

C# Language

Today you will learn

- The .NET Languages
- Variables and Data Types
- Variable Operations
- Object-Based Manipulation

The .NET Languages

- The .NET Framework ships with two core languages that are commonly used for building ASP.NET applications:

VB and C#.

- These languages are, to a large degree, functionally equivalent.

Variables and Data Types

- **Variables** can store numbers, text, dates, times, and they can even point to full-fledged objects.
- When you declare a variable, you give it a name, and you specify the type of data it will store.

//Declare a string variable named myName.

string myName;

Note : Comments are descriptive text that is ignored by the compiler.

C# comments provides two basic types of comments:

// A single-line C# comment

/* A multiple-line
C# comment*/

Variables and Data Types

- Every .NET language uses the same variable data types.
- Different languages may provide slightly different names (for example, a VB Integer is the same as a C# int), but the **CLR makes no distinction**-in fact, they are just two different names for the same base data type (in this case, it's System.Int32).
- Because languages share the same core data types, you can easily use objects written in one .NET language in an application written in another .NET language.
- No data type conversions are required.

Variables and Data Types

| C# Name | VB Name | .NET Type Name | Contains |
|---------|---------|----------------|--|
| byte | Byte | Byte | An integer from 0 to 255. |
| short | Short | Int16 | An integer from -32,768 to 32,767. |
| int | Integer | Int32 | An integer from -2,147,483,648 to 2,147,483,647. |
| long | Long | Int64 | An integer from about -9.2e18 to 9.2e18. |
| float | Single | Single | A single-precision floating-point number from approximately -3.4e38 to 3.4e38 (for big numbers) or -1.5e-45 to 1.5e-45 (for small fractional numbers). |
| double | Double | Double | A double-precision floating-point number from approximately -1.8e308 to 1.8e308 (for big numbers) or -5.0e-324 to 5.0e-324 (for small fractional numbers). |
| decimal | Decimal | Decimal | A 128-bit fixed-point fractional number that supports up to 28 significant digits. |
| char | Char | Char | A single Unicode character. |
| string | String | String | A variable-length series of Unicode characters. |
| bool | Boolean | Boolean | A true or false value. |
| * | Date | DateTime | Represents any date and time from 12:00:00 AM, January 1 of the year 1 in the Gregorian calendar, to 11:59:59 PM, December 31 of the year 9999. Time values can resolve values to 100 nanosecond increments. Internally, this data type is stored as a 64-bit integer. |
| * | * | TimeSpan | Represents a period of time, as in ten seconds or three days. The smallest possible interval is 1 <i>tick</i> (100 nanoseconds). |
| object | Object | Object | The ultimate base class of all .NET types. Can contain any data type or object. (You'll take a much closer look at objects in Chapter 3.) |

** If the language does not provide an alias for a given type, you must use the .NET type name.*

Assignment and Initializers

- Once you've declared your variable, you can freely assign values to them, as long as these values have the correct data type.
- Here's the code that shows this two-step process:

```
string myName;  
myName = "Matthew ";
```

- You can also assign a value to a variable in the same line that you declare it.

```
string myName = "Matthew ";
```

Assignment and Initializers

- **C# does not allow uninitialized variables.**
- This means the following code will not compile in C#:

```
int number;                //Number is uninitialized  
number = number + 1; //This causes a compile error
```

- Proper way to write above code:

```
int number=0;              //Number now contains 0  
number = number + 1; //Number now contains 1
```

Assignment and Initializers

- If you're declaring and initializing a variable in a single statement, and if the C# compiler can infer the correct data type based on the value you're using, you don't need to specify the data type.

Here's an example:

```
var myString = "This is also a string"
```

- `myString` is created as string, even though the statement doesn't indicate the string data type.

Arrays

- **Arrays** allow you to store a series of values that have the same data type.
- Each individual value in the array is accessed using one or more index numbers.
- Typically, arrays are laid out contiguously in memory.
- All arrays start at a fixed lower bound of 0.
- Because counting starts at 0, the highest index is actually one less than the number of elements.

