversion valid: {valid}");  Console.ReadLine();  } |   Place a screenshot of the result in the box below.   |  | | --- | |  |   Notice the value of i and valid. Quiz Question 7 |

# Enumeration

An enumeration, declared using the enum keyword, is a data type that represents a set of named constants. It's a way to assign meaningful names to integral values for improved code readability and maintainability.

Declare using the enum keyword.

Assign meaningful names to integral values.

Instead of using numbers, which might not be intuitive, we can use names.

It’s possible to convert an enum’s name using ToString(). More about that later.

|  |
| --- |
| using System;  class Program  {  enum Days  {  Sunday = 1  , Monday = 2  , Tuesday = 3  , Wednesday = 4  , Thursday = 5  , Friday = 6  , Saturday = 7  }  static void Main()  {  string wednesday = Days.Wednesday.ToString();  int dayValue = (int)Days.Wednesday;  Console.WriteLine($"{wednesday} is day {dayValue}");  Console.WriteLine($"{Days.Saturday} is day {(int)Days.Saturday}");  Console.ReadLine();  }  } |

Use $"" (string interpolation) to save coding.

It’s possible to convert an enum’s underlying value using a cast.

|  |  |  |
| --- | --- | --- |
| Executing this code will produce the following result: | |  | | --- | | Wednesday is day 4  Saturday is day 7 | |

|  |  |  |  |
| --- | --- | --- | --- |
| A yellow bell with a white background  AI-generated content may be incorrect. | | enum members can only be assigned integer values (e.g., byte, int, long). The assignment of "A" or 11.3 to Sunday will result in a syntax error. | |
| 1. Now you try:   This example assigns the values implicitly.  Copy the code in the example above into your editor and modify the enum to implicitly set the Days values.   |  |  | | --- | --- | | enum Days  {  Sunday  , Monday  , Tuesday  , Wednesday  , Thursday  , Friday  , Saturday  } | Here the values of the enum elements start from 0 and increase by one for each element.  Sunday = 0, Monday = 1 and so on. |   In the box below, write the values given for Wednesday and Saturday after you have made the change.   |  | | --- | | 3, 6 |   Why are the values this way. | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. Now you try:   Write a program to enumerate the four seasons implicitly, Winter being the first season. Have your program display meaningful names and the underlying values for each season.  Place your enum in the box below:   |  | | --- | | Enum Seasons  {  Winter  , Spring  , Summer  , Autumn  } | |  |   Write the necessary code inside the Main() method to produce the following result:   |  |  | | --- | --- | | Winter = 0  Spring = 1  Summer = 2  Autumn = 3 |  |   Quiz Questions 8 and 9 |

# Arrays

The .NET Array class represents a fixed-length sequential collection of items all of the same type. The key properties of an array are:

A Fixed Length: Once declared, its size cannot change and to expand you need to create a new array.

Is Indexed: Each item can be accessed using an index, starting from position 0.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  To explore this further, paste the code below into the VS Code Editor.  [] Required to declare anarray type.   |  |  | | --- | --- | | static void Main()  {  string[] hawaiianIslands = new string[3];  hawaiianIslands[0] = "Hawaii";  hawaiianIslands[1] = "Maui";  hawaiianIslands[2] = "Oahu";  Console.WriteLine(hawaiianIslands[2]);  Console.ReadLine();  } | An array of length 3 is created.  Each element is assigned a value.  Index 2 points to the third element. |  |  |  |  |  | | --- | --- | --- | --- | | Executing the code will produce this result. | |  | | --- | | Oahu | |  | | |

In the array example above, the **Declare then Assign** approach is used. The array was first declared, specifying its type and size. Then later, the values were assigned to each of the elements.

The code below shows you how to **Declare and Assign Simultaneously**. You can declare an array and at the same time fill it with values. Like this:

|  |
| --- |
| string[] hawaiianIslands = {"Hawaii", "Maui", "Oahu"}; |

|  |  |  |
| --- | --- | --- |
| This automatically creates a string[] array containing 3 elements and fills it with the specified values. The output from the new code should be exactly the same as before: | |  | | --- | | Oahu | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. Now you try:   Declare an array of integers named ages with a length of 5. Assign the following values to the array: 20, 25, 30, 35, 40. Use the **Declare then Assign** **Simultaneously** approach. Then write each element to the console.   |  |  |  |  | | --- | --- | --- | --- | | Executing your code will produce this result: | |  | | --- | | 20  25  30  35  40 | | Quiz Questions 10 and 11 | |

