Chapter Two

Core C# Programming

Outline

- The Anatomy of a Simple C# Program
- System Data Types and Corresponding C# Keywords
- Working with String Data
- Narrowing and Widening Data Type Conversions
- Understanding Implicitly Typed Local Variables
- C# Iteration Constructs
- Decision Constructs and the Relational/Equality Operators
- Methods and Parameter Modifiers
- Understanding C# Arrays
- Understanding the enum and structure Types
- Understanding Value Types and Reference Types
- Understanding C# Nullable Types

The Anatomy of a Simple C# Program

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace SimpleCSharpApp
{
class Program
{
static void Main(string[] args)
{
//body
}
}
}
```

Variations on the Main() Method

By default, Visual Studio will generate a Main() method that has a void return value and an array of string types as the single input parameter. This is not the only possible form of Main(), however.

It is permissible to construct your application's entry point using any of the following signatures (assuming it is contained within a C# class or structure definition):

```
// int return type, array of strings as the parameter.
static int Main(string[] args)
{
// Must return a value before exiting!
return 0;
}
// No return type, no parameters.
static void Main()
{
}
// int return type, no parameters.
static int Main()
{
// Must return a value before exiting!
return 0;
}
```

Which one to choose?

- Obviously, your choice of how to construct Main() will be based on two questions.

➢ First, do you want to return a value to the system when Main() has completed and your program terminates? If so, you need to return an int data type rather than void.

➢ Second, do you need to process any user-supplied, command-line parameters? If so, they will be stored in the array of strings.

System Data Types and Corresponding C# Keywords

- Like any programming language, C# defines keywords for fundamental data types, which are used to represent local variables, class data member variables, method return values, and parameters.
- C# data type keywords are actually shorthand notations for full-blown types in the System namespace.

Variable Declaration and Initialization

```
static void LocalVarDeclarations()
{
Console.WriteLine("=> Data Declarations:");
// Local variables are declared as so:
// dataType varName;
int myInt;
string myString;
Console.WriteLine();
}
```

NOTE:-Be aware that it is a *compiler error to make use of a local variable before assigning an initial value.*

Intrinsic Data Types and the new Operator

- All intrinsic data types support what is known as a *default constructor .This feature* allows you to create a variable using the new keyword, which automatically sets the variable to its default value.

- bool variables are set to false.
- Numeric data is set to 0 (or 0.0 in the case of floating-point data types).
- char variables are set to a single empty character.
- BigInteger variables are set to 0.
- DateTime variables are set to 1/1/0001 12:00:00 AM.
- Object references (including strings) are set to null.

Continued....

```
static void NewingDataTypes()
{
Console.WriteLine("=> Using new to create variables:");
bool b = new bool();          // Set to false.
int i = new int();                // Set to 0.
double d = new double();  // Set to 0.
DateTime dt = new DateTime(); // Set to 1/1/0001
   12:00:00 AM
Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
Console.WriteLine();
}
```

Working with String Data

- System.String provides a number of methods you would expect from such a utility class, including methods that return the length of the character data, find substrings within the current string, and

convert to and from uppercase/lowercase.

Continued....

- Static member functions

Compare(), Format(), Concat()

- Instance member functions

Contains(), Equals(), Insert(), PadLeft()&PadRight(), Remove(), Replace(), Split(), Trim(), ToUpper()&ToLower()

- Property member

Length

Escape Characters

- As in other C-based languages, C# string literals may contain various *escape characters*
- They Qualify how the character data should be printed to the output stream.
- Each escape character begins with a backslash, followed by a specific token.

Continued….

\‘	Inserts a single quote into a string literal.
\“	Inserts a double quote into a string literal.
\\\\	Inserts a backslash into a string literal. This can be quite helpful when defining file or network paths.
\a	Triggers a system alert (beep). For console programs, this can be an audio clue to the user.
\n	Inserts a new line (on Windows platforms).
\r	Inserts a carriage return.
\t	Inserts a horizontal tab into the string literal.

Narrowing and Widening Data Type Conversions

- WIDENING is the term used to define an implicit upward cast that does not result in a loss of data.
- the compiler implicitly *widens each short to an int.*
- NARROWING is the logical opposite of widening, in that a larger value is stored within a smaller data type variable.
- The CLR is unable to apply a *narrowing operation.*

Continued…

//widening example

