Writing Basic Programs

[Introduction 1](#_Toc422138651)

[By the end of this lesson, you will be able to 1](#_Toc422138652)

[Structuring Code 3](#_Toc422138653)

[Statements 3](#_Toc422138654)

[General Structure 4](#_Toc422138655)

[Example: Comments 5](#_Toc422138656)

[Header Documentation 5](#_Toc422138657)

[Defining collapsible regions 7](#_Toc422138658)

[Creating Text-Based Applications 9](#_Toc422138659)

[Displaying Output 9](#_Toc422138660)

[Kinds of Strings to Display 9](#_Toc422138661)

[Collecting Input 11](#_Toc422138662)

[Representing Data 14](#_Toc422138663)

[Where Values are Stored 15](#_Toc422138664)

[Example: Data Types 16](#_Toc422138665)

[Converting between Value and Reference Types 17](#_Toc422138666)

[Example: Boxing 17](#_Toc422138667)

[Integral Types 18](#_Toc422138668)

[Floating-Point Types 20](#_Toc422138669)

[Casting 22](#_Toc422138670)

[Other Types 23](#_Toc422138671)

[char 23](#_Toc422138672)

[bool 24](#_Toc422138673)

[string 24](#_Toc422138674)

[void 25](#_Toc422138675)

[Defining Variables and Constants 26](#_Toc422138676)

[Identifiers 26](#_Toc422138677)

[Constants 26](#_Toc422138678)

[Local Variables 27](#_Toc422138679)

[Activity: Create a Simple Console Application 29](#_Toc422138680)

[Part 1: Start a New Project 29](#_Toc422138681)

[Part 2: Create a simple text interface 31](#_Toc422138682)

[If you have time 32](#_Toc422138683)

[Wrap Up 33](#_Toc422138684)

[Defining Sets of Constants 34](#_Toc422138685)

[Allowing Empty Value Types 38](#_Toc422138686)

[Syntax 38](#_Toc422138687)

[Casting Nullable types 39](#_Toc422138688)

[Using Operators 41](#_Toc422138689)

[Math Operators 41](#_Toc422138690)

[Assignment 41](#_Toc422138691)

[Increment and Decrement Operators 42](#_Toc422138692)

[Relational Operators 42](#_Toc422138693)

[Logical Operators 43](#_Toc422138694)

[Full Order of Operations 44](#_Toc422138695)

[Branching 46](#_Toc422138696)

[Unconditional Branching 46](#_Toc422138697)

[Reusing Method Names 47](#_Toc422138698)

[Passing Data with Parameters 49](#_Toc422138699)

[Conditional Branching 50](#_Toc422138700)

[The If/Else Statements 50](#_Toc422138701)

[The Switch Statement 52](#_Toc422138702)

[Looping 55](#_Toc422138703)

[The While Loop 55](#_Toc422138704)

[The Do-While Loop 57](#_Toc422138705)

[The For Loop 59](#_Toc422138706)

[Activity: Validating User Input 63](#_Toc422138707)

[If You Have Time 67](#_Toc422138708)

[Exercise Solution 67](#_Toc422138709)

## Introduction

The goal of this lesson is to lay a solid foundation for programming in the C# language. This will allow you to:

* Quickly create basic programs in C#
* Read and understand segments of C# code
* Learn the later lessons more efficiently

|  |  |
| --- | --- |
|  | Although some parts of this lesson may seem basic to an experienced programmer, these concepts are critical for your success during later lessons. |

### By the end of this lesson, you will be able to

* Concepts (things to understand):
* Explain why Epic has strictly enforced conventions for identifiers in C#
* Describe the difference between a field and a constant
* Describe the difference between a value type and a nullable value type and when you would use one over the other
* Explain why you would divide code into collapsible regions
* Describe the difference between a strongly-typed programming language like C# vs. a loosely-typed programming language like M
* Explain why you would use one data type vs. another
* Explain why you would use an enumeration
* Explain the difference between an input variable, a variable passed using the out keyword, and a variable passed using the ref keyword
* Explain the difference between value and reference types
* Explain why exceptions can occur during casting
* Explain how and why you might reuse a method name
* Explain how namespaces are used at Epic
* Algorithms (things to be able to do):
* Write statements
* Group statements into blocks
* Write comments in various forms
* Read from and write to the Console
* Use operators to form expressions
* Cast between data types
* Define methods
* Identify which method is called given several overloaded methods
* Branch conditionally
* Create looping code

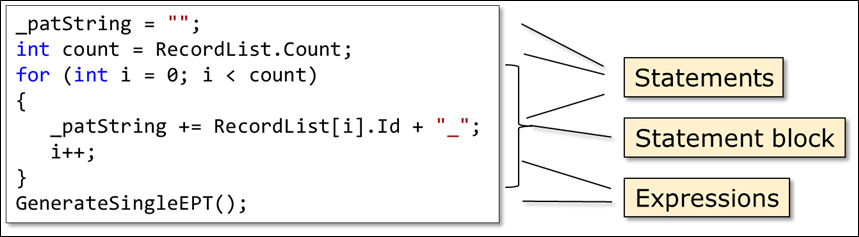
## Structuring Code

Like other programming languages, there are some basic rules you need to follow when writing valid code in C#.

### Statements

There are three general forms of code in C#:

|  |  |
| --- | --- |
| Statement | A line of code in C#. Every statement must end with a semicolon  ; |
| Statement block | A group of statements enclosed in braces: { }.  Can be used in place of statements.  Statement blocks usually have headers that indicate when that block of code will run. |
| Expressions | Statements that evaluate to a value.  Can sometimes be used in place of statements. |



|  |  |
| --- | --- |
|  | Try It Yourself: Code Structure   1. Start Visual Studio 2. Create a new project for in-class examples  * FILE > New Project > Visual C# > Console Application * Name: ClassExamples * Location: (Use the default location)  1. Notice the code that was created:  using System; using System.Collections.Generic; using System.Linq;using System.Text; using System.Threading.Tasks; namespace ClassExamples {   class Program   {     static void Main(string[] args)     {     }   } } 2. Based on this example, can code blocks be nested?  * Yes  1. Is a semicolon required when ending a statement block?  * No |

### General Structure

The general code structure of C# varies from other languages like M and VB that you may already be familiar with:

* White space (e.g., spaces, tabs and blank lines) is allowed. Add white space to make code readable.
* There is no line continuation character. A statement ends when a semicolon is reached.
* There are multiple ways to include comments:
* Single line; denoted by // (two forward slashes)
* Multiple lines; enclosed within /\* .......... \*/

#### Example: Comments

|  |
| --- |
| /\* This is a block comment    It can span multiple lines \*/  // End of line comment |

|  |  |
| --- | --- |
|  | Try It Yourself: Whitespace and Comments   1. Try adding white space and blank lines in various places. Determine where whitespace is allowed and where it isn't. 2. Try adding several single-line comments using // 3. Try adding several multi-line comments using /\* ... \*/ 4. Use the mouse or SHIFT + UP/DOWN  to select multiple lines of code at once 5. Press CTRL+K and then CTRL+C. What happened?  * All of the selected lines were commented out using //  1. Press CTRL+K and then CTRL+U. What happened?  * The lines of code were uncommented. |

#### Header Documentation

C# has header-documentation built directly into the language in the form of XML comments, so, there isn't a standard Epic header template.

* XML documentation allows for ease of documentation of code.
* XML comments start with ///
* They can be applied to types or members.
* You may also embed XML tags which are then extracted to build HTML documentation that can be shared.

##### Predefined XML Tags

Here are a few of the most commonly used XML documentation tags.

|  |  |
| --- | --- |
| summary | Place a general description of what the purpose of the code you are documenting is.  <summary>Place your description here</summary> |
| remarks | Use to add information, supplementing the information specified with summary. |
| param | Documents the purpose of a parameter. Like other languages, parameters can be input, output or both, so be sure to include that in the description. An attribute of the param element is name, which should match the name of the parameter you are documenting:  <param name="nameOfParameter">Description</param> |
| returns | Describe the data that this code returns, if it returns anything.  <returns>description</returns> |
| exception | Exceptions are thrown if code encounters an unexpected error and cannot continue as a result. Use this to document the kinds of exceptions that could be thrown, if they are known ahead of time.  <exception>description</exception> |
| example | Use this to provide examples of how the code could/should be used.  <example>code</example> |
| c | Use to indicate that some part of another element is code.  <example>To call this method, use  <c>class.method()</c> </example> |

|  |  |
| --- | --- |
|  | For more XML comment elements, see:  [https://msdn.microsoft.com/en-us/library/b2s063f7(v=vs.80).aspx](https://msdn.microsoft.com/en-us/library/b2s063f7%28v=vs.80%29.aspx) |

##### Example: Header documentation

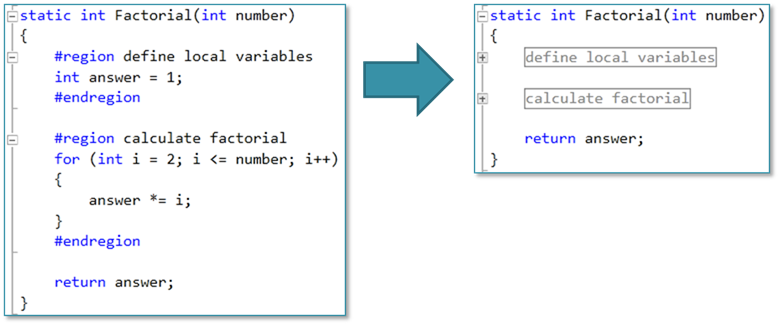
|  |
| --- |
| /// <summary> /// Calculate the factorial of a number /// </summary> /// <param name="number">  /// The number to calculate the factorial of  /// </param> /// <returns>The factorial of the number</returns> static int Factorial(int number) {   int answer = 1;    for (int i = 2; i <= number; i++)   {     answer \*= i;   }   return answer; } |

|  |  |
| --- | --- |
|  | Try it Yourself: Header Documentation   1. Directly above the line block beginning with static void Main(string args[]), insert a new line. 2. On the new line, type three slashes: ///. What happened?  * Adds XML comments: /// <summary> ///  /// </summary> /// <param name="args"></param> static void Main(string[] args)  1. Directly above the line marked class Program, insert a new line. 2. On the new line type three slashes: ///. How is this result different?  * There isn't a param comment  1. Try adding three slashes above the namespace and using statements. What happens?  * Nothing. XML comments shouldn't be placed in those locations  1. Remove the /// comments from above the namespace and using statements. 2. Fill out the header documentation with whatever you want. 3. Within the code block below Main, type Main( and then wait. What do you notice regarding the header comments?  * They appear in the intellisense popup. |