## Array Length

Sometimes, you need to know how many elements are stored in an array. You can achieve this by using the .Length property, which returns the total number of elements in the array. In the example below, .Length is appended to the hawaiianIslands array, returning the number of elements it contains.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Paste the main methos below into your VS Code Editor.   |  | | --- | | static void Main()  {  string[] hawaiianIslands = {"Hawaii", "Maui", "Oahu"};  Console.WriteLine(hawaiianIslands.Length);  Console.ReadLine();  } |   Use the array’s Length property to determine the number of properties in the array.   |  |  |  |  | | --- | --- | --- | --- | | Executing the code will produce this result. | |  | | --- | | 3 | |  | | |

## Array Sort

When you need to arrange an array in numerical or alphabetical order, you can use the Array.Sort() method.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  To explore these features further, paste the code below into the VS Code Editor.  Use the Array.Sort(*array*) method to sort the array in alphabetical order.   |  | | --- | | static void Main()  {  string[] surfingIslands = { "Maui", "Oahu", "Hawaii" };  Array.Sort(surfingIslands);  Console.WriteLine(surfingIslands[0]);  Console.WriteLine(surfingIslands[1]);  Console.WriteLine(surfingIslands[2]);  Console.ReadLine();  } |  |  |  |  |  | | --- | --- | --- | --- | | Executing the code will produce this result. | |  | | --- | | Hawaii  Maui  Oahu | |  | | |

## Multidimensional Arrays

Multidimensional arrays occur when each element of the array contains another array, essentially "arrays in arrays", A two-dimensional array can be imagined as a table with rows and columns.

The table below illustrates a two-dimensional array with three rows and two columns.

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| 0 | "Hawaii" | "The Big Island" |
| 1 | "Maui" | "The Valley Isle" |
| 2 | "Oahu" | "The Gathering Place" |

To access the value in a specific cell, you use two indices instead of one. The two indices correspond to the row and column of the cell you're trying to access.

|  |  |
| --- | --- |
| A yellow bell with a white background  AI-generated content may be incorrect. | Two-dimensional arrays are 0-indexed. Meaning the cell containing "Hawaii" is accessed with the [0,0] indices. |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  To explore these features further, paste the code below into the VS Code Editor.  [,] Required to declare a two-dimensional array.   |  |  | | --- | --- | | static void Main()  {  string[,] islandsInfo = new string[3, 2]  {  {"Hawaii", "The Big Island"},  {"Maui", "The Valley Isle"},  {"Oahu", "The Gathering Place"}  };  Console.WriteLine(islandsInfo[2, 1]);  Console.ReadLine();  } | The cells are accessed using the row index and the column index. |  |  |  |  |  | | --- | --- | --- | --- | | Executing the code will produce this result. | |  | | --- | | The Gathering Place | |  | | |

Just like reading a cell’s value, you can assign a value by specifying the row and column indices. You can assign a value to a cell like this.

|  |
| --- |
| int[,] matrix = new int[2, 3];  matrix[1, 2] = 42; |

|  |  |
| --- | --- |
| 1. Now you try: 2. Declare a two-dimensional integer array that has 2 rows and four columns. 3. Assign 97 to the cell positioned specifically at row 1, column 2. 4. Then write the value positioned specifically at row 1, column 2 to the console along with a short message.   Executing your code will produce the following result.   |  | | --- | | The number you assigned to row 1 column 2 is 97 |   Quiz Questions 12 and 13 |

## Multidimensional Array Length

To find out the length of any specific dimension of a multidimensional array, use the GetLength() method. You can see this method used in the code below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Copy the following code into your code editor.  [,,] Required to declare a three-dimensional array.   |  |  | | --- | --- | | static void Main()  {  int[,,] threeDimensional = new int[3, 2, 4];  Console.WriteLine($"1st Dimension: {threeDimensional.GetLength(0)}");  Console.WriteLine($"2nd Dimension: {threeDimensional.GetLength(1)}");  Console.WriteLine($"3rd Dimension: {threeDimensional.GetLength(2)}");  Console.ReadLine();  } | Writes 3  Writes 2  Writes 4 |  |  |  |  |  | | --- | --- | --- | --- | | Executing the code will produce this result: | |  | | --- | |  | |  | | |

Quiz Question 14

# C# Operators

Operators allow developers to manipulate data and variables in meaningful ways. They are the building blocks of any program, allowing for basic arithmetic operations to more complex logic evaluations.