```
class Program
{
static void Main(string[] args)
{
Console.WriteLine("***** Fun with type conversions *****");
// Add two shorts and print the result.
short numb1 = 9, numb2 = 10;
Console.WriteLine("{0} + {1} = {2}",
numb1, numb2,Add(numb1, numb2));
Console.ReadLine();
}
static int Add(int x, int y)
{
return x + y;
}}
```

Continued...

//Narrowing example

```
class Program
{
static void Main(string[] args)
{
Console.WriteLine("***** Fun with type conversions *****");
// Compiler error below!
short numb1 = 30000, numb2 = 30000;
short answer = Add(numb1, numb2);
Console.WriteLine("{0} + {1} = {2}",
numb1, numb2, answer);
Console.ReadLine();
})
```

Note:-In this case, the compiler reports the following error:

Cannot implicitly convert type 'int' to 'short'. An explicit conversion exists (are you missing a cast?)

Casting

- When you need narrowing operation, you must apply an *explicit cast*
- using the C# casting operator ().

Syntax by example

```
double x = 1234.7;
int a;
a = (int)x; // Cast double to int.
```

Understanding Implicitly Typed Local Variables

- It is always good practice to explicitly specify the data type of each variable
- The C# language does provide for implicitly typing of local variables using the var keyword.
- The var keyword can be used in place of specifying a specific data type (such as int,bool, or string).
- the compiler will automatically infer the underlying data type based on the initial value used to initialize the local data point.

Example

```
static void DeclareImplicitVars()
{
// Implicitly typed local variables
// are declared as follows:
// var variableName = initialValue;
var myInt = 0;
var myBool = true;
var myString = "Time, marches on...";
}
```

Restrictions on Implicitly Typed Variables

1. implicit typing applies *only to local variables in a method or property scope.*

- It is illegal to use the var keyword to define return values, parameters, or field data of a custom type.

2. local variables declared with the var keyword must be assigned an initial value at the exact time of declaration and cannot be assigned the initial value of null.

Continued..

- It is permissible, however, to assign an inferred local variable to null after its initial assignment
- Furthermore, it is permissible to assign the value of an implicitly typed local variable to the value of other variables, implicitly typed or not.
- Also, it is permissible to return an implicitly typed local variable to the caller, provided the method return type is the same underlying type as the var-defined data point.

C# Iteration Constructs

- All programming languages provide ways to repeat blocks of code until a terminating condition has beenmet.
- C# provides the following four iteration constructs
    - for loop
    - while loop
    - do/while loop
    - foreach/in loop

For loop

```
for ( variable initialization; condition;   variable update )
 {

// Code to execute while the condition is true


}
```

While loop

```
While(condition)
{
// Code to execute while the condition is
   true

//Variable Update
}
```

Note:- variable initialization is outside the loop

Do while looop

```
do{
// Code to execute while the condition is true
//Variable Update
}while(condition);
```

Note:- variable initialization is outside the loop

Decision Constructs and the Relational/Equality Operators

- C# defines two simple constructs to alter the flow of your program, based on various contingencies:
  - The if/else statement
  - The switch statement

The If/else statements

- C# if/else statements typically involve the use of the C# operators shown in Table in order to obtain a literal Boolean value.
- Unlike in C and C++, the if/else statement in C# operates only on Boolean expressions, not ad hoc values such as –1 or 0.
- Types
    - If only
    - If…else
    - If…else if
    - Nested if

Continued..

```
if(Boolean expression) {
//Statements will execute if the
Boolean expression is true
}
```

```
if(Boolean expression){
 //Executes when the Boolean
expression is true
}
else{
//Executes when the Boolean
expression is false
}
```

```
if(Boolean_expression 1){
//Executes when the Boolean
expression 1 is true }
else if(Boolean_expression 2){
//Executes when the Boolean
expression 2 is true }
else if(Boolean_expression3)
{//Executes when the Boolean
expression 3 is true }
else { //Executes when the none of the
above condition is true. }
```