Arrays

- When you create an array in C#, you simply specify the upper bound:

```
// Create an array with four strings (from index 0 to index 3).  
string[] stringArray = new string[4];
```

```
// Create a 2 x 4 grid array (with a total of eight integers).  
int[,] intArray = new int[2, 4];
```

Arrays

- You can also fill an array with data at the same time that you create it.
- In this case, you don't need to explicitly specify the number of elements, because .NET can determine it automatically:

```
// Create an array with four strings, one for each number from 1 to 4.  
string[] stringArray = {"1", "2", "3", "4"}
```

Arrays

- The same technique works for **multidimensional arrays**, except that two sets of curly brackets are required

// Create a 4 x 2 array (a grid with four rows and two columns).

```
int[,] intArray = {{1, 2}, {3, 4}, {5, 6}, {7, 8}}
```

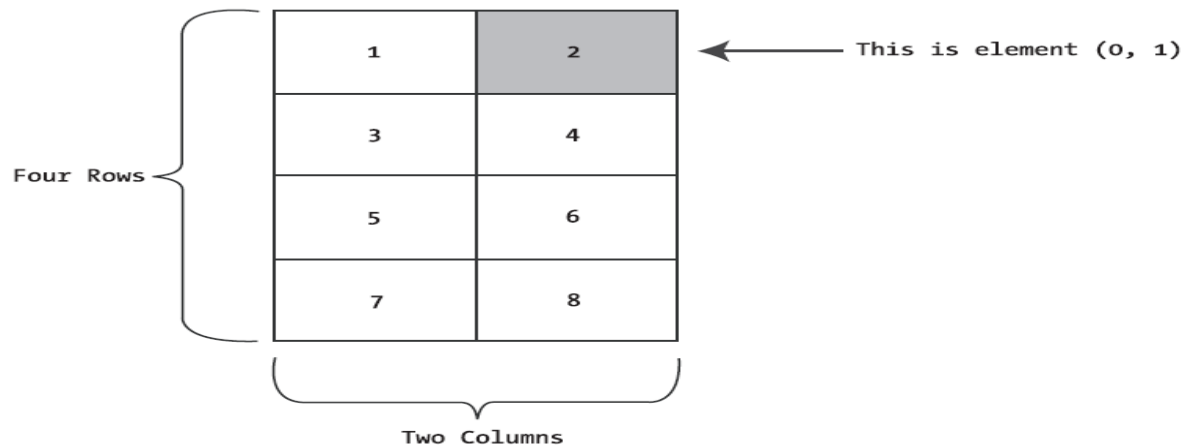


Figure 2-1. A sample two-dimensional array of integers

Arrays

- To access an element in an array, you specify the corresponding index number in square brackets.
- **Array indices are always zero-based.** That means `myArray[0]` accesses the first value in a one dimensional array, `myArray[1]` accesses the second value, and so on.
- In a two-dimensional array, you need two index numbers:

```
int[,] intArray = { {1, 2}, {3, 4}, {5, 6}, {7, 8} }
```

```
// Access the value in row 0 (first row), column 1 (second column).
```

```
int element;
```

```
element = intArray[0, 1] // Element is now set to 2.
```

The ArrayList

- In many cases, it's easier to use a full-fledged collection rather than an array.
- **Collections** are generally better suited to modern object-oriented programming and are used extensively in ASP.NET.
- The .NET class library provides many types of collection classes, including sorted lists, key-indexed lists (dictionaries), and queues.
- One of the simplest collection classes that .NET provides is the **ArrayList**, which always allows **dynamic resizing**.
- Here's a snippet of C# code that uses an ArrayList:

```
// Create an ArrayList object.  
ArrayList dynamicList = new ArrayList();
```

The ArrayList

- The ArrayList is not strongly typed, so you can add any data type.

```
dynamicList.Add("one");  
dynamicList.Add("two");  
dynamicList.Add("three");
```

/*Retrieve the first string. Notice that the object must be converted to a string*/

```
string item = Convert.ToString(dynamicList[0])
```

Difference between Array and ArrayList

| Arrays | ArrayLists |
|--|---|
| These are strong type collection and allow to store fixed length | Array Lists are not strong type collection and size will increase or decrease dynamically |
| In arrays we can store only one datatype either int, string, char etc... | In arraylist we can store all the datatype values |
| Arrays belong to System.Array namespace | Arraylist belongs to System.Collection namespaces |

Enumerations

- An **enumeration** is a group of related **constants**, each of which is given a **descriptive name**.
- Each value in an enumeration corresponds to a preset integer.
- In your code, however, you can refer to an enumerated value by name, which makes your code clearer and helps prevent errors.