### Defining collapsible regions

* Code that collapses to a heading
* Useful for outlining code:

|  |
| --- |
| static int Factorial(int number) {    #region define local variables    int answer = 1;   #endregion    #region calculate factorial   for (int i = 2; i <= number; i++)   {     answer \*= i;   }   #endregion    return answer; } |



|  |  |
| --- | --- |
|  | Try It Yourself: Regions   1. Try including several regions in your CodeExample project. 2. Try nesting one region inside another one |

## Creating Text-Based Applications

Throughout this course you'll create text-based programs, which use the console to read and write text, so it's important to be familiar with some of the most common Console methods. In order to use any Console method, you need to have access to the System namespace, which includes the Console's definition.

|  |  |
| --- | --- |
|  | A method is a function or subroutine that is also a member of a class. In C# all functions and subroutines are part of a class, so we will refer to all callable code as methods. |

|  |  |
| --- | --- |
| Namespace | A way to logically divide code into different modules. By only including namespaces that you actually use, it will be easier to find the code you are looking for. This will make programming faster and less error prone. |
| using System; | System is the namespace that includes many of the most commonly used framework features from .NET, including the Console. |

|  |  |
| --- | --- |
|  | When using Visual Studio, the using System; statement (as well as other namespaces) will be included automatically in every program. |

### Displaying Output

The console has several methods for writing output to text.

|  |  |
| --- | --- |
| Console.Write | Writes a string to the Console, keeping the cursor on the same line as the string.  Console.Write("Enter name: "); |
| Console.WriteLine | Write a string to the Console and move the cursor to the next line.  Console.WriteLine("Hello world!"); |

#### Kinds of Strings to Display

A string written to the console can be a literal (enclosed in double quotes) or you can use placeholders inside of the string to dynamically substitute in values from variables. To include multiple place holders in the same string, increment the value of the place holder (enclosed in { }) by 1 for each additional placeholder. The variables are listed after the string in the order in which they should replace the place holders.

|  |  |
| --- | --- |
| Literal | Enclosed in double quotes  Console.WriteLine("Hello world!"); |
| Constructed | Use placeholders that are replaced at runtime. |

|  |
| --- |
| string greeting = "Hello"; string subject = "world"; Console.WriteLine("{0} {1}!", greeting, subject); |

|  |  |
| --- | --- |
|  | You can also construct strings that you store in variables using the string.Format method. For example, given these variables:  string greeting = "Hello"; string subject = "world";  these two statements are equivalent:   1. Console.WriteLine("{0} {1}!", greeting, subject); 2. string message = string.Format("{0} {1}!", greeting, subject); Console.WriteLine(message);   Constructed strings are particularly useful with internationalization (I18N) because the translator can rearrange the placeholders during translation. |

|  |  |
| --- | --- |
|  | Try using code snippets!  Code snippets are pre-defined code-segment templates that you can use to make your time programming more efficient. Think of them as smart text for source code.  To use a code snippet, type the snippet, then press TAB twice. If there are fields, you can tab through the fields of the snippet to populate them. Try this snippet:  cw to quickly enter Console.WriteLine()  More snippets will be introduced as they become relevant. |



### Collecting Input

There are several ways to get input from the user through the console.

|  |  |
| --- | --- |
| Read | Reads the next character from the input stream.  char aChar = Console.Read(); |
| ReadKey | Reads the next keystroke.  Use to determine exactly which key the user pressed, or to respond to any key being pressed.  ConsoleKeyInfo key = Console.ReadKey();  One common use is to paus execution until the user enters anything. |
| ReadLine | Reads the next line of characters (until a carriage return), resulting in a string value:  string name = Console.ReadLine(); |

If the input is supposed to be numeric, you'll need a way to convert the string into a number. One way to perform this action is to access the Parse method from the desired numeric class. Later in this course we'll discuss alternate methods of converting a string into a number.

|  |
| --- |
| Console.Write("Please enter a string: "); string text = Console.ReadLine();  Console.Write("Please enter an integer: "); int number = int.Parse(Console.ReadLine());  Console.WriteLine("You entered '{0}' and {1}", text, number); Console.ReadKey(); |

|  |  |
| --- | --- |
|  | If the input into the Parse method cannot be successfully converted into the specified numeric type, C# will throw an exception. |

|  |  |
| --- | --- |
|  | For now, you can think of an exception as an error. If an exception is not handled by the surrounding code, the application will crash. |

|  |  |
| --- | --- |
|  | Try It Yourself: Console   1. In your ClassExamples project, within the code block that starts with static void Main(string args[]), add the following line of code:  * Console.WriteLine("Hello world!");  1. Run the project by clicking Start, or pressing F5 2. What happened?  * The program ran, but quit immediately after.  1. Add this code below the call to WriteLine:  * Console.ReadKey();  1. What happens this time?  * The program pauses  1. Add the prompt "Press any key to continue" immediately before the call to Console.ReadKey. 2. Experiment by prompting the user for information and storing it as a string or an int. Try passing a non-numeric string to the number prompts to see what an exception looks like. |

Sample solution:

|  |
| --- |
| static void Main(string[] args) {   // Write a message   Console.WriteLine("Hello world!");   //   // Read in a string   Console.Write("Enter a string: ");   string input = Console.ReadLine();   Console.WriteLine("You entered {0}.", input);   //   // Read in an integer   Console.Write("Enter a number: ");   int number = int.Parse(Console.ReadLine());   Console.WriteLine("You entered {0}.", number);   //   // Pause until a key is pressed   Console.WriteLine("<Press any key to continue>");   Console.ReadKey(); } |

|  |  |
| --- | --- |
|  | When you hit the play button in Visual Studio, how does Visual Studio know what code to run?  You have several files in your project -- you should be able to see App.config, Program.cs, and (if you expand Properties), AssemblyInfo.cs.  When determining what code to run, Visual Studio looks only at the files in your startup project - the bolded project. (Right now, you only have one project, but you'll have more later.) In those files, it looks for a method called Main(string[] args) and tries to run that method.  The code you just wrote is in the Main method, so that's what runs.  As a test, try changing the name of the Main method or creating another Main method. Your code will not compile. |

## Representing Data

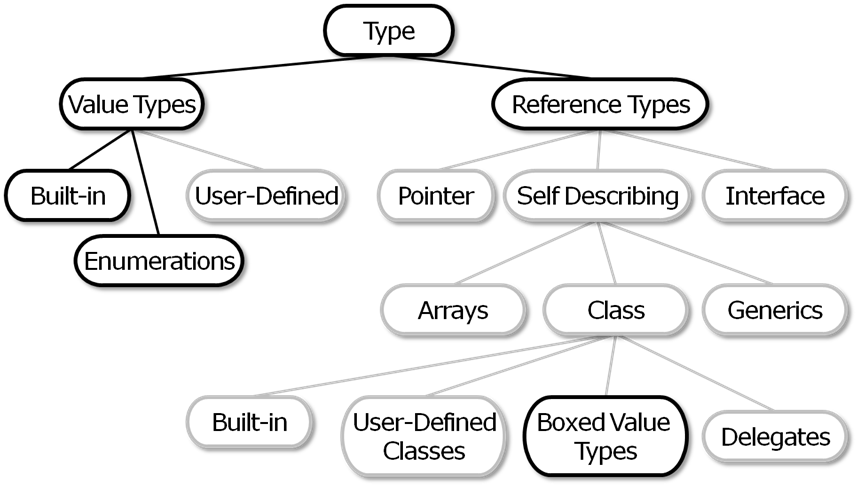
To be able to program effectively in any programming language, it is critical to understand how that language represents data.

|  |  |
| --- | --- |
|  | In M, there were only two data types, numbers and strings. In C# there are many more. There is string, int, bool, decimals, etc., and because programmers can make new types, the number of types that exist is an ever expanding frontier.  C# is a strongly-typed language. In M, you can pretty much store any piece of data in any variable. Not in C#. The data that you put into a variable has to match the type of that variable. If it doesn't match, you're code won't compile. This is what it means for a language to be strongly-typed. Strongly-typed languages allow the compiler to check for potential data errors before the code is even run. |

|  |  |
| --- | --- |
|  | The main advantages of a strongly-typed language (one with many strict data types) are   * Safety - a strongly-typed language can prevent errors from occurring that a weakly-typed language couldn't prevent. By demanding that the types of certain variables be declared at compile-time, the compiler can be smarter about what code will work and what code will fail. It can then prevent failing code from ever being created in the first place. * Readability - because variables have a defined type, the code tends to be more readable. * Ease of debugging - data type bugs arising from unintended data type mutations can be difficult to track down. The situation gets particularly nasty is when the code runs smoothly but produce unexpected results. Strongly-typed languages make most data type mutations obvious by demanding that the programmer make their intentions clear at compile-time. |

C# data types can be broadly broken into two groups: Value Types and Reference Types. These differ based upon where the data is actually stored and how it is accessed. We will explore some data types now and some during later lessons.

Because C# is a strongly-typed language, when you declare a variable, you must declare the data type of that variable along with the name. The variable may then only contain values of or references to the given type.