```
if(Boolean_expression 1)
{
 //Executes when the Boolean
expression 1 is true
if(Boolean_expression 2)
{
 //Executes when the Boolean
expression 2 is true
}
}//end of the outer if
```

C# Relational and Equality Operators

C# Equality/Relational Operator	Example Usage	Meaning in Life
==	if(age == 30)	Returns true only if each expression is the same.
!=	if("Foo" != myStr)	Returns true only if each expression is different.
<	if(bonus < 2000)	Returns true if expression A (bonus) is less than, greater than, less than or equal to, or greater than or equal to expression B (2000).
>	if(bonus > 2000)	
<=	if(bonus <= 2000)	
>=	if(bonus >= 2000)	

Conditional Operators

Operator	Example	Meaning in Life
&&	if(age == 30 && name == "Fred")	AND operator. Returns true if all expressions are true.
||	if(age == 30 || name == "Fred").	OR operator. Returns true if at least one expression is true
!	if(!myBool)	NOT operator. Returns true if false, or false if true.

Note The && and || operators both "short circuit" when necessary. This means that after a complex expression has been determined to be false, the remaining subexpressions will not be checked. If you require all expressions to be tested regardless, you can use the related & and | operators.

The switch Statement

- the switch statement allows you to handle program flow based on a predefined set of choices.
- One nice feature of the C# switch statement is that you can evaluate string data in addition to numeric data.

syntax

```
switch(expression)
{
 case value : //Statements
 break; //optional
 case value :
//Statements
break; //optional
//You can have any number of case statements.
default : //Optional
//Statements
 }
```

Note C# demands that each case (including default) that contains executable statements have a terminating break or goto to avoid fall-through.

Methods and Parameter Modifiers

- Method is an object’s behaviour.
- methods can be implemented within the scope of classes or structures (as well as prototyped within interface types)
- They may be decorated with various keywords (e.g., static, virtual, public, new) to qualify their behaviour.

➢ Our methods has followed the following basic format:

```
// Recall that static methods can be called directly
// without creating a class instance.
class Program
{
// static returnType MethodName(paramater list) { /* Implementation */ }
static int Add(int x, int y){ return x + y;
}
```

- While the definition of a method in C# is quite straightforward, there are a handful of keywords that you can use to control how arguments are passed to the method in question.

Continued..

Parameter Modifier	Meaning in Life
(None)	If a parameter is not marked with a parameter modifier, it is assumed to be passed by value, meaning the called method receives a copy of the original data.
out	Output parameters must be assigned by the method being called, and therefore, are passed by reference. If the called method fails to assign output parameters, you are issued a compiler error.

Continued..

Parameter Modifier	Meaning in Life
ref	The value is initially assigned by the caller and may be optionally reassigned by the called method (as the data is also passed by reference). No compiler error is generated if the called method fails to assign a ref parameter.
params	This parameter modifier allows you to send in a variable number of arguments as a single logical parameter. A method can have only a single params modifier, and it must be the final parameter of the method. In reality, you might not need to use the params modifier all too often; however, be aware that numerous methods within the base class libraries do make use of this C# language feature.

The Default by Value Parameter-Passing Behaviour

```
static int Add(int x, int y)
{
// Caller will not see these changes
// as you are modifying a copy of the
// original data.
x = 10000;
y = 88888;
int ans = x + y;
return ans;
}
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Methods *****\n");
// Pass two variables in by value.
int x = 9, y = 10;
Console.WriteLine("Before call: X: {0}, Y: {1}", x, y);
Console.WriteLine("Answer is: {0}", Add(x, y));
Console.WriteLine("After call: X: {0}, Y: {1}", x, y);
Console.ReadLine();
}
```

The out Modifier

- Methods that have been defined to take output parameters (via the out keyword) are under obligation to assign them to an appropriate value before exiting the method scope
- if you fail to do so, you will receive compiler errors.

Continued..