Enumerations

- Here's an example of an enumeration that defines different types of users:

// Define an enumeration called UserType with three possible values.

```
enum UserType
{
    Admin,
    Guest,
    Invalid
}
```

- Now you can use the **UserType** enumeration as a special data type that is restricted to one of three possible values.

Enumerations

- You assign or compare the **enumerated value** using the **dot notation** shown in the following example:

```
// Create a new value and set it equal to the UserType.Admin  
constant.
```

```
UserType newUserType;  
newUserType = UserType.Admin
```

- Internally, **enumerations are maintained as numbers**. In the preceding example, 0 is automatically assigned to Admin, 1 to Guest, and 2 to Invalid.

Variable Operations

- You can use all the standard types of variable operations in C#. When working with numbers, you can use various math symbols.

| Operator | Description | Example |
|----------|--|-----------------|
| + | Addition | $1 + 1 = 2$ |
| - | Subtraction | $5 - 2 = 3$ |
| * | Multiplication | $2 * 5 = 10$ |
| / | Division | $5.0 / 2 = 2.5$ |
| % | Gets the remainder left after integer division | $7 \% 3 = 1$ |

Variable Operations

- C# follows the conventional order of operations, performing exponentiation first, followed by multiplication and division and then addition and subtraction.
- You can also control order by grouping subexpressions with parentheses.

```
int number;
```

```
number = 4 + 2 * 3
```

```
//number will be 10.
```

```
number = (4 + 2) * 3
```

```
//number will be 18.
```

Variable Operations

- When dealing with strings, you can use the addition operator (+), to join two strings.

//Join two strings together.

```
fullName = firstName + " " + lastName
```

- In addition, C# also provides special shorthand assignment operators. Here are a few examples:

//Add 10 to myValue (the same as myValue = myValue + 10).

```
myValue += 10
```

Type Conversions

- Converting information from one data type to another is a fairly common programming task.
- For example, you might retrieve text input for a user that contains the number you want to use for a calculation.
- Conversions are of two types:
widening and **narrowing**.
- *Widening* conversions always succeed. For example, you can always convert a number into a string, or a 16-bit integer into a 32-bit integer.

Type Conversions

- On the other hand, *narrowing* conversions may or may not succeed, depending on the data.

If you're converting a 32-bit integer to a 16-bit integer, you could encounter a runtime error if the 32-bit number is larger than the maximum value that can be stored in the 16-bit data type.

- Some strings can't be converted to numbers.
- A failed **narrowing** conversion will lead to an unexpected runtime error.

Type Conversions

- To perform an *explicit* data type conversion in C#, you need to specify the type in parentheses before the expression you're converting.
- Here's how you could rewrite the earlier example with explicit conversions:

```
int count32 = 100;
```

```
short count16;
```

```
/*Explicitly convert your 32-bit number into a 16-bit number. If count32 is  
too large to fit, .NET will discard some of the info.*/
```

```
count16 = (short)count32;
```

Type Conversions

- Casting cannot be used to convert numbers to string, or vice versa. Instead use the following code:

```
string countString = "10";  
// Convert the string "10" to the numeric value 10  
int count = Convert.ToInt32(countString);  
  
// Convert numeric value 10 into the string "10".  
countString = Convert.ToString(count);
```

- Converting string to number will not work if string contains letters or other non-numeric characters.

Object-Based Manipulation

- .NET is object-oriented to the core. In fact, even ordinary numeric variables like the ones you've seen earlier are really full-fledged objects in disguise.
- This means that common data types have the built-in smarts to handle basic operations.
- For example, all strings are actually complete string objects, with useful methods and properties (such as a Length property that counts the number of characters in the string.)

Object-Based Manipulation

- As an example, every type in the .NET class library includes a **ToString()** method.
- The default implementation of this method **returns the class name**. In simple variables, a more useful result is returned: the string representation of the given variable.
- The following code snippet demonstrates how to use the ToString() method with an integer:

```
string myString;  
int myInteger = 100;  
// Convert a number to a string. myString will have the contents "100".  
myString = myInteger.ToString();
```

Object-Based Manipulation

- To understand the previous example, you need to remember that all integer variables are based on the `Int32` type in the .NET class library.
- The `ToString()` method is built into the `Int32` type, so it's available when you use an integer in any language.