### Where Values are Stored

Value types and reference types are stored in different memory locations. The two locations are the stack and the heap.

|  |  |
| --- | --- |
| Stack | The stack is the location in memory that tracks what code is currently being executed, as well as any local variables that have been defined. The top level stack indicates the code that is currently being executed, while lower levels contain code that will be returned to once the higher levels are complete. The stack has the following properties:   * Quick access * Minimal overhead |
| Value Types | The data for a value type variable is allocated in the stack level where it is defined. When the stack level containing the variable moves out of scope, the data stored in that variable is no longer available. |
| Heap | The heap is a separate memory location. It can be accessed by any reference type variable in any stack level. Objects in the heap are cleaned up periodically by the .NET framework's garbage collector. Objects are only be cleaned up by the garbage collector when there is no way to reach the object from the stack. |
| Reference Types | When declaring a variable of a reference type, the variable on the stack does not hold the actual data. Instead, it holds a memory address that points other code to a memory location on the heap. This memory address is often called a reference or a pointer.  If a stack level that holds a reference type variable is removed, then the reference is also removed, but the data will remain on the heap until it is garbage collected (assuming there is no othe rpath to that data from the stack). |

#### Example: Data Types

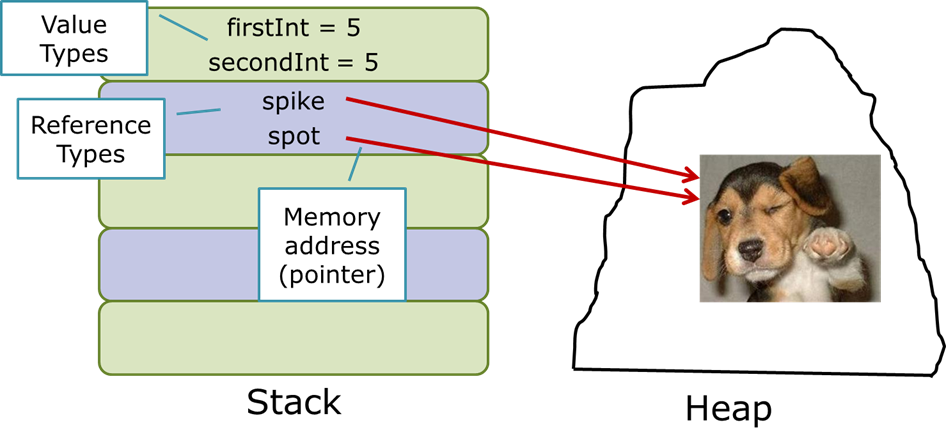
In this example, there are two value type variables and two reference type variables defined. The value types are firstInt and secondInt, while the reference types are spike and spot.

|  |
| --- |
| // Value types int firstInt = 5; int secondInt = firstInt;  // Reference types Dog spike = new Dog(); Dog spot = spike; |

When one variable is assigned the value of another, whatever is stored on the stack is copied from the source variable and assigned to the target variable. For value types, this means that the actual information that we are interested in (5 in this case) is duplicated and stored in two separate memory locations on the stack.

Because reference types store a heap memory address in the stack, it is the address that is copied from one variable to the next. Although the memory address is stored in two separate variables, the information that they reference on the heap is still the same.

This scenario is illustrated in the following diagram:



### Converting between Value and Reference Types

From time to time, you may need to convert between reference and value types. This is particularly useful when using collections (lists, dictionaries, etc.) of mixed types. The term "boxing" refers to the process of converting a value type to a reference type, while "unboxing" is the reverse.

|  |  |
| --- | --- |
| Boxing | Convert a value type to a reference type. This is an implicit conversion and could happen without any indication if a reference type is needed. |
| Unboxing | Convert a reference type to a value type. This needs to be done explicitly. Not all reference types can be cast into values. |

#### Example: Boxing

|  |
| --- |
| int myIntegerVar = 12; object myObjectVar = myIntegerVar;     // Boxing int myOtherIntVar = (int) myObjectVar; // Unboxing |

* The type object is the most basic built-in reference type and can be used to box up any value type.
* Object is a synonym for object from the Intermediate Language (IL). The lower-case version is preferred.

|  |  |
| --- | --- |
|  | Try It Yourself: Type Conversions   1. Open the immediate window using DEBUG > Windows > Immediate.  * To test code without making it part of your solution, you can use the immediate window. This is similar to the VB immediate window, or the M prompt.  1. Type the following: int myIntegerVar = 12; object myObjectVar = myIntegerVar; int myOtherIntVar = (int) myObjectVar; 2. Next, type the following: int myNextInt = myObjectVar; 3. What happened?  * An error occurs because the reference type needs to be explicitly unboxed  1. Close the console that opened during step 2. |

### Integral Types

The integral data types available in C# are listed in the table below. The first column corresponds to C#'s name for each type, while the second column represents the built-in Intermediate Language (IL) data type to which the C# data type maps.

|  |  |
| --- | --- |
|  | Having a built-in set of IL data types allows every language in the .NET Framework to give different names for each data type. |

|  |  |  |  |
| --- | --- | --- | --- |
| C# Type | IL Type | Size (bytes) | Signed? |
| sbyte | Sbyte | 1 | Yes |
| short | Int16 | 2 | Yes |
| int | Int32 | 4 | Yes |
| long | Int64 | 8 | Yes |
| byte | Byte | 1 | No |
| ushort | UInt16 | 2 | No |
| uint | UInt32 | 4 | No |
| ulong | Uint64 | 8 | No |

* The data type of int is the most common integral type, as it is the standard 32 bit value.
* The default value for all integral types is 0.
* The range of values that each type can hold depends on whether or not it is signed. Signed types can hold positive or negative values, ranging from -2^(n-1) to 2^(n-1)-1, where n is the number of bits in the type. Unsigned types can only hold positive values, ranging from 0 to (2^n)-1, where n is the number of bits in the type.
* When representing literals in C#, apply a suffix to the literal value to use a specific type; otherwise the default int type will be used.

|  |  |
| --- | --- |
| Suffix | Type |
| U | uint or ulong |
| L | long or ulong |
| UL | ulong |

|  |
| --- |
| int intOne = 5; // Assumes int default uint intTwo = 10U; // Unsigned int |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Try It Yourself: Integral Types   1. Open the immediate window. 2. Clear it using right click > Clear All 3. Type the following: int myInt = int.MaxValue; 4. What does this value represent? The maximum positive value that any integer may contain. 5. How could you compute it? 2^(32-1)-1 = 2147483647 6. Type the following several times in a row (you can use the up arrow instead of retyping it)  * ++myInt;  1. What happened and why did it happen? The integer overflowed, which switched the sign bit from 0 (positive) to 1 (negative). 2. Fill out the following table using the immediate window:  |  |  |  | | --- | --- | --- | | Type | MinValue | MaxValue | | sbyte | -128 | 127 | | byte | 0 | 255 | | short | -32768 | 32767 | | ushort | 0 | 65535 | | int | -2147483648 | 2147483647 | | uint | 0 | 4294967295 | | long | -9223372036854775808 | 9223372036854775807 | | ulong | 0 | 18446744073709551615 | |

### Floating-Point Types

The floating-point types available in C# are listed in the table below. The C# Type, System Type, and Size columns are equivalent to the columns in the integral types table. The Range column shows how many significant figures each type can hold.

|  |  |  |  |
| --- | --- | --- | --- |
| C# Type | IL Type | Size (bytes) | Range |
| float | Single | 4 | 7 figures |
| double | Double | 8 | 15 figures |
| decimal | Decimal | 16 | 28 figures |

* "float" and "double" carry plenty of precision and should handle most floating-point type needs. Only use "decimal" if absolutely necessary (for example, when representing money).
* The default value for all floating-point types is 0.0
* Much like with integral types, when specifying literals for floating-point types, apply a suffix to force the use of a particular data type. If no suffix is used, the value will default to a double.

|  |  |
| --- | --- |
| Suffix | Type |
| F or f | float |
| D or d | double |
| M or m | decimal |

|  |
| --- |
| double dubOne = 9.1; // Assumes double  float floatOne = 10.813F; // Float |

|  |  |
| --- | --- |
|  | Try It Yourself: Floating-Point Types   1. In the immediate window, examine the max and min values of float, double and decimal. 2. Record the result of each of the following:  * double value1 = 5f; * 5.0, No error * double value2 = 5; * 5.0, No error * double val3 = 5m; * Cannot implicitly convert type 'decimal' to 'double'. An explicit conversion exists (are you missing a cast?) |

### Casting

In C#, it is possible to convert between the different data types. There are two types of casting:

|  |  |
| --- | --- |
| Implicit casting | Conversion to another data type without any special syntax. Usually, this is used to convert from a data type that uses less memory to a data type that uses more (small to large).  int x = 123456;  long y = x;  // Implicit conversion |
| Explicit casting | Conversion to another data type by explicitly declaring the cast in parentheses. Usually, this is required to convert from a data type that uses more memory to a data type that uses less (large to small).  int x = 123456;  short z = (short) x;  // Explicit casting |

If during the cast you are simply adding more bytes to the value, all data is preserved, so an implicit cast is okay. If casting removes bytes that could be in use, data loss is possible, so an explicit cast is required.

|  |  |
| --- | --- |
|  | If the data can't be cast correctly during an explicit cast, an InvalidCastException will be thrown at runtime. Issues with implicit casts will be caught at compile time. |

|  |
| --- |
| int x = 123456; long y = x; // Implicit conversion short z = (short) x; // Explicit conversion |

|  |  |
| --- | --- |
|  | Try It Yourself: Conversions   1. Which of the following require explicit conversions?    1. int to long no    2. sbyte to byte yes    3. float to int yes    4. int to float no    5. decimal to double yes    6. double to decimal yes 2. Verify your answers in the immediate window. |

### Other Types

There are several other data types in C#, including:

|  |  |  |  |
| --- | --- | --- | --- |
| C# Type | IL Type | Size  (bytes) | Default Value |
| char | Char | 2 | '\0' |
| bool | Boolean | 1 | false |
| string | String | 20 minimum | null |
| void | void | N/A | N/A |

#### char

Represents a single Unicode character.

The following kinds of literal characters should be enclosed in single quotes:

* Characters
* Unicode
* Escape characters

|  |  |
| --- | --- |
| Character | 'A' |
| Unicode format | '\u0041' |
| Escape character | '\n' |

The escape characters recognized by C# are:

|  |  |
| --- | --- |
| \' | Single quote |
| \" | Double quote |
| \\ | Backslash |
| \0 | Null |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Horizontal tab |

The escape characters may be used individually or as part of a string.