```
// Output parameters must be assigned by the called method.
static void Add(int x, int y, out int ans)
{
ans = x + y;
}
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Methods *****");
// No need to assign initial value to local variables
// used as output parameters, provided the first time
// you use them is as output arguments.
int ans;
Add(90, 90, out ans);
Console.WriteLine("90 + 90 = {0}", ans);
Console.ReadLine();
}
```

Continued..

- C# out modifier does serve a very useful purpose: it allows the caller to obtain multiple outputs from a single method invocation.
- Remember that you must use the out modifier when you invoke the method, as well as when you implement the method:

```
static void FillTheseValues(out int a, out string b, out bool c){
a = 9;
b = "Enjoy your string.";
c = true;
}
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Methods *****");
...
int i; string str; bool b;
FillTheseValues(out i, out str, out b);
Console.WriteLine("Int is: {0}", i);
Console.WriteLine("String is: {0}", str);
Console.WriteLine("Boolean is: {0}", b);
Console.ReadLine();
}
```

The ref Modifier

- Reference parameters are necessary when you wish to allow a method to operate on (and usually change the values of) various data points declared in the caller's scope
- sorting or swapping routine

ref vs out

- Output parameters do not need to be initialized before they passed to the method. The reason for this is that the method must assign output parameters before exiting.
- Reference parameters must be initialized before they are passed to the method. The reason for this is that you are passing a reference to an existing variable. If you don't assign it to an initial value, that would be the equivalent of operating on an unassigned local variable.

Continued…

```
// Reference parameters.
public static void SwapStrings(ref string s1, ref string s2)
{
string tempStr = s1;
s1 = s2;
s2 = tempStr;
}
//This method can be called as follows:
static void Main(string[] args){
Console.WriteLine("***** Fun with Methods *****");
string str1 = "Flip";
string str2 = "Flop";
Console.WriteLine("Before: {0}, {1} ", str1, str2);
SwapStrings(ref str1, ref str2);
Console.WriteLine("After: {0}, {1} ", str1, str2);
Console.ReadLine();
}
```

The params Modifier

- C# supports the use of *parameter arrays using the params keyword.*
- The params keyword allows you to pass into a method a variable number of identically typed parameters (or classes related by inheritance) as a single logical parameter.
- arguments marked with the params keyword can be processed if the caller sends in a strongly typed array or a comma delimited list of items.

Continued…

```
// Return average of "some number" of doubles.
static double CalculateAverage(params double[] values)
{
Console.WriteLine("You sent me {0} doubles.", values.Length);
double sum = 0;
if(values.Length == 0)
return sum;
for (int i = 0; i < values.Length; i++)
sum += values[i];
return (sum / values.Length);
}
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Methods *****");
// Pass in a comma-delimited list of doubles...
double average;
average = CalculateAverage(4.0, 3.2, 5.7, 64.22, 87.2);
Console.WriteLine("Average of data is: {0}", average);
// ...or pass an array of doubles.
double[] data = { 4.0, 3.2, 5.7 };
average = CalculateAverage(data);
Console.WriteLine("Average of data is: {0}", average);
// Average of 0 is 0!
Console.WriteLine("Average of data is: {0}", CalculateAverage());
Console.ReadLine();
}
```

Defining Optional Parameters

- This technique allows the caller to invoke a single method while omitting arguments deemed unnecessary.

```
static void EnterLogData(string message, string owner = "Programmer")
{
Console.Beep();
Console.WriteLine("Error: {0}", message);
Console.WriteLine("Owner of Error: {0}", owner);
}
//calling
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Methods *****");
EnterLogData("Oh no! Grid can't find data");
EnterLogData("Oh no! I can't find the payroll data", "CFO");
Console.ReadLine();
}
```

Invoking Methods Using Named Parameters

- Named arguments allow you to invoke a method by specifying parameter values in any order you choose.
- Thus, rather than passing parameters solely by position you can choose to specify each argument by name using a colon operator.

Continued...

```
static void DisplayFancyMessage(ConsoleColor textColor,ConsoleColor backgroundColor, string message){
// Set new colors and print message.
Console.ForegroundColor = textColor;
Console.BackgroundColor = backgroundColor;
Console.WriteLine(message);
//calling
static void Main(string[] args){
Console.WriteLine("***** Fun with Methods *****");
DisplayFancyMessage(message: "Wow! Very Fancy indeed!",
textColor: ConsoleColor.DarkRed,
backgroundColor: ConsoleColor.White);
DisplayFancyMessage(backgroundColor: ConsoleColor.Green,
message: "Testing...“,textColor: ConsoleColor.DarkBlue);
}
```

Understanding Method Overloading

- Like other modern object-oriented languages, C# allows a method to be *overloaded.*
- Simply put, when you define a set of identically named methods that differ by the number (or type) of parameters, the method in question is said to be overloaded.

Continued…

```
// C# code.
class Program
{
static void Main(string[] args)
{
}
// Overloaded Add() method.
static int Add(int x, int y)
{ return x + y; }
static double Add(double x, double y)
{ return x + y; }
static long Add(long x, long y)
{ return x + y; }
}
```

Understanding C# Arrays

- an array is a set of data items, accessed using a numerical index.
- Creating array

```
      int[] myInts = new int[3];
      string[] books = new string[100];
```

- fill an array of 3 Integers

```
   myInts[0] = 100;
   myInts[1] = 200;
   myInts[2] = 300;
```

- // Now print each value.