## Arithmetic Operators

The arithmetic operators allow developers to perform basic mathematical computations. Below is a table illustrating the commonly used arithmetic operators in C#.

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Description | Example (a=5, b=3) |
| + | Addition | Adds two operands. | a + b = 8 |
| - | Subtraction | Subtracts the second operand from the first. | a - b = 2 |
| \* | Multiplication | Multiplies two operands. | a \* b = 15 |
| / | Division | Divides the first operand by the second. | a / b = 1.67 |
| % | Modulo | Returns the remainder when the first is divided by the second. | a % b = 2 |
| ++ | Increment | Increases the value by 1. | a++ = 6 |
| -- | Decrement | Decreases the value by 1. | a-- = 4 |

|  |  |  |
| --- | --- | --- |
| Hands-on Practice:  Place the code below in your editor and execute it.   |  | | --- | | static void Main() {  // Variables initialization  float a = 5.3f;  float b = 4.2f;  float c = 6.0f;  float result;  // Displaying original values  Console.WriteLine($"A = {a} B = {b} C = {c}");  Console.WriteLine("-------------------------");  // Displaying addition operation  result = a + b;  Console.WriteLine($"A + B = {result:F2}");  // Displaying multiplication operation  result \*= c;  Console.WriteLine($"(A + B) \* C = {result:F2}");  Console.ReadLine();  } |   Executing your code will produce the following result.   |  | | --- | | A = 5.3 B = 4.2 C = 6  -------------------------  A + B = 9.50  (A + B) \* C = 57.00 | |

|  |  |  |
| --- | --- | --- |
| 1. Now you try:   Continue with Example above, add the following comparison operators:  • Subtraction  • Division  • Modulo  • Increment  • Decrement  Executing your code will produce the following result.   |  |  | | --- | --- | | A = 5.3 B = 4.2 C = 6  --------------------------  A + B = 9.50  (A + B) \* C = 57.00  A - B = 1.10  A / B = 1.26  A % B = 1.10  A after A++ = 6.30  B after B-- = 3.20 | Quiz Questions 15 and 16 | |

## Comparison Operators

Comparison operators enable developers to evaluate two values, producing a boolean outcome (True or False). This outcome helps in making decisions within the program based on the results of the comparisons.

Below is a table of the most common comparison operators in C#:

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Description | Example (where a=5, b=3) | Returned |
| == | Equal to | a == b | False |
| != | Not equal | a != b | True |
| > | Greater than | a > b | True |
| < | Less than | a < b | False |
| >= | Greater than equal to | a >= b | True |
| <= | Less than equal to | a <= b | False |

Let's explore how comparison operators work in an example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hands-on Practice:  Place the code below in your editor and execute it.   |  | | --- | | static void Main()  {  // Variables initialization  float a = 5.3f;  float b = 4.2f;  // Displaying original values using string interpolation  Console.WriteLine($"A = {a} B = {b}");  Console.WriteLine("-----------------");  // Comparing A and B using equality operator  Console.WriteLine($"A == B: {a == b}");  // Comparing A and B using not-equal operator  Console.WriteLine($"A != C: {a != b}");  Console.ReadLine();  } |  |  |  |  | | --- | --- | --- | | Executing your code will produce the following result. | |  | | --- | |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Now you try:   Continue with Example above, add the following comparison operators:  • greater than  • less than  • greater than equal to  • less than equal to.   |  |  |  | | --- | --- | --- | | Executing your code will produce this output: | |  | | --- | |  | |   Quiz Question 17 |

## Logical Operators

Logical operators are used to test conditions and produce a boolean result (true or false). They are fundamental in controlling program flow by allowing complex evaluations in decision-making constructs.

Consider these variables for the examples below: bool a = true.

bool b = false.

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Description | Example | Returned |
| && | True if both conditions are true | a && b | false |
| || | True if at least one condition is true | a || b | true |
| ! | Reverses the result (true becomes false) | !(a && b) | true |