* Individually: '\\'
* In a string: "In M, the integer division operator is \\"

#### bool

Can be true or false. Is false by default.

#### string

Strings are reference types in C#. They are composed of an immutable sequence of Unicode characters. String literals are enclosed in double quotes. Within the double quotes, you may include any of the escape characters listed above in addition to your regular characters. Optionally, you can prefix a string with the @ symbol, indicating that the string should be read as is (not evaluating escape characters).

|  |
| --- |
| // Equivalent strings string normal = "\\\\server\\fileshare"; string verbatim = @"\\server\fileshare"; |

#### void

A special type that is not a data type used for variables, but instead a return type for a method. If a method does not have a value to return, it will have a void return type. Methods are covered in the Branching section of this lesson.

|  |  |
| --- | --- |
|  | Try It Yourself: What is the type?   1. What is the most appropriate data type for the following values?  * 10 int * 10U uint * 15.605 double * 15F float * '\n' char * "This is not not a string" string * true or false bool  1. Verify each of your guesses using the immediate window. For example, if you guessed that 10 is a float, you can check that using:  10.GetType() == typeof(float);  If it is false, then guess again! |

## Defining Variables and Constants

### Identifiers

Identifiers are used to name entities in the system, such as constants, variables and user defined data types. Identifiers in C# are whole words, starting with a letter or an underscore, and are case sensitive.

The type of data being identified will dictate the case style of the identifier. Following Microsoft rules, the two cases to consider are:

* Camel case: myButton
* Pascal case: MyButton

|  |  |
| --- | --- |
|  | As more structures are introduced, you should ensure that you name them appropriately. See Appendix A - Name Conventions Cheat Sheet in the C Sharp Training Companion for details. |

### Constants

A constant is a value that cannot change. When declaring a constant, the identifier needs to be in Pascal case, so every word starts with a capital letter.

|  |  |
| --- | --- |
|  | Constants are static by default. We'll cover what this means in a later lesson. |

|  |
| --- |
| Syntax: const type identifier = value;  Example: const float TaxRate = 5.5F; |

|  |  |
| --- | --- |
|  | Try It Yourself: Constants  Constants cannot be defined in the immediate window, because they must be part of a class, so we'll add a constant to your ClassExamples project.   1. Notice that in Program.cs there is a class named Program defined. Add a constant to that class:  * const string YouCannotChange = "Balrog";  1. On the first line of code in Main, set a break point by clicking in the grey margins to the left of the line. You should see a red dot appear. 2. Run the solution. 3. Open the Immediate window.  * Because you ran your solution and set a breakpoint prior to opening the immediate window, you now have access to the context of the code that is currently executing. This is a nifty debug trick!  1. Type in the constant name:  * YouCannotChange  1. What happened?  * Displays "Balrog"  1. Try setting the constant to another value  * YouCannotChange="Gandalf";  1. What happened?  * An error is thrown: "The left-hand side of an assignment must be a variable, property or indexer"  1. Clear the breakpoint that you set earlier by clicking on the red dot in the margin. |

### Local Variables

Local variables are variables defined in methods, either in the parameter list or inside the method. To define a local variable, you need to indicate the data type and variable name. Local variable names should be in Camel case, meaning the first letter in the word in the variable name is lower case, while the rest of the words begin with an uppercase letter. Optionally, an initial value assignment can be included on the same line as the declaration.

|  |
| --- |
| Syntax:  type varName [ = value];  Examples: int patientAge; string patientName = "Sickly, Sam"; |

|  |  |
| --- | --- |
|  | You must assign a value to local variables before using them, or your code will not compile. |

You can initialize local variables to their defaults, but you need to do so explicitly:

|  |
| --- |
| int myVar = default(int);  // This is equivalent to assigning 0 |

|  |  |
| --- | --- |
|  | Although local variables do not have an initial state, other kinds of variables do. For example:   * Static fields * Class instance fields * Array elements   We will cover these kinds of variables in later lessons. |

|  |  |
| --- | --- |
|  | Try It Yourself: Local Variables   1. In ClassExamples > Program > Main, define a new local variable:  * string myString; * myString = myString + " abcd";  1. Build the solution using BUILD > Build Solution (or by pressing F7). What happens?  * Error: Use of unassigned local variable 'myString'  1. Initialize myString to an empty string  * string myString = "";  1. Build the solution again. What happens this time?  * Builds correctly  1. Use the immediate window and default(<type>) to discover the default value of each of the following types:    1. string null    2. double 0.0    3. bool false    4. object null |

### Activity: Create a Simple Console Application

In this exercise you will learn how to write a new console-based program.

#### Part 1: Start a New Project

The first step to create a new console application is creating an executable project.

1. Start Visual Studio
2. Create a new project

* File > New > Project

1. Console Application
2. Uncheck Create directory for solution
3. Name: SimZoo
4. Location: C:\EpicSource\C Sharp
5. Click OK

|  |  |
| --- | --- |
|  | Take a look at your solution in the Solution Explorer (sidebar).  What is a solution? A solution is an organized way of viewing a group of projects. Usually, a solution will group projects that depend on each other.  A project is simply a collection of files. These files, when compiled with the C# compiler, will collectively produce a single portable executable (e.g. a file with a .dll or .exe extension) and perhaps some other files used by it. In the case of your program, it will produce a .exe file that can be run when double-clicked on.   1. Add a write statement to the console and a Console.ReadKey into your Main method. 2. Compile your project. 3. Try navigating to the Debug folder of your project:    1. Solution Explorer > SimZoo (project) > right click > Open folder in file explorer    2. Open bin > Debug 4. Double click SimZoo.exe.   Your program runs! This executable is the output produced when your project compiles. If you had multiple projects in your solution, multiple portable executables would be produced. |

1. Edit the project properties:
   1. Solution Explorer > SimZoo (project) > right click > Properties



Solution Explorer

1. Change the default namespace to "Epic.Training.SimZoo.Text"

* Application Tab > Default namespace > Epic.Training.SimZoo.Text

|  |  |
| --- | --- |
|  | If you use CodeSearch to search client code for "class Program" (the text at the top of your code), you'll find numerous C# files that also define Programs. Many of these may be included together in the same application (e.g. Hyperspace). If that happens, how does the application know which one to use?  The namespace of your project helps to keep the code in your project logically distinct from code written by other teams at Epic, and even other companies, such as Microsoft.  A class you write (like Program) might have the same name as other classes written by other people. But because namespaces are long and have many pieces, the chances of two classes sharing the same name and the same namespace are very small.  Namespaces are useful for another reason, too. By making namespaces descriptive and including many pieces, they can aid in organizing large collections of code.  At Epic, we use the following convention for namespaces to help organize and keep our code distinct:  Epic.<Owner>.<Application>.[Functional area].[Platform]   * Epic: This prefix separates our code from that of third parties * Owner: The division owning the code. * Examples: Core (Foundations), Clinical, Billing, Access, etc. * Application: The application within the division that this code belongs to. * Examples in Clinical: Ambulatory, Inpatient, OR, Orders, etc. * Functional area (optional): A sub-division within the application. * Examples in Clinical > Orders: Administration, Data, Services * Platform (optional): The medium on which the code will run. * Examples: Text, Web, WinForms, WPF   Notice that just by looking at the namespace, you can already tell the context in which a piece of code will be used. |

1. What is each part of code for the project that you just created?
   1. Owner: Training
   2. Application: SimZoo
   3. Platform: Text
2. Save your changes and close the application properties.

#### Part 2: Create a simple text interface

1. Remove the "Naming Conventions Cheat Sheet" from the following section so that you can use it while writing code.
2. Edit Program.cs
3. Change the namespace from "SimZoo" to "Epic.Training.SimZoo.Text":

|  |
| --- |
| namespace Epic.Training.SimZoo.Text |

1. Using the Console methods that you learned earlier in this lesson, display the message "<Press any key to continue>" and then pause execution until the user presses any key.
2. Define constants within the Program class to store your first and last name (as the author):

|  |
| --- |
| class Program {   const string AuthorFirstName = "<your name>";   const string AuthorLastName = "<your name>";   ... |

1. Before displaying the "<Press any key to continue>" message, output the following:

* "SimZoo Text v. 1.0"
* "Author: {1}, {0}", where "{0}" is your first name and "{1}" is your last name.

1. Save and run the application to verify that it works as expected.
2. After the"<Press any key to continue>" message,

* Clear the screen using the method Clear from the Console class.

1. Ask the user to enter a kind of animal to include in the zoo (as a string).
2. Ask the user how many of that kind of animal should live in the zoo (as an int).
3. Display the animal name and number of animals back to the user.
4. Pause the screen a second time.
5. Save and test your code.
6. Document your code.

* Use single line comments to describe what is going on in main
* Use XML comments to document the following:
* The Program class
* Each constant
* The main method

1. Define a region called "class constants" around your constants

#### If you have time

Try using codesearch to find sample code from the new web framework.

