Methods and Enums

Methods or Functions

- Official C# terminology makes a distinction between functions and methods.
- In C# terminology, the term “function” includes not only methods, but also other non-data members of a class or struct.
- This includes methods, properties, constructors, indexers, operators, destructors.

- On the contrast, data members are: fields, constants, and events.

Methods in C#

- A method is a group of statements that together perform a task.
- Every C# program has at least one class with a method named `Main`.

- To use a method, you need to:
  - Define the method
  - Call the method

Defining Methods in C#

- When you define a method, you basically declare the elements of its structure.
- The syntax for defining a method in C# is as follows:

```
<Access Modifier> <Return Type> <Method Name>(Parameter List)
{
      Method Body
}
```

Defining Methods in C#

```
<Access Modifier> <Return Type> <Method Name>(Parameter List)
{
      Method Body
}
```

Following are the various elements of a method:

- Access Modifier: This determines the visibility of a variable or a method from another class.
- Return Type: A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is `void`.
- Method Name: Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- Parameter List: Enclosed between parentheses, the parameters are used to pass and receive data from a method. Parameters are optional.
- Method Body: This contains the set of instructions needed to complete the required activity.

Examples of Methods

- Method without parameters and without return type:

```
public void Add()
{
      int a = 10;
      int b = 20;

      int sum = a + b;

      Console.WriteLine("Sum is " + sum);
}
```

Examples of Methods

- Method with parameters and without return type:

```
public void Add(int x, int y)
{
      int sum = x + y;
      Console.WriteLine("Sum is " + sum);
}
```

Examples of Methods

- Method without parameters and with return type:

```
public int Add()
{
      int x = 10;
      int y = 20;

      int sum = x + y;

      return sum;
}
```

Examples of Methods

- Method with parameters and with return type:

```
public int Add(int x, int y)
{
      int sum = x + y;

      return sum;
}
```

Method Parameters

- When method with parameters is called, you need to pass the parameters to the method.
- There are different ways that parameters can be passed to a method:
- Value Parameters:
  - This method copies the actual value of an argument into the formal parameter of the method.
  - In this case, changes made to the parameter inside the method have no effect on the argument.
- Reference Parameters:
  - This method copies the reference to the memory location of an argument into the formal parameter.
  - This means that changes made to the parameter affect the argument.
- Output Parameters:
  - This method helps in returning more than one value.
- Parameter Array:
  - You can pass **n** number of parameters to a method.

Example of Passing By Value and Reference

```
public static void Main()
{
      int i = 0;
      string s = "hello";
      int[] a = { 0, 1, 2, 4, 8 };

      Console.WriteLine("Before Changes:");
      Console.WriteLine("i = " + i);
      Console.WriteLine("s = " + s);
      Console.WriteLine("a[0] = " + a[0]);

      ChangeValues(i, s, a);
      Console.WriteLine("\nAfter Changes:");
      Console.WriteLine("i = " + i);
      Console.WriteLine("s = " + s);
      Console.WriteLine("a[0] = " + a[0]);
}
static void ChangeValues(int num, string str, int[] array)
{
      num = 100;        str = "bye";      array[0] = 100;
}
```

```
Output:

Before Changes:
i = 0
s = hello
a[0] = 0

After Changes:
i = 0
s = hello
a[0] = 100
```

Passing Parameters by Value

- In this mechanism, when a method is called, a new storage location is created for each value parameter.
- The values of the actual parameters are copied into them.
- Hence, the changes made to the parameter inside the method have no effect on the argument.
- Example on the next slide demonstrates the concept.