|  |  |  |
| --- | --- | --- |
| static void Main()  {  bool a = true;  bool b = false;  bool both = a && b;  bool result = !(both);  Console.WriteLine("a = " + a);  Console.WriteLine("b = " + b);  Console.WriteLine("(a && b) = " + both);  Console.Write("!(a && b) = " + result);  Console.ReadLine();  } | Since a is true and b is false,  a && b evaluates to false.  The ! operator negates the result of a && b and so result becomes true.   |  | | --- | | a = True  b = False  (a && b) = False  !(a && b) = True | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hands-on Practice:  Place the code below in your editor and execute it.   |  | | --- | | static void Main()  {  // Boolean inputs  bool x = true;  bool y = false;  bool z = true;  // Logical operations  bool allTrue = x && y && z;  bool atLeastOneTrue = x || y || z;  bool negationOfAllTrue = !(x && y && z);  // Output results  Console.WriteLine("All true: " + allTrue);  Console.WriteLine("At least one true: " + atLeastOneTrue);  Console.WriteLine("Negation of all true: " + negationOfAllTrue);  Console.ReadLine();  } |  |  |  |  | | --- | --- | --- | | Executing your code will produce the following result. | |  | | --- | | All true: False  At least one true: True  Negation of all true: True | | |

Quiz Questions 18 and 19

# Conditional Statements

Conditional statements allow you to execute specific blocks of code based on certain conditions.

## If, Else if, Else Statement

The if, else if, and else statement provides a way to conditionally execute blocks of code.

|  |  |  |
| --- | --- | --- |
| For readability, the code block should be indented between the curly braces starting with  ‘{ ‘ and ending with ‘}’.  An else if can be used to test other conditions to execute alternative code blocks. | if (condition)  {  // Block of code  }  else if (anotherCondition)  {  // Block of code  }  else  {  // Block of code  } | When the if condition is true, this block of code is executed.  The final else is a catch-all and the final code block is executed when all conditions evaluate to false. |

|  |  |
| --- | --- |
| Statement | Description |
| if(condition) {…} | Specifies a block of code to execute if a specified condition is true. |
| else {…} | Specifies a block of code to execute if the same condition is false. |
| else if(condition) {…} | Specifies a new condition to test if the first condition is false. |

|  |  |
| --- | --- |
| Hands-on Practice:  Place the code below in your editor and execute it.   |  | | --- | | static void Main()  {  float a = 5.3f, b = 4.2f, c = 6.0f;  Console.WriteLine($"a = {a} b = {b} c = {c}");  Console.WriteLine("-------------------------");  if (a != b)  {  Console.WriteLine("a is not equal to b.");  }  else if (b == c)  {  Console.WriteLine("b is equal to c.");  }  else  {  Console.WriteLine("No condition are met.");  }  Console.ReadLine();  } | |

|  |  |
| --- | --- |
| Hands-on Practice: Continued  Executing your code will produce the following result.   |  | | --- | | a = 5.3 b = 4.2 c = 6  -------------------------  a is not equal to b. | |

Quiz Questions 20, 21 and 22

## Ternary Operator (Shorthand If...Else)

There is a more concise way to perform a simple check. It’s known as the ternary operator ?. You can replace a simple if else statement with a single line. The statement consists of three operands.

|  |
| --- |
| condition ? Console.WriteLine("it’s true!") : Console.WriteLine("it’s false!"); |

When this condition is true, this is executed; otherwise, this is selected.

Let's explore how the ternary operator works with an example:

|  |  |  |
| --- | --- | --- |
| Hands-on Practice:  Place the code below in your editor and execute it.   |  | | --- | | public static void Main()  {  const int TORNADO\_THRESHOLD = 40;  int windSpeed = 35;  string alert = windSpeed >=  TORNADO\_THRESHOLD ? "Tornado alert!" : "Weather is calm.";  Console.Write(alert);  Console.ReadLine();  } |   Executing your code will produce the following result.   |  | | --- | | Weather is calm. | |

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Now you try:   Continue with Example above, change the wind speed to 40.  Executing your code will produce the following result.   |  |  |  | | --- | --- | --- | | |  | | --- | | Tornado alert! | | Quiz Question 23 | |

## Switches

A switch statement is a control flow statement that allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

A screenshot of a computer code

Description automatically generated

When should you use a switch?

Imagine having a long chain of if-else statements to check against multiple values. This is where the switch statement shines. It makes the code more readable when dealing with multi-way branching.