1. http://codesearch/ > Client Code
2. File name ends with: "cs"
3. Try the following search Strings:
   1. namespace Epic.Core
   2. namespace Epic.Billing
   3. namespace Epic.Clinical

|  |  |
| --- | --- |
|  | Keep in mind that you have not learned enough to fully understand everything that is going on. |

#### Wrap Up

1. What is the name convention for local variables?

* camelCase

1. What is the name convention for constants?

* PascalCase

1. What is the purpose of a namespace?

* To logically divide different categories of code

1. Number the following in the order that they appear in the namespace

* Platform: 5
* Epic: 1
* Application: 3
* Owner: 2
* Functional Area: 4

1. Which parts of the namespace are optional?
   1. Platform
   2. Functional Area

## Defining Sets of Constants

Enumerations are special types that only allow a finite and discrete set of constant values. Defining enumerations, rather than just using a range of integer values, makes the code easier to read because there is an identifier associated with each value. Additionally, it provides hints to the compiler as to what options are available for variables of the given type.

|  |  |
| --- | --- |
|  | An enumeration is similar to a Chronicles category list. |

|  |
| --- |
| Syntax [modifiers] enum EnumTag {         enumerator-list }  Example enum Size {         Small,      // Assumes a value of 0         Regular,    // Assumes a value of 1         Large = 5,         Gigantic,   // Assumes a value of 6 }  Syntax to use an enumeration: enumTag varname [ = enumTag.member];  Example  ExampleSizes shirtSize = Sizes.Gigantic; |

An enumeration consists of a set of named constants. In C#, enumerations are classified as value types. An enumeration type declaration gives the name of what is called the "enumeration tag" and defines the set of named constants. Enumeration tags should be in Pascal case, and may be defined in a namespace or in a class. If an enum is defined in a class, the enum values will be accessed through that class (similar to constants).

|  |  |
| --- | --- |
|  | Define an enum in a namespace if you want to make that enum available to many classes. If the enum is only useful for a particular class, define the enum inside that class. |

The keyword enum is used when defining an enumerated type. Commas separate individual members of the enumeration list, and the members can be assigned a specific value, or a value will be automatically calculated by incrementing the previous value.

Here is another example, where each tag is assigned a specific value:

|  |
| --- |
| enum Temperature {    Cold=0,    FreezingPoint = 32,    LightJacketWeather = 60,    SwimmingWeather = 80,    BoilingPoint = 212, } |

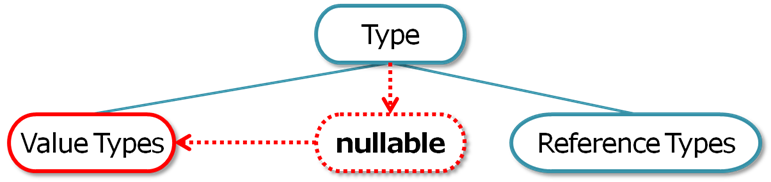
|  |  |
| --- | --- |
|  | Try It Yourself: Enumerations   1. In your ClassExamples project, create an enumeration of various kinds of fruits, just inside the namespace:   enum Fruit   {     Apples,     Bananas,     Cranberries,     Durian,   } 2. Set a breakpoint on the first line in the Main method. 3. Run your solution. Execution should stop at your breakpoint. 4. In the immediate window, type Fruit myFavorite = Fruit and then a period. What happened?  * There is a popup that lists the various options  1. Select Apples, enter a semicolon and then press enter 2. Cast the variable myFavorite to its int equivalent: (int)myFavorite 3. What is the output?  * 0  1. Stop execution and remove the breakpoint. 2. Modify your Main method by adding the following: |

|  |
| --- |
| string fruitOptions = string.Join(",", Enum.GetNames(typeof(Fruit))); Console.Write("Enter a fruit ({0}):", fruitOptions); string intput = Console.ReadLine(); Fruit myFruit = (Fruit)Enum.Parse(typeof(Fruit), intput); Console.WriteLine("You chose {0}", myFruit); |

|  |  |
| --- | --- |
|  | Try It Yourself: Enumerations (Continued)   1. Run it a few times. 2. Try entering in both the names of fruit as well as numbers inside and outside the range 0-3 3. Does entering an invalid string throw an exception?  * Yes  1. Does entering an invalid number throw an exception?  * No.  1. Describe what each line of code above does    1. Creates a string of the various fruit options    2. Writes the options to the screen    3. Reads in the user's selection    4. Converts the input from a string to a Fruit    5. Outputs the chosen fruit |

## Allowing Empty Value Types

There are times where it might be useful to represent the absence of information within a value-type. By default, this isn't possible, but it can be done if by using a nullable value type rather than the normal kind.



|  |  |
| --- | --- |
|  | Why use nullable types?  Think about a patient's age.   * What does 0 (the default for int) mean? * The patient is a newborn * What does a null value mean? * We don't know the patient's age |

### Syntax

To define:

T? variable

To check for a value:

  if (variable != null){ ... }

or

  if (variable.HasValue){ ... }

|  |
| --- |
| int? age = null; if (age == null) {     Console.WriteLine("Age unknown"); } age = 25; if (age.HasValue) {     Console.WriteLine("Now the age is {0}", age); } |

|  |  |
| --- | --- |
|  | You can also use System.Nullable<T> variable to define a nullable type, which is the full name of the class. Typically the abbreviated definition of T? is preferred. |

### Casting Nullable types

One tricky concept is that a value type, T, and its nullable equivalent, T?, are not the same because T? can be null but T cannot.  This means that T can be implicitly cast to T? but not the other way around.  If you cast a T? to T and the value of the T? is null, you will encounter an invalid-cast exception.

A safer way to cast is to use the ?? operator:

* valueType = nullableType ?? default;

If nullableType is null, then the default value is assigned, which avoids the invalid-cast exception.

|  |
| --- |
| int? age;  int knownAge = age; //will not compile int knownAge = (int)age; //fails at runtime if age is null int knownAge = age ?? -1; //assigns -1 if age is null int knownAge = age ?? default(int); //assigns 0 if age is null |

|  |  |
| --- | --- |
|  | Try It Yourself: Nullable types   1. Open the Immediate window. 2. Type: int knownAge = null;  * What is the result? Cannot convert null to 'int' because it is a non-nullable value type  1. Type: int? age = null; 2. Try comparing age against null: age == null  * What is the result? true  1. A test that is arguably easier to read is: age.HasValue  * What is the result? false  1. Type: age=25; 2. What is the result of each test?  * age == null false * age.HasValue true  1. Type: knownAge = age;    1. What is the result? Cannot implicitly convert type 'int?' to 'int' 2. Type: knownAge = age ?? default(int);    1. What is the result? 25 |

## Using Operators

C# supports all the mathmatical operators that you expect to find in a programming language.

### Math Operators

|  |  |
| --- | --- |
| \* | Multiplication |
| / | Division |
| % | Modulus |
| + | Addition |
| - | Subtraction |

The return value for division depends on dividend and divisor.

#### Assignment

There are several operators that combine a mathmatical expression and assignment together.  These provide shorthand that can make your code less verbose.

|  |  |
| --- | --- |
| \*= | Multiply-assign  X \*= Y; is equivalent to X = X \* Y; |
| /= | Divide-assign  X /= Y; is equivalent to X = X / Y; |
| %= | Modulo-assign  X %= Y is equivalent to X = X % Y; |
| += | Add-assign  X += Y; is equivalent to X = X + Y; |
| -= | Subtract-assign  X -= Y; is equivalent to X = X - y; |

#### Increment and Decrement Operators

The increment and decrement operators are useful when iterating through sets of indexed data.  They are used to increase or decrease teh value of an integral type by one, either before or after the surrounding expression is evaluated.

|  |  |
| --- | --- |
| ++ | Increment |
| -- | Decrement |

|  |
| --- |
| Examples: value++;  // Increments by 1 AFTER use value--;  // Decrements by 1 AFTER use ++value;  // Increments by 1 BEFORE use --value;  // Decrements by 1 BEFORE use |

|  |  |
| --- | --- |
|  | Try It Yourself  What is the output? int x = 2; int y = 2; Console.WriteLine("x = {0}",x++);2  Console.WriteLine("y = {0}",++y);3 |

### Relational Operators

Relational operators are used to compare two expressions and returrn a bool; true or false.

|  |  |
| --- | --- |
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |

### Logical Operators

Logical operators are used to compare two Boolean expressions and return true or false.

|  |  |
| --- | --- |
| && | And |
| || | Or |
| ! | Not |

* C# will short circuit.
* Example:  1 == 2 && PerformanceIntensiveMethod()will evaluate to false before performing the performance intensive method because the result of the && operation will always be false if the first operand is false.

|  |  |
| --- | --- |
|  | C# also supports Bit Manipulation   * << Shift bits left * >> Shift bits right * &  Bitwise and * ^  Bitwise exclusive or * |  Bitwise or * ~  Bitwise not |

|  |  |
| --- | --- |
|  | To form boolean expressions with nullable boolean variables, the bitwise operators must be used. Because null is a non-determinant state, the results are slightly different when one or both of the values in the boolean expression are null:   * true  & null = null * false & null = false * null  & null = null * true  | null = true * false | null = null * null  | null = null |

### Full Order of Operations

The order of operations starts with the operators on the first line (Unary) and then works its way down to the lowest line (Assignment).

Operators on the same level are processed from left to right unless otherwise specified. For example, in the expression 2 % 3 \* 2, the mod happens first beceause they have equal presidence. Parenthesis can be used to adjust the order that operators are evaluated.

|  |  |
| --- | --- |
| Unary | +      -      ! |
| Multiplicative | \*      /      % |
| Additive | +      - |
| Bit Shift | <<     >> |
| Relational | <      >      <=      >= |
| Equality | ==     != |
| Bitwise logicals | &  then ^  then | |
| Conditional logicals | && then || |
| Assignment | =      \*=      /=      %/      +=      -= |

|  |  |
| --- | --- |
|  | Try It Yourself: Expressions   1. Evaluate the following expressions:    1. 6 + 2 \* 3 - 4/2      10    2. 3 \* 4/2 + 3 - 1      8    3. 6 + 2 \* (3-4)/2      5    4. 3 \* 4/ (2 + 2) - 1   2    5. (1 < 3) && (9 > 3)   true    6. (3 > 4) || (1 < 2)   true    7. (9 != 1)             true    8. !(3 == 2)            true 2. Verify your work using the immediate window |

## Branching

Code branching is essential to building robust programs. Unconditional branching promotes code reuse and maintainability, while conditional branching allows you to execute the right code at the right time.

In this section, you will learn how to perform both types of branching.

### Unconditional Branching

Unconditional branching occurs when you call a method. Execution will jump to the method and will return to the calling code once the method is complete.