Passing Parameters by Value

```
static void Main(string[] args)
{
      int a = 10, b = 20;

      Console.WriteLine("Before swapping:");
      Console.WriteLine("a: " + a);
      Console.WriteLine("b: " + b);

      Swap(a, b);

      Console.WriteLine("\nAfter swapping:");
      Console.WriteLine("a: " + a);
      Console.WriteLine("b: " + b);
}

static void Swap(int x, int y)
{
      int temp = x;    // save the value of x
      x = y;           // put y into x
      y = temp;        // put temp into y
}
```

```
Output:

Before swapping:
a: 10
b: 20

After Swapping:
a: 10
b: 20
```

Passing Parameters by Reference

- A reference parameter is a reference to a memory location of a variable.
- When you pass parameters by reference, unlike value parameters, a new storage location is not created for these parameters.
- The reference parameters represent the same memory location as the actual parameters that are supplied to the method.
- You can pass value type parameters as reference parameters using the `ref` keyword.
- Example on the next slide demonstrates the concept.

Passing Parameters by Reference

```
static void Main(string[] args)
{
      int a = 10, b = 20;

      Console.WriteLine("Before swapping:");
      Console.WriteLine("a: " + a);
      Console.WriteLine("b: " + b);

      Swap(ref a, ref b);

      Console.WriteLine("\nAfter swapping:");
      Console.WriteLine("a: " + a);
      Console.WriteLine("b: " + b);
}

static void Swap(ref int x, ref int y)
{
      int temp = x;     // save the value of x
      x = y;            // put y into x
      y = temp;         // put temp into y
}
```

```
Output:

Before swapping:
a: 10
b: 20

After Swapping:
a: 20
b: 10
```

Passing Parameters by Reference

- When the code on the previous slide is compiled and executed, it produces the following result:

```
Before swapping:
a: 10
b: 20

After Swapping:
a: 20
b: 10
```

- This shows that the values have changed inside the `Swap` method and this change reflects in the `Main` method.

- You also need to add the `ref` keyword when you invoke the method `Swap(ref a, ref b);`

- *Always remember, any variable must be initialized before it is passed into a method, whether it is passed in by value or by reference.*

Output Parameters

- C# requires that variables be initialized with a starting value before they are accessed, whether they are passed by value or passed by reference.
- Often, the starting value of the variable that is passed by reference is unimportant.
- That value will be overwritten by the method, which may never even look at the previous value.
- However, there is a way to circumvent the C# compiler's insistence on initial values for input arguments.
- You do this with the `out` keyword.
- When a method's input argument is prefixed with `out`, that method can be passed a variable that has not been initialized.
- The variable is passed by reference, so any changes that the method makes to the variable will persist when control returns from the called method.
- You must use the `out` keyword when you call the method, as well as when you define it.

Output Parameters

- A `return` statement is used for returning only one value from a method.
- Sometimes, there is a need for methods to be able to output more than one value.
- Again, this can be accomplished by using the output parameters.

Output Parameters

```
static void Main(string[] args)
{
    int a = 10, b = 20, sum, mul; // sum and mul aren’t initialized

    Calculate(a, b, out sum, out mul);

    Console.WriteLine("Sum = " + sum);
    Console.WriteLine("Mul = " + mul);
}

static void Calculate(int a, int b, out int sum, out int mul)
{
    sum = a + b;
    mul = a * b;
}
```

```
Output:

Sum = 30
Mul = 200
```

Exercise

- Write a method TestString that finds out if a string has period and/or comma.
- The TestString method accepts a string and two bool output parameters: hasPeriod and hasComma.
- The output parameters are set true if a period and/or comma is found in the string, false otherwise.

- You can check for a character in a string by using the Contains() method on a string, like this:

```
if (str.Contains(',')) { … }
```

- Incorporate the TestString method into an app that receives a string from the user and provide the appropriate outputs.