```
   foreach(int i in myInts)
           Console.WriteLine(i);
```

C# Array Initialization Syntax

- In addition to filling an array element by element, you are also able to fill the items of an array using C# array initialization syntax.
- // Array initialization syntax using the new keyword.

```
string[] stringArray = new string[]{ "one", "two",
    "three" };
```

- // Array initialization syntax without using the new keyword.

```
bool[] boolArray = { false, false, true };
```

- // Array initialization with new keyword and size.

```
int[] intArray = new int[4] { 20, 22, 23, 0 };
```

Continued..

- If there is a mismatch between the declared size and the number of initializers you are issued a compile-time error.
- whether you have too many or too few initializers
- // OOPS! Mismatch of size and elements!

```
int[] intArray = new int[2] { 20, 22, 23, 0 };
```

Implicitly Typed Local Arrays

- the var keyword can be used to define *implicitly typed local arrays*
- Using this technique, you can allocate a new array variable without specifying the type contained within the array itself
- an implicitly typed local array does not default to System.Object
- // Error! Mixed types!

var d = new[] { 1, "one", 2, "two", false };

Continued..

```
static void DeclareImplicitArrays()
{
Console.WriteLine("=> Implicit Array Initialization.");
// a is really int[].
var a = new[] { 1, 10, 100, 1000 };
Console.WriteLine("a is a: {0}", a.ToString());
// b is really double[].
var b = new[] { 1, 1.5, 2, 2.5 };
Console.WriteLine("b is a: {0}", b.ToString());
// c is really string[].
var c = new[] { "hello", null, "world" };
Console.WriteLine("c is a: {0}", c.ToString());
Console.WriteLine();
}
```

Working with Multidimensional Arrays

- C# supports two varieties of multidimensional arrays.

➢ ***rectangular array***

- which is simply an array of multiple dimensions, where each row is of the same length.

➢ ***jagged array***

- This arrays contain some number of inner arrays, each of which may have a different upper limit
- an array of arrays

Example

```
// A rectangular MD array.
int[,] myMatrix;
myMatrix = new int[6,6];
```

- // A jagged MD array (i.e.,).
- // Here we have an array of 5 different arrays.

```
 int[][] myJagArray = new int[5][];
```

Arrays As Arguments or Return Values

- Arrays can be passed as an argument or receive it as a member return value.

```
static void PrintArray(int[] myInts)
{
for(int i = 0; i < myInts.Length; i++)
Console.WriteLine("Item {0} is {1}", i, myInts[i]);
}
static string[] GetStringArray()
{
string[] theStrings = {"Hello", "from", "GetStringArray"};
return theStrings;
}
```

The System.Array Base Class

Member of Array Class	Meaning in Life
Clear()	This static method sets a range of elements in the array to empty values (0 for numbers, null for object references, false for booleans).
CopyTo()	This method is used to copy elements from the source array into the destination array.
Length	This property returns the number of items within the array.
Rank	This property returns the number of dimensions of the current array.

The System.Array…..

Member of Array Class	Meaning in Life
Reverse()	This static method reverses the contents of a one-dimensional array.
Sort()	This static method sorts a one-dimensional array of intrinsic types. If the elements in the array implement the IComparer interface, you can also sort your custom types

Sorting code example

```
class Program
    {
        static void Main(string[] args)
        {
            int[] no = {5,3,4,2,1};
            Console.WriteLine("array befor sorting");
            foreach (int i in no)
                Console.Write(i + "  ");
            Console.WriteLine();
            Console.WriteLine("array after sorting");
            Array.Sort(no);
            foreach (int i in no)
                Console.Write(i + "  ");
            Console.WriteLine();

        }
    }
```