The String Type

- The following code snippet shows several ways to manipulate a string using the methods in the String type:

```
String myString = "This is a test string"
myString = myString.Trim()           // = "This is a test string"
myString = myString.Substring(0, 4)   // = "This"
myString = myString.ToUpper()         // = "THIS"
myString = myString.Replace("IS", "AT") // = "THAT"
int length = myString.Length          // = 4
```

The String Type

- The first few statements use built-in methods of the **String** type, such as **Trim()**, **Substring()**, **ToUpper()** and **Replace()**.
- Each of these methods generates a new string object, which replaces the current contents of the **MyString** variable.
- The final statement uses the built-in **Length** property of the **String** type, which returns an integer that represents the number of characters in the string.

The String Type

| Member | Description |
|---------------------------------------|---|
| Length | Returns the number of characters in the string (as an integer). |
| ToUpper() and ToLower() | Returns a copy of the string with all the characters changed to uppercase or lowercase characters. |
| Trim(), TrimEnd(), and TrimStart() | Removes spaces (or the characters you specify) from either end (or both ends) of a string. |
| PadLeft() and PadRight() | Adds the specified character to the appropriate side of a string as many times as necessary to make the total length of the string equal to the number you specify. For example, "Hi".PadLeft(5, '@') returns the string @@@@Hi. |
| Insert() | Puts another string inside a string at a specified (zero-based) index position. For example, Insert(1, "pre") adds the string <i>pre</i> after the first character of the current string. |
| Remove() | Removes a specified number of characters from a specified position. For example, Remove(0, 1) removes the first character. |
| Replace() | Replaces a specified substring with another string. For example, Replace("a", "b") changes all <i>a</i> characters in a string into <i>b</i> characters. |
| Substring() | Extracts a portion of a string of the specified length at the specified location (as a new string). For example, Substring(0, 2) retrieves the first two characters. |
| StartsWith() and EndsWith() | Determines whether a string starts or ends with a specified substring. For example, StartsWith("pre") will return either true or false, depending on whether the string begins with the letters <i>pre</i> in lowercase. |
| IndexOf() and LastIndexOf() | Finds the zero-based position of a substring in a string. This returns only the first match and can start at the end or beginning. You can also use overloaded versions of these methods that accept a parameter that specifies the position to start the search. |
| Split() | Divides a string into an array of substrings delimited by a specific substring. For example, with Split(".") you could chop a paragraph into an array of sentence strings. |
| Join() | Fuses an array of strings into a new string. You must also specify the separator that will be inserted between each element (or use an empty string if you don't want any separator). |

The DateTime and TimeSpan Types

- The **DateTime** and **TimeSpan** data types also have built-in methods and properties.
- These class members allow you to perform three useful tasks:
 - Extract a part of a DateTime (for example, just the year) or convert a TimeSpan to a specific representation (such as the total number of days or total number of minutes).
 - Easily perform date and time calculations.
 - Determine the current date and time and other information (such as the day of the week or whether the date occurs in a leap year).

The DateTime and TimeSpan Types

- For example, the following block of code creates a DateTime object, sets it to the current date and time, and adds a number of days. It then creates a string that indicates the year that the new date falls in (for example, 2016).

```
DateTime myDate = DateTime.Now;  
myDate = myDate.AddDays(100);  
string dateString = myDate.Year.ToString();
```

The DateTime and TimeSpan Types

- The DateTime and TimeSpan classes also support the $+$ and $-$ arithmetic operators

```
DateTime myDate1 = DateTime.Now;
```

```
TimeSpan interval = TimeSpan.FromHours(3000);
```

```
DateTime myDate2 = myDate1 + interval;
```

```
// Subtracting one DateTime object from another produces a TimeSpan
```

```
TimeSpan difference;
```

```
difference = myDate2 - mydate1
```