To call a method (i.e., branch):

* Add parenthesis after the method name
* Use dot notation to call methods in other classes

The method is finished when:

* The end of the method is reached
* The keyword return is encountered (with or without an expression, depending on if the method return type is void or not)

|  |
| --- |
| Defining a method: static string PromptUserForInput(string prompt) {   Console.Write(prompt);   return Console.ReadLine(); }  Calling methods:  // In the same class string name = PromptUserForInput("Enter your name: "); // In a different class Console.WriteLine("Your name is {0}", name); |

|  |  |
| --- | --- |
|  | Try It Yourself: Unconditional Branching   1. In your ClassExamples project, highlight the code that you use to read a fruit from the console. Do not include the line that writes the output to the console. In other words, highlight only these lines:  * string fruitOptions =    string.Join(",", Enum.GetNames(typeof(Fruit))); Console.Write("Enter a fruit: ({0}):", fruitOptions); string intput = Console.ReadLine(); Fruit myFruit = (Fruit)Enum.Parse(typeof(Fruit), intput);  1. Choose: right click > Refactor > Extract Method  * Refactoring is the process of changing the structure of code without changing its function. The purpose is usually to make it more modular and extensible.  1. Give the new method the name GetFruit 2. The method that was created should be:  * private static Fruit GetFruit() {   string fruitOptions =      string.Join(",", Enum.GetNames(typeof(Fruit)));   Console.Write("Enter a fruit: ({0}):", fruitOptions);   string intput = Console.ReadLine();   Fruit myFruit = (Fruit)Enum.Parse(typeof(Fruit), intput);   return myFruit; }  1. The location where the code was originally located should now be:  * Fruit myFruit = GetFruit(); Console.WriteLine("You chose {0}", myFruit);  1. Place a breakpoint on the line where GetFruit is called and assigned to myFruit. 2. Run the application 3. Press F11 to step into the method 4. Press F10 to step through a few lines. 5. Hover the mouse over various variables to see what the values are. You can also see all local variable values that are defined in the Locals window (DEBUG > Windows > Locals, or ALT + 4) 6. To run execution until you return from the current method call, press SHIFT + F11 7. Press F5 to run to completion. 8. Clear the breakpoints |

#### Reusing Method Names

|  |  |
| --- | --- |
|  | Why?  Why reuse method names?   * If two methods do essentially the same thing, it would be confusing to use different names. |

Possible if the signature (parameter list) is different. This feature is known as overloading.

|  |
| --- |
| //A method to drive a car  public double DriveCar(double miles)  {   \_milage += miles;   return \_mileage; }  //Another method to drive a car that uses different information  public double DriveCar(double averageSpeed, double time) {   return Drive(averageSpeed \* time / 60.0); } |

When code calls a method, the method that is called is the one whose signature matches the arguments passed into the method.

|  |  |
| --- | --- |
|  | Try It Yourself: Reusing Method Names   1. In your ClassExamples project, hold down CTrl and click on Writeline in Console.WriteLine("Hello world!"). This will take you to the definition of  Console.WriteLine. 2. How many different methods of the Console class are named WriteLine?  * 19  1. Notice how they each method takes in different input. 2. Which method did you call when you wrote Console.WriteLine("Hello world!")?  * public static void WriteLine(string value)  1. Add a new line of code that says Console.WriteLine(52.3); 2. Which method will be called this time?  * public static void WriteLine(double value);  1. Verify your answer by going to the definition via ctrl + Click. Notice which line the cursor defaults to. |

|  |  |
| --- | --- |
|  | In Visual Studio, you can use ctrl + Click to go to the definition of a method. The keyboard shortcut for this is f12. If the code is defined in the same assembly (project), you'll see the full definition. You can try this with GetFruit. If the method is defined in a different assembly, Visual Studio does not have access to the source code. This was the case with Console.WriteLine. Instead, it will pull the metadata from the manifest so you can see the XML documentation and the method signature. |

#### Passing Data with Parameters

By default, passing parameters into a method is input only (passed by value). But there are other ways to pass parameters as well.

* For output parameters use the out keyword
* For input/output parameters use the ref keyword

You'll need to include the ref/out keywords both when defining the method and when calling it.

|  |
| --- |
| //This method uses the return value to inform whether the car succeeded in driving or not. It also passes back the gas required to drive the car via an output parameter.  public bool TryDriveCar(double milesDesired, out double gasRequired)  {   gasRequired = milesDesired / MilesPerGallon;   if (gasRequired <= \_gallonsOfGas)    {     Drive(milesDesired);     return true;   }   return false; }  //Use the out keyword when calling the method also  double gasNeeded;  bool hasDriven = TryDriveCar(100, out gasNeeded)  if (hasDriven)  {    Console.WriteLine("The car drove 100 miles and used {0} gallons of gas.", gasNeeded);  }  else  {    Console.WriteLine("The car needs {0} gallons of gas to drive that far.", gasNeeded);  } |

The ref keyword works very similarly to the out keyword. The expectation is that a method using out will not use the data passed in. A method using ref will use the data passed in.

|  |  |
| --- | --- |
|  | You don't need to pass reference type variables by ref! Since a reference type variable is just a pointer to the heap, the pointers for both the argument and the parameter (the variable passing the data and the variable receiving the data) will point to the same spot on the heap. Change the data on the heap, and both pointers will point to the changed data.  Passing a reference type variable by ref means something very different and is only done in rare circumstances. It means that the object that was passed into the method may not be the same object that gets passed out of the method. Remember that a reference type variable just contains a memory address. If you pass a memory address by ref, that memory address can be changed to point to a different spot on the heap (a completely different object). |

### Conditional Branching

Conditional branching occurs when a segment of code may or may not be executed, depending on the result of a Boolean expression. There are two types of conditional branches: If/else and switch.

#### The If/Else Statements

The if statement allows you to conditionally execute one or more lines of code based on the result of a Boolean expression. The else statement allows you to conditionally execute code when the previous if statement evaluates to false.

There are two forms of the if statement:

|  |
| --- |
| // One statement executed when expression is true if (Boolean-expression) statement;  // All statements within block are executed when expression is true if (Boolean-expression) {   statement;    ... } |

At Epic, we will always use blocks of code after an if statement. In other words, always include the braces: {}. An else is optional and should only be used when it makes sense to do so.

|  |
| --- |
| if (value1 > value2) {   Console.WriteLine("First is bigger"); } else if(value1 < value2) {   Console.WriteLine("Second is bigger"); } else {   Console.WriteLine("Values are equal"); } |

|  |  |
| --- | --- |
|  | Use the snippet switch to generate the basic code for an if statement.  The if code snippet also comes with a field (true). When there are multiple fields, you can tab through to populate them. Try these snippets:  if:     if (true)     {     }  else:     else     {     } |

|  |  |
| --- | --- |
|  | Try It Yourself   1. In the ClassExamples project, after reading in a fruit, add an If/else statement:  * if(myFruit == Fruit.Apples) {   Console.WriteLine("I also like them apples!"); } else {   Console.WriteLine("I wish we all liked them apples..."); }  1. Test the code to make sure it works as expected. |

#### The Switch Statement

The switch statement allows code to branch to one of several different options depending on the value of a single expression. The value is compared against a set of constant-expressions.

* bool
* char, string
* int, uint, long, ulong
* enum
* Nullable version of above types

A default case is also allowed, which is a catch all for values not explicitly given their own case. The default is not required.

|  |
| --- |
| Syntax:  switch(expression) {   case constant-expression1:     statements;     break;   case constant-expression2:     statements;     break;   ...   [default:     statements;     break;] } |

|  |  |
| --- | --- |
|  | Use the snippet switch to generate the basic code for a switch statement:    switch (switch\_on)   {     default:   } |

* Use break; to end a case

|  |  |
| --- | --- |
|  | * Use goto default; to jump to the default case from elsewhere in the switch statement. * You can group cases together to execute same code:  switch(expression) {   case const-expr1:   case const-expr2:     // const-expr1 and const-expr2 fall here     statements;      break; } |

|  |
| --- |
| Random random = new Random(); //Number gen int roll = random.Next(1, 20); switch (roll) {     case 20:         Console.WriteLine("Critical hit! Roll again to confirm...");         break;     case 1:         Console.WriteLine("You swing too hard and dislocate your shoulder.");         break;     default:         Console.WriteLine("Do the math.");         break; } |

|  |  |
| --- | --- |
|  | Try It Yourself: Conditional Branching   1. In the ClassExamples project, comment out the if/else statement. 2. Replace it with an equivalent switch statement:  * switch (myFruit) {   case Fruit.Apples:     Console.WriteLine(       "I also like them apples!");     break;   default:     Console.WriteLine(       "I wish we all liked them apples...");     break; }  1. Modify the switch so there is a specific message of disgust for each non-apple fruit, but they all call into the default after (other than apples). For example:    1. switch (myFruit) {   case Fruit.Apples:     Console.WriteLine(       "I also like them apples!");     break;   case Fruit.Bananas:     Console.WriteLine(       "Bananas? Really? Looks like you slipped up.");     goto default;   default:     Console.WriteLine(       "I wish we all liked them apples...");     break; } |

## Looping

Looping is another basic task that a programmer must perform in all programming languages. Like branching, it promotes code reuse.

For example, if you need to read in 100 strings from the console, you can write the code once and then repeat it 100 times with a looping construct, rather than copying the code 100 times.

Also, certain tasks are impossible without the ability to repeat some code an indeterminate number of times.