Exercise - Solution

```csharp
static void Main(string[] args)
{
      Console.Write("Enter a string that might contain a period and/or comma: ");
      string s = Console.ReadLine();

      TestString(s, out bool hasPeriod, out bool hasComma);

      if (hasPeriod || hasComma)
            Console.WriteLine("The string contains a period and/or comma");
      else
            Console.WriteLine("The string doesn't contain a period and/or comma");
}

static void TestString(string str, out bool hasPeriod, out bool hasComma)
{
      hasPeriod = hasComma = false;
      if (str.Contains('.'))
            hasPeriod = true;
      if (str.Contains(','))
            hasComma = true;
}
```

Parameter Arrays

- Some methods can benefit from having an unbounded number of arguments passed to them.
- `Console.WriteLine` method is an example of this.
- The first argument is a format string, which is a string with placeholders.
- The remainder arguments to fill in those placeholders depends on the string itself.

- For example:

```
Console.WriteLine("{0} + {1} = {2}", a, b, a + b);
```

- The use of format strings comes in handy when dealing with quite a bit of formatting, where string concatenation would become very ugly and unreadable:

```
Console.WriteLine(a + " + " + b + " = " + (a + b));
```

Parameter Arrays

```
Output:

Number of elements: 3
101 102 103

Number of elements: 10
0 1 2 3 4 5 6 7 8 9
```

```
static void Main(string[] args)
{
    int[] numbers = {101, 102, 103};

    PrintArray(numbers);                              // passing an array
    PrintArray(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);  // passing comma-
                                               separated arguments
}
static void PrintArray (params int[] num)
{
    Console.WriteLine("Number of elements: " + num.Length);

    foreach(int i in num)
        Console.Write(i + " ");
}
```

Named Arguments

- Invoking methods, the variable name need not be added to the invocation.
- However, if you have a method signature like the following to move a rectangle:

```
public void MoveAndResize(int x, int y, int width, int height)
{
        // method body
}
```

- And you invoke it with the following code snippet, it’s not clear from the method call what numbers are used for what:

```
rect.MoveAndResize(30, 40, 20, 40);
```

- You can change the invocation to make it immediately clear what the numbers mean:

```
rect.MoveAndResize(x: 30, y: 40, width: 20, height: 40);
```

Named Arguments

- Typically, parameters need to be passed into a method in the same order that they are defined.
- Named arguments allow you to pass in parameters in any order.

```
static string PersonInfo(string name, int age)
{
     Console.WriteLine("Name = " + name);
     Console.WriteLine("Age = " + age);
}
```

- The following two method calls will result in the same output:

```
PersonInfo("John", 34);
PersonInfo(age: 34, name: "John");
```

Optional Arguments

- Arguments can also be optional.
- While defining the method, supply a default value for optional parameters, which must be the last ones defined.

```
static void PersonInfo(string name, int age = 21)
{
     Console.WriteLine("Name = " + name);
     Console.WriteLine("Age = " + age);
}
```

- Different ways to call this method:

```
PersonInfo("John", 34);
PersonInfo("John");   // only passing the name; age will be 21
```

Optional Arguments

- Different ways to call a method with optional arguments.

```
static void PersonInfo(string name = "John", int age = 21)
{
        Console.WriteLine("Name = " + name);
        Console.WriteLine("Age = " + age);
}
```

```
// Omit the optional parameters
PersonInfo();

// Omit second optional parameter
PersonInfo("Mark");

// You can't omit the first but keep the second
// PersonInfo(37); // Not allowed

// Classic calling syntax
PersonInfo("Mark", 37);

// Specify one named parameter
PersonInfo(age: 37);

// Specify both named parameters
PersonInfo(age: 37, name: "Mark");
```

Expression-Bodied Methods

- If the implementation of a method consists of just one statement, C# gives a simplified syntax to method definitions: expression-bodied methods.
- You don’t need to write curly brackets and the `return` keyword with the new syntax.
- The operator `=>` (lambda operator) is used to distinguish the declaration of the left side of this operator to the implementation that is on the right side.