The DateTime Members

| Member | Description |
|--|--|
| Now | Gets the current date and time. You can also use the <code>UtcNow</code> property to take the current computer's time zone into account. <code>UtcNow</code> gets the time as a <i>coordinated universal time</i> (UTC). Assuming your computer is correctly configured, this corresponds to the current time in the Western European (UTC+0) time zone. |
| Today | Gets the current date and leaves time set to 00:00:00. |
| Year, Date, Month, Hour, Minute, Second, and Millisecond | Returns one part of the <code>DateTime</code> object as an integer. For example, <code>Month</code> will return 12 for any day in December. |
| DayOfWeek | Returns an enumerated value that indicates the day of the week for this <code>DateTime</code> , using the <code>DayOfWeek</code> enumeration. For example, if the date falls on Sunday, this will return <code>DayOfWeek.Sunday</code> . |
| Add() | Adds a <code>TimeSpan</code> to a <code>DateTime</code> and returns the result as a new <code>DateTime</code> . For convenience, these operations are mapped to the <code>+</code> and <code>-</code> operators, so you can use them instead when performing calculations with dates. |

The DateTime Members

| | |
|--|---|
| <code>Subtract()</code> | Subtracts a <code>TimeSpan</code> or <code>DateTime</code> from another <code>DateTime</code> . Returns a <code>TimeSpan</code> that represents the difference. |
| <code>AddYears()</code> , <code>AddMonths()</code> , <code>AddDays()</code> , <code>AddHours()</code> , <code>AddMinutes()</code> , <code>AddSeconds()</code> , <code>AddMilliseconds()</code> | Accepts an integer that represents a number of years, months, and so on, and returns a new <code>DateTime</code> . You can use a negative integer to perform a date subtraction. |
| <code>DaysInMonth()</code> | Returns the number of days in the specified month in the specified year. |
| <code>IsLeapYear()</code> | Returns true or false depending on whether the specified year is a leap year. |
| <code>ToString()</code> | Returns a string representation of the current <code>DateTime</code> object. You can also use an overloaded version of this method that allows you to specify a parameter with a format string. |

The TimeSpan Members

| Member | Description |
|---|--|
| Days, Hours, Minutes, Seconds, Milliseconds | Returns one component of the current TimeSpan. For example, the Hours property can return an integer from -23 to 23. |
| TotalDays, TotalHours, TotalMinutes, TotalSeconds, TotalMilliseconds | Returns the total value of the current TimeSpan as a number of days, hours, minutes, and so on. The value is returned as a double, which may include a fractional value. For example, the TotalDays property might return a number like 234.342. |
| Add() and Subtract() | Combines TimeSpan objects together. For convenience, these operations are mapped to the + and - operators, so you can use them instead when performing calculations with times. |
| FromDays(), FromHours(), FromMinutes(), FromSeconds(), FromMilliseconds() | Allows you to quickly create a new TimeSpan. For example, you can use TimeSpan.FromHours(24) to create a TimeSpan object exactly 24 hours long. |
| ToString() | Returns a string representation of the current TimeSpan object. You can also use an overloaded version of this method that allows you to specify a parameter with a format string. |

The Array Type

- All arrays in .NET are actually instances of the Array type
- Length property or the GetLength() method

```
int[] myArray= {1, 2, 3, 4, 5};
```

```
int numberOfElements;
```

```
numberOfElements = myArray.Length //NumberOfElements = 5
```

```
int bound;
```

```
//Zero represents the first dimension of an array.
```

```
bound = myArray.GetUpperBound(0) //Bound = 4
```

The Array Type

- Specific dimension as parameter to `GetUpperBound()`

// Create a 4 X 2 array (a grid with four rows and two columns).

```
int[,] intArray = { {1, 2}, {3, 4}, {5, 6}, {7, 8} }
```

```
int rows = intArray.GetUpperBound(0) + 1    // Rows = 4
```

```
int columns = intArray.GetUpperBound(1) + 1  // Columns = 2
```


The Array Type

| Member | Description |
|-------------------------------------|---|
| Length | Returns an integer that represents the total number of elements in all dimensions of an array. For example, a 3×3 array has a length of 9. |
| GetLowerBound() and GetUpperBound() | Determines the index position of the last element in an array. As with just about everything in .NET, you start counting at zero (which represents the first dimension). |
| Clear() | Resets part or all of an array's contents. Depending on the index values that you supply. The elements revert to their initial empty values (such as 0 for numbers, and an empty string for strings). |
| IndexOf() and LastIndexOf() | Searches a one-dimensional array for a specified value and returns the index number. You cannot use this with multidimensional arrays. |
| Sort() | Sorts a one-dimensional array made up of comparable data such as strings or numbers. |
| Reverse() | Reverses a one-dimensional array so that its elements are backward, from last to first. |

End of lecture