|  |  |  |
| --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor.  Place a breakpoint on the switch  expression then execute your code.   |  | | --- | | public static void Main()  {  const int SUCCESS = 200;  const int NOT\_FOUND = 404;  const int UNAUTHORIZED = 401;  int htmlReturnCode = 404;  switch (htmlReturnCode)  {  case SUCCESS:  Console.Write("Success!");  break;  case NOT\_FOUND:  Console.Write("Page not found.");  break;  case UNAUTHORIZED:  Console.Write("You are not authorized.");  break;  default:  Console.Write("Unknown error.");  break;  }  Console.ReadLine();  } |   Step through the code (F10) and pay attention to the path taken.  Executing your code will produce the following result.   |  | | --- | | Page not found. | |

|  |
| --- |
| 1. Now you try:   Now, modify the value of htmlReturnCode in the example above so each case is tested.  Leave the breakpoint where it is. Then step through each case as well as the default. |

|  |  |
| --- | --- |
| 1. Now you try:   • Create a new C# Console Application.  • Prompt the user to enter a number between 1 and 7, representing the days of the week.   * Monday is 1 and Sunday is 7.   • Use a switch statement to evaluate the user's input.  • Print out the corresponding day of the week. If the user enters a number outside the range of 1-7, notify them of the error.  Starter code:   |  | | --- | | static void Main()  {  Console.Write("Enter a number between 1 and 7: ");  string input = Console.ReadLine();  if (!int.TryParse(input, out int day))  {  Console.WriteLine("Error: please enter " +  "a valid number.");  }  else  {  // Your switch statement here.  }  Console.ReadLine();  } | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1. Now you try: Continued   Expected behavior:   |  |  | | --- | --- | | Enter a number between 1 and 7: 1  That day is Monday. | When 1 is entered. |  |  |  | | --- | --- | | Enter a number between 1 and 7: #  Error: please enter a valid number. | When # is entered. |  |  |  | | --- | --- | | Enter a number between 1 and 7: 9  Error: number must be between 1 and 7. | When 9 is entered. |   Quiz Questions 24 and 25 |

# Methods

As your programs grow in complexity, it's not wise to cluster all your logic within the Main method, as we've done so far in this lesson. Instead, it's a best practice to organize related statements into separate units known as methods. A method is a designated block of code that encapsulates a series of statements.

## Why Use Methods

Avoid Repetition:Instead of writing the same code multiple times, a method can be called multiple times.

Modularity:Methods enable you to break down a complex operation into smaller, more manageable parts.

Readability:By naming methods appropriately, you make the code more descriptive and easier to understand.

## Method Syntax

Methods are declared by specifying the access level, return type, name, and parameters. The diagram below illustrates the components that make up the Main method used in your applications so far.

A screen shot of a computer

Description automatically generated

Access Specifier: Determines the visibility of the method (e.g. public, private). Default is private.

Static Keyword: While we'll delve deeper later, understand that static methods can call other static methods without creating an object. For your current needs, ensure that all of your methods are declared as static.

Return Type:Specifies the data type of the method's return value. If there's nothing to return, use void.

Method Name:A unique, case-sensitive identifier for the method.

Parameter(s): Variables passed into the method. They are enclosed within parentheses and separated by commas.

Method Signature:Comprises the method name and its parameters, excluding the return type. C# allows methods with the same name but different signatures.

Method Body:Contains the logic (statements) for the method, enclosed between { and }.

## Flow Of Execution

Understanding the flow of execution is crucial in programming. It precisely defines the order in which lines of code are executed during a program's run. Every console application begins its journey with the Main method.

When a method is invoked, the flow takes a detour: it jumps to the first line of the called method, processes its instructions, and then returns to resume where it left off before the method call.

The example below demonstrates the flow of execution in a program with method calls.

|  |  |  |  |
| --- | --- | --- | --- |
| Hands-on Practice:  Copy the code below into your code editor.  Place a breakpoint on this method call: GreetUser("Craig").   |  |  |  | | --- | --- | --- | | class Program  {  public static void Main()  {  GreetUser("Craig");  int result = AddNumbers(5, 3);  Console.WriteLine("Sum: " + result);  }  static void GreetUser(string name)  {  Console.WriteLine("Hello " + name + "!");  }  static int AddNumbers(int a, int b)  {  return a + b; // Return statement  }  Console.ReadLine();  } | Step through the code using F11 and pay attention to the path taken.  Calling AddNumbers() method will return an int value which is assigned to result.  Executing the code completely will produce the following result.   |  | | --- | | Hello Craig!  Sum: 8 |   Try it again. This time step through your code using F10.  What is different? | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hands-on Practice:  Place the code below in your editor and execute it.   |  |  | | --- | --- | | *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* | using System;  namespace ConsoleApp1  {  class Program  {  static float MultiplyTwoNumbers(float a, float b)  {  float c = a \* b;  return c;  }  static void ShowResult(string operation, float result)  {  Console.Write("The " + operation + " is " + result);  }  public static void Main()  {  float product = MultiplyTwoNumbers(3f, 4.25f);  ShowResult("product", product);  Console.ReadLine();  }  }  } |  * Place a breakpoint on the first executable line in the main method (line 20) and start the application. Line 20 invokes the MultiplyTwoNumbers() method and passes into it two float parameters. * Go ahead and press F11 to step into the MultiplyTwoNumbers() method. * Continue with F11 until control is returned to the Main method and continues to line 22. This line invokes the ShowResult() method and passes along string and float values. ShowResult() writes to the console and returns void. * Press F11 to step into the ShowResult() method. * Continue with F11 until control is returned to the Main method and continues to line 24.  |  |  |  | | --- | --- | --- | | * Press F5 to continue and you will produce this result: | |  | | --- | | The product is 12.75 | | |