C# has three looping constructs:

* The While Loop
* The Do-While Loop
* The For Loop

### The While Loop

A while loop executes code within the loop while a condition is true. The Boolean expression is tested before the loop executes. Because the test happens before the code is executed, the code may not execute at all.

|  |
| --- |
| Syntax:  while (Boolean-expression) {   //or statement block } |

|  |  |
| --- | --- |
|  | Use the snippet while to generate the basic code for a while loop:    while (true)   {   } |

|  |
| --- |
| Example: int number; Console.Write("Enter an integer: "); while (!int.TryParse(Console.ReadLine(), out number))  {   Console.WriteLine("That is not an integer.");   Console.Write("Enter an integer: "); } |

|  |  |
| --- | --- |
|  | Try It Yourself: While Loops   1. In your ClassExamples project, modify GetFruit so that it prompts the user for a fruit over and over again until a valid fruit is selected.  * Hint 1: Use the Enum.TryParse<Fruit>(string,out parsedFruit) method along with a while loop.  1. If you get stuck, examine the following code example. |

|  |
| --- |
| private static Fruit GetFruit() {    Fruit myFruit = default(Fruit);   bool isValid = false;   string fruitOptions = string.Join(",", Enum.GetNames(typeof(Fruit)));   Console.Write("Enter a fruit ({0}):", fruitOptions);    while (!isValid)   {     if(Enum.TryParse<Fruit>(Console.ReadLine(), out myFruit))     {       isValid = Enum.IsDefined(typeof(Fruit), myFruit);     }     if(!isValid)     {       Console.WriteLine("That is not a Fruit.");       Console.WriteLine("Enter the name of a fruit exactly as listed");       Console.Write("Enter a fruit ({0}):", fruitOptions);     }   }   return myFruit;  } |

In general, TryParse is a safer form of Parse. Rather than throwing an exception if the conversion fails, it simply returns false. If the conversion succeeds, then TryParse returns true, and the second argument (marked with out) returns the parsed value.

|  |  |
| --- | --- |
|  | The method Enum.TryParse<T>(string, out enumVal) is an example of a generic method because it uses the <T> syntax as part of the method name. The type that is parsed and returned through enumVal is specified using the T generic parameter when the method is called.  Generic methods will be covered more in a later lesson. |

|  |  |
| --- | --- |
|  | One limitation of both Enum.Parse and Enum.TryParse is that they permit integer results when a number that is not defined in the enumerated type is entered.  To filter out integers that are outside the range, pass the resulting (parsed) enum value into the Enum.IsDefined method:  EnumType parsedValue; string input = Console.ReadLine(); bool isValid = false;  if(Enum.TryParse<EnumType>(input, out parsedValue)) {   isValid = Enum.IsDefined(typeof(EnumType), parsedValue); } |

### The Do-While Loop

A Do-While Loop executes code within the loop while a condition is true. The Boolean expression is tested after the loop executes, so the code in the loop will always execute at least one time.

|  |
| --- |
| Example:  do {   //statement block } while (Boolean-expression); |

|  |
| --- |
| Example:  int number; do  {   Console.Write("Enter an integer: ");   if (int.TryParse(Console.ReadLine(), out number)) {     break; //Exit the loop   }   Console.WriteLine("That is not an integer."); } while (true); |

|  |  |
| --- | --- |
|  | Use the snippet do to create the basic code for a do-while loop:    do   {   } while (true); |

|  |  |
| --- | --- |
|  | Try It Yourself: Do-While Loops  Rewrite the fruit validation loop in the ClassExamples project so it uses a do-while loop instead of a while loop. If you get stuck, examine the following code example box. |

|  |
| --- |
| private static Fruit GetFruit() {   string fruitOptions = string.Join(",", Enum.GetNames(typeof(Fruit)));    Fruit myFruit = default(Fruit);   bool isValid = false;   string input = null;    do   {     if (!string.IsNullOrWhiteSpace(input))     {       Console.WriteLine("That is not an Fruit.");       Console.WriteLine("Enter the name of a fruit exactly as listed");     }     Console.Write("Enter a fruit: ({0}):", fruitOptions);     input = Console.ReadLine();      if(Enum.TryParse<Fruit>(input, out myFruit))     {       isValid = Enum.IsDefined(typeof(Fruit), myFruit);     }   } while (!isValid);    return myFruit; } |

### The For Loop

A For Loop is an iterative-style looping construct. The Boolean test is performed before the loop executes.

* All parts of for are optional
* continue; jumps to the next iteration

|  |
| --- |
| Syntax:  for(initializers; Boolean-expression; iterators) statementOrBlock; |

|  |  |
| --- | --- |
|  | Use the snippet for to create the basic code for a for loop that iterates over the length of a collection from smallest to largest index:    for (int i = 0; i < length; i++)   {   }  Use the snippet forr to create the basic code for a for loop that iterates over the length of a collection in reverse, from largest to smallest:    for (int i = length - 1; i >= 0; i--)   {   } |

|  |
| --- |
| Example:  for (int counter=0; counter<10; counter++) {    Console.WriteLine("counter: {0}", counter); } |

|  |  |
| --- | --- |
|  | If you need to track multiple variables at once, separate multiple initializers and iterators with commas:    for(int i=0, j=0; i < 10; i++, j++); |

|  |  |
| --- | --- |
|  | Try It Yourself: For Loops  Since Enum.TryParse<Fruit> will accept both the name and index of a fruit, it would be better to include the index in the list of options. This is easy to do using a for loop.   1. Comment out this line of code: string fruitOptions =    string.Join(",", Enum.GetNames(typeof(Fruit))); 2. On the next line, define an array of strings:  * string [] fruitOptionsArray = Enum.GetNames(typeof(Fruit)); * Note: We will cover arrays in more detail in a later lesson.  1. Using the snippit, create this for loop to iterate through all the options: for (int optionIndex = 0; optionIndex < fruitOptionArray.Length; optionIndex++) { } 2. In the for loop, prepend the index of each string in front of the name: fruitOptionArray[optionIndex] = string.Format("[{0}] {1}",    optionIndex, fruitOptionArray[optionIndex]); 3. On the next line, join the array elements together into a comma-delimited string:  string fruitOptions = string.Join(",",fruitOptionArray); 4. Test your code to ensure it works as expected 5. The GetFruit method is looking a bit cluttered. Extract code that creates the fruit options string to a separate method called GetFruitOptions. 6. Examine the following code example for the solution. |

|  |
| --- |
| private static Fruit GetFruit() {   string fruitOptions = GetFruitOptions();   Fruit myFruit = default(Fruit);   bool isValid = false;   string input = null;    do   {     if (!string.IsNullOrWhiteSpace(input))     {       Console.WriteLine("That is not an Fruit.");       Console.WriteLine(         "Enter the name of a fruit exactly as listed");     }      Console.Write("Enter a fruit: ({0}):", fruitOptions);     input = Console.ReadLine();      if (Enum.TryParse<Fruit>(input, out myFruit))     {       isValid = Enum.IsDefined(typeof(Fruit), myFruit);     }   } while (!isValid);   return myFruit; }  private static string GetFruitOptions() {   string[] fruitOptionArray = Enum.GetNames(typeof(Fruit));   for (int optionIndex = 0; optionIndex < fruitOptionArray.Length; optionIndex++)   {     fruitOptionArray[optionIndex] =        String.Format("[{0}] {1}", optionIndex, fruitOptionArray[optionIndex]);   }   string fruitOptions = string.Join(",", fruitOptionArray);   return fruitOptions; } |

|  |  |
| --- | --- |
|  | Try It Yourself: Generic Methods  If you have time and with a little extra work, you could get this code to generate the list of menu options for any enum, not just the fruit enum.  This is an advanced exercise, so try it out if you're interested.   1. Investigate the definition of Enum.TryParse by right clicking on it and selecting Goto definition (You can also arrow the cursor to it and then press F12) 2. Copy the definition that it jumps to and paste it into your Program class as a new method 3. Modify the method definition as follows:    1. Change the return type from bool to string    2. Change the name from TryParse to GetEnumOptions    3. Remove the parameters    4. Replace the terminating semicolon with a code block.    5. Paste the code from GetFruitOptions into the code block of GetEnumOptions    6. Everywhere that the type Fruit is used, use TEnum instead. This represents the type passed in by the user.    7. Change the variable names to not include the word "fruit". There is a rename option in the refactor menu (right-click > Refactor).    8. Compare your code to the following example |

|  |
| --- |
| public static string GetEnumOptions<TEnum>() where TEnum : struct {   string[] optionArray = Enum.GetNames(typeof(TEnum));   for (int optionIndex = 0;         optionIndex < optionArray.Length; optionIndex++)   {     optionArray[optionIndex] =        String.Format("[{0}] {1}", optionIndex, optionArray[optionIndex]);   }   string options = string.Join(",", optionArray);   return options; } |

|  |  |
| --- | --- |
|  | The where keyword places restrictions on the kinds of types that can be used in a generic method. In this case, we are forcing the class specified to be a value type using the struct keyword. We'll cover structs in more detail in a later lesson. |

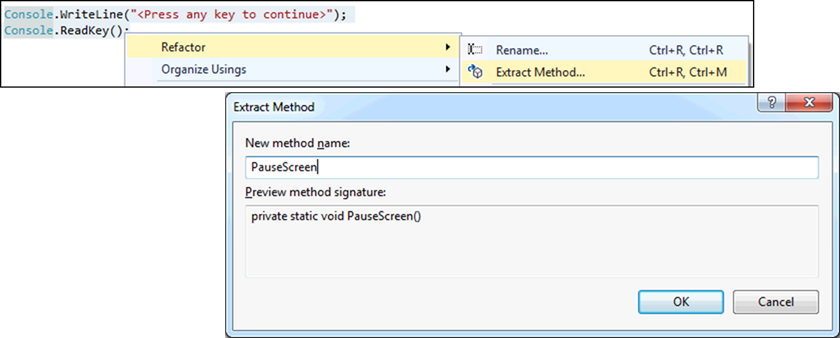
### Activity: Validating User Input

In this exercise you will use branching and looping to validate user input in your SimZoo application.

1. You already have code that informs the user to "Press any key to continue" and then pauses execution. Rather than repeating both lines every time you want to pause, create a custom method in the Program class to do this.

* Use the refactoring option to do this.