- **Link:**
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/expression-bodied-members

Expression-Bodied Methods

- The right side of the lambda operator defines the implementation of the method.
- Curly brackets and a `return` statement are not needed.
- What’s returned is the result of the statement, and the result needs to be of the same type as the method declared on the left side, which is a `double` in this code snippet:

```
public double FindSquare(double num) => num * num;
```

- The above code is same as:

```
public double FindSquare(double num)
{
        return num * num;
}
```

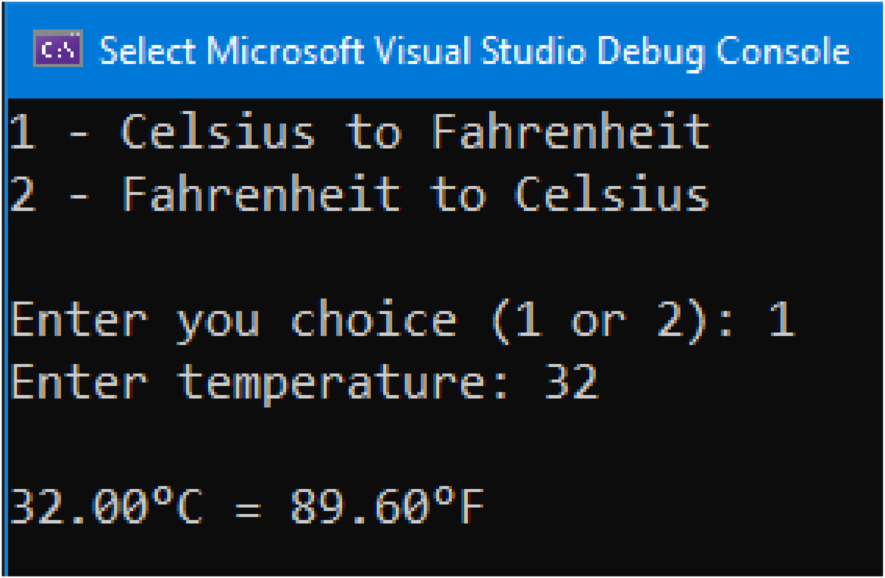
Do It Yourself!

- **Exercise: Temperature Conversions:**
- Implement the following methods:
  a) Method `Fahrenheit` returns the Fahrenheit equivalent of a Celsius temperature, using the calculation:

```
f = 9.0 / 5.0 * c + 32;
```

  b) Method `Celsius` returns the Celsius equivalent of a Fahrenheit temperature, using the calculation"

```
c = 5.0 / 9.0 * ( f - 32 );
```



```
1 - Celsius to Fahrenheit
2 - Fahrenheit to Celsius

Enter you choice (1 or 2): 1
Enter temperature: 32

32.00°C = 89.60°F
```

- Use the methods from parts (a) and (b) to write an app that enables the user either to enter a Celsius temperature and display the Fahrenheit equivalent or enter a Fahrenheit temperature and display the Celsius equivalent.
- Round off the output to 2 decimal places and display the unit (°F or °C).
- Display a degree symbol as well. (Unicode: `\u00B0`)

🛠 Do It Yourself!

- Exercise: **Parking Charges:**
- A parking garage charges a $2.00 minimum fee to park for up to three hours.
- The garage charges an additional $0.50 per hour for each hour or part thereof in excess of three hours.
- The maximum charge for any given 24-hour period is $10.00.
- Assume that no car parks for longer than 24 hours at a time.
- Write an app that calculates and displays the parking charges for each customer who parked in the garage.
- You should enter the hours parked for each customer. The app should display the charge for the current customer and should calculate and display the running total of previous receipts.
- The app should use method `CalculateCharges` to determine the charge for each customer.

Do It Yourself!

- Exercise: **Exponentiation:**
- Write a method `IntegerPower( base, exponent )` that returns the value of base^{exponent}.
- For example, `IntegerPower(3, 4)` calculates 3^4 (or 3 * 3 * 3 * 3).
- Assume that exponent is a positive integer and that base is an integer.
- Method `IntegerPower` should use a `for` or `while` loop to control the calculation.
- ***Do not