**Note:** Debug F10 will step over a method but will still process all the code inside the method. F11 serves a similar but slightly different purpose: it steps into an invoked method, allowing you to observe the flow inside the called method.

## Visual Studio Colour-Coding

The Visual Studio IDE enhances code readability by colour-coding various elements. Different colours for different parts of the code break up the text and make it easier to read and understand.

|  |  |
| --- | --- |
| A yellow bell with a white background  AI-generated content may be incorrect. | Visual Studio theme colours can differ, so the colour coding you observe may change based on the specific theme you have chosen. I have chosen the dark theme. |

Below is a breakdown of how Visual Studio categorizes and colours parts of your C# code in **Dark mode**:

|  |  |
| --- | --- |
| Comments | Will be in green. |
| Keywords | Like static, class, using, and string will be in blue. |
| Strings | Literals like "Hello, SSD!" will be in red. |
| Control flow | Such as if, else, and foreach may be highlighted in pink. |
| Namespaces | And class properties will be white (System, args.Length). |
| Classes | Will be in turquoise (Console, Program). |
| Variables | Such as message and args will be in light blue. |
| Methods | Like Main and DisplayGreeting will be in yellow. |

|  |  |
| --- | --- |
| A yellow bell with a white background  AI-generated content may be incorrect. | Recognizing and understanding the color-coding in Visual Studio can enhance your coding efficiency, aiding in quick identification of elements, highlighting potential errors, and improving overall code readability. |

Quiz Questions 26 and 27

## Method Overloading

If two or methods perform similar routines, you may want to assign the same name to each of them. This is only allowed if the method signatures are different. Meaning each method must have a different arrangement of input parameters. This is known as method overloading.

The code below implements three different overload variations of the same method, AddTwoNumbers().

|  |  |  |  |
| --- | --- | --- | --- |
| Hands-on Practice:  Place the code below in your editor and execute it.  Method signatures must be unique.   |  |  | | --- | --- | | static int AddTwoNumbers(int a, int b)  {  Console.Write("Adding integer values : ");  return a + b;  }  static float AddTwoNumbers(float a, float b)  {  Console.Write("Adding float values : ");  return a + b;  }  static double AddTwoNumbers(double a, double b)  {  Console.Write("Adding double values : ");  return a + b;  }  static float AddTwoNumbers(int a, float b)  {  Console.Write("Adding an integer and a float : ");  return a + b;  }  public static void Main() {  Console.WriteLine($"3.0f + 4.5f = {AddTwoNumbers(3.0f, 4.5f)}");  Console.WriteLine($"3.83 + 6.22 = {AddTwoNumbers(3.83, 6.22)}");  Console.WriteLine($"2 + 2 = {AddTwoNumbers(2, 2)}");  Console.WriteLine($"3 + 4.5f = {AddTwoNumbers(3, 4.5f)}");  Console.ReadLine();  } | This method takes two int parameters and returns their sum.  This method takes two float parameters and returns their sum.  This method takes two double parameters and returns their sum.  This method demonstrates overloading with mixed data types. It takes an int and a float parameter and returns their sum as a float. |   Executing your code will produce the following result.   |  | | --- | | Adding float values : 3.0f + 4.5f = 7.5  Adding double values : 3.83 + 6.22 = 10.05  Adding integer values : 2 + 2 = 4  Adding an integer and a float : 3 + 4.5f = 7.5 | |

|  |  |  |
| --- | --- | --- |
| 1. Now you try:   Continue with Example above, add the following method to your code.   |  | | --- | | static string AddTwoNumbers(int z, int y)  {  int sum = z + y;  return sum.ToString();  } |   Place a screenshot of errors shown in your Error list panel.   |  | | --- | |  |   Quiz Questions 28, 29 and 30 |