Path: Highlight code > right click > Refactor > Extract method



Refactoring to extract a method

|  |
| --- |
| /// <summary> /// Pause the screen until a key is pressed /// </summary> private static void PauseScreen() {   Console.WriteLine("<Press any key to continue>");   Console.ReadKey(); } |

1. Use the PauseScreen method in place of the original two lines everywhere they were used.
2. Repeat the process that you used to create the PauseScreen method to create a new method called ShowSplashScreen. It should include the following lines of code:

|  |
| --- |
| Console.WriteLine("SimZoo Text v. 1.0"); Console.WriteLine("Author: {1}, {0}", AuthorFirstName, AuthorLastName); PauseScreen(); Console.Clear(); |

1. Call the ShowSplashScreen method rather than calling the included lines of code directly.
2. Create another method called GetAnimal. This method will be slightly different because it will have two output parameters:

|  |
| --- |
| /// <summary> /// Get the name of a type of animal and the number /// of that type of animal to add to the zoo. /// </summary> /// <param name="animalName">The entered animal name</param> /// <param name="animalCount">The number of that animal</param> private static void GetAnimal(out string animalName, out uint animalCount) {   // Get the animal name   ...   // Get the number of animals   ... } |

1. Change GetAnimal so that it only allows non-null animal names

* Use a do-while loop that stops looping after a non-null name has been entered
* To verify that the name isn't null or only consists of white space, use the method IsNullOrWhitespace from the string class. It returns a Boolean.

1. Change GetAnimal so that the entered number of animals parses without throwing an exception and only allows positive integers

* Use a while loop for this.
* What kind of data type only allows positive numbers? uint
* Use that data type to call the TryParse method.

1. The next step is to create a menu for the user to choose options from. You will use an enumerated type to collect the various end-user options in a convenient location.

* Create an enumerated type of menu options, including one for "Add" and one for "Quit". Be sure to place it above the Program class, just inside the namespace declaration:

|  |
| --- |
| namespace Epic.Training.SimZoo.Text {     /// <summary>     /// Enumeration of menu options     /// </summary>     enum MenuOption     {         /// <summary>         /// Add an animal to the zoo         /// </summary>         Add,         /// <summary>         /// Exit SimZoo         /// </summary>         Quit     } |

1. You need to create a menu loop to repeatedly present the two options to the user, but you also want to record the number of times the loop iterates for reporting purposes.

* Use a for loop to take care of both of these requirements:

|  |
| --- |
| int loopCount; bool done = false; for (loopCount = 1; !done ; loopCount++) {     ...     PauseScreen(); } |

1. Create a new private static method named DisplayMenu that returns a nullable MenuOption (the enumeration you created in a previous step).

* If the value returned is null, that means the user selected an option that is not part of the enumeration.
* Examine the following code to see how this could be done:

|  |
| --- |
| /// <summary> /// Display a list of menu options to the user /// </summary> /// <returns> /// The selected option, or null if a valid option was  /// not selected /// </returns> private static MenuOption? DisplayMenu() {   MenuOption result;    // Display the menu   Console.Clear();   Console.WriteLine("{0}. ({1}) Add an animal to the zoo",      (uint)MenuOption.Add, MenuOption.Add);   Console.WriteLine("{0}. ({1}) Quit SimZoo",      (uint)MenuOption.Quit, MenuOption.Quit);   Console.Write("\n\nWhat do you want to do?:");    // Get the result   string input = Console.ReadLine();    // Put into title case (matches MenuOption members)   input = CultureInfo.CurrentCulture.TextInfo.ToTitleCase(input);    // Parse the input (accepts number or letter)   if (Enum.TryParse<MenuOption>(input, out result))   {     return result;   }   return null; } |

|  |  |
| --- | --- |
|  | In order to make the above code you will have to add the following code to the top of your .cs file:  using System.Globalization;  The using directive will be discussed in more detail later. |

|  |  |
| --- | --- |
|  | The CultureInfo class is used to access features that change depending on the current locale and language. For example, title case doesn't always mean the same thing in every language. |

|  |  |
| --- | --- |
|  | This code is not I18N compliant  The problem is that the enumerated type names are being used directly as user input. This is fine for now, but if this code needed to be translated to other languages then it would need to be adjusted. The problem is that it is impossible to tokenize enum members.  The solution would be to create a method that maps the type names to strings and then to tokenize those strings. |

1. Call DisplayMenu from the menu loop and store it in a nullable MenuOption variable.
2. Use a switch statement to respond to the user's choice.

* If the user chose MenuOption.Quit, set done to true and report the number of iterations of the menu loop.
* If the user chose Add, Call GetAnimal.
* If the user chose anything else, inform the user that the selection was not valid.

1. Try running your solution. What number do the menu options start from?
2. Modify the MenuOptions enum so that the options start from 1.

#### If You Have Time

* Modify the menu so that the option list is generated from the enum definition rather than being coded by hand
* Ensure that invalid numeric options are filtered out using Enum.IsDefined

#### Exercise Solution

using System;  
using System.Globalization;  
  
namespace Epic.Training.SimZoo.Text  
{  
  /// <summary>  
  /// Enumeration of menu options  
  /// </summary>  
  enum MenuOption  
  {  
    /// <summary>  
    /// Add an animal to the zoo  
    /// </summary>  
    Add = 1,  
    /// <summary>  
    /// Exit SimZoo  
    /// </summary>  
    Quit  
  }  
  
  /// <summary>  
  /// The class used to drive the user interface  
  /// </summary>  
  class Program  
  {  
    #region class constants  
    /// <summary>  
    /// The first name of the author  
    /// </summary>  
    const string AuthorFirstName = "FirstName";  
    /// <summary>  
    /// The last name of the author  
    /// </summary>  
    const string AuthorLastName = "LastName";  
    #endregion  
  
    /// <summary>  
    /// The entry point into the application  
    /// </summary>  
    /// <param name="args">Set of command-line arguments</param>  
    static void Main(string[] args)  
    {  
      ShowSplashScreen();  
      string animalName;  
      uint animalCount;  
      int loopCount;  
      bool done = false;  
      for (loopCount = 1; !done; loopCount++)  
      {  
        MenuOption? selection = DisplayMenu();  
        switch (selection)  
        {  
          case MenuOption.Add:  
            GetAnimal(out animalName, out animalCount);  
            Console.WriteLine("You added {0} {1} to your zoo!",   
              animalCount, animalName);  
            break;  
          case MenuOption.Quit:  
            Console.WriteLine("Quitting...");  
            Console.WriteLine(  
              "(Statistics: The loop iterated {0} times)",   
              loopCount);  
            done = true;  
            break;  
          default:  
            Console.WriteLine(  
              "That was not a valid selection.  Please try again");  
            break;  
        }  
        PauseScreen();  
      }  
    }  
  
    /// <summary>  
    /// Display a list of menu options to the user  
    /// </summary>  
    /// <returns>  
    /// The selected option, or null if a valid option was not selected  
    /// </returns>  
    private static MenuOption? DisplayMenu()  
    {  
      MenuOption result;  
  
      // Display the menu  
      Console.Clear();  
      Console.WriteLine("{0}. ({1}) Add an animal to the zoo",   
        (uint)MenuOption.Add, MenuOption.Add);  
      Console.WriteLine("{0}. ({1}) Quit SimZoo",   
        (uint)MenuOption.Quit, MenuOption.Quit);  
      Console.Write("\n\nWhat do you want to do?:");  
  
      // Get the result  
      string input = Console.ReadLine();  
  
      // Put into title case (matches MenuOption members)  
      input = CultureInfo.CurrentCulture.TextInfo.ToTitleCase(input);  
  
      // Parse the input (accepts number or letter)  
      if (Enum.TryParse<MenuOption>(input, out result))  
      {  
        return result;  
      }  
      return null;  
    }  
  
    /// <summary>  
    /// Get the name of a type of animal and the number   
    /// of that type of animal to add to the zoo.  
    /// </summary>  
    /// <param name="animalName">The entered animal name</param>  
    /// <param name="animalCount">The number of that animal</param>  
    private static void GetAnimal(  
      out string animalName, out uint animalCount)  
    {  
      do // Get the animal name  
      {  
        Console.Write(  
          "Enter the name of an animal to include in the zoo: ");  
        animalName = Console.ReadLine();  
        if (!string.IsNullOrWhiteSpace(animalName))  
        {  
          break;  
        }  
        Console.WriteLine("You must enter a name for the animal.");  
      } while (true);  
  
      // Get the number of animals  
      Console.Write("Enter the number of this kind of animal: ");  
      while (!uint.TryParse(Console.ReadLine(), out animalCount))  
      {  
        Console.WriteLine("You must enter a positive integer");  
        Console.WriteLine("Enter the number of this kind of animal: ");  
      }  
    }  
  
    /// <summary>  
    /// Display the opening splash screen on the console.  
    /// </summary>  
    private static void ShowSplashScreen()  
    {  
      Console.WriteLine("SimZoo Text v. 1.0");  
      Console.WriteLine("Author: {1}, {0}",   
        AuthorFirstName, AuthorLastName);  
      PauseScreen();  
      Console.Clear();  
    }  
  
    /// <summary>  
    /// Pause the screen until a key is pressed  
    /// </summary>  
    private static void PauseScreen()  
    {  
      Console.WriteLine("<Press any key to continue>");  
      Console.ReadKey();  
    }  
  }  
}

© 2015 Epic Systems Corporation. All rights reserved. Certain information contained herein is confidential and must be treated accordingly. After Visit Summary, Analyst, ASAP, Beaker, BedTime, Break-the-Glass, Breeze, Cadence, Canto, Care Elsewhere, Care Everywhere, Charge Router, Chronicles, Clarity, Cogito ergo sum, Cohort, Colleague, Comfort, Community Connect, Country Connect, Cupid, Epic, EpicCare, EpicCare Link, Epicenter, Epic Earth, EpicLink, EpicOnHand, EpicWeb, Haiku, Healthy Planet, Hyperspace, Identity, IntraConnect, Kaleidoscope, Light Mode, Lucy, MyChart, MyEpic, OpTime, OutReach, Patients Like Mine, Phoenix, Powered by Epic, Prelude, Radar, RedAlert, Region Connect, Resolute, Revenue Guardian, Rover, SmartForms, Sonnet, Stork, Tapestry, Trove, Trusted Partners, Welcome, Willow, Wisdom, With the Patient at Heart and World Connect are registered trademarks, trademarks or service marks of Epic Systems Corporation in the United States and/or in other countries. Other product or company names referenced herein may be trademarks of their respective owners. U.S. and international patents issued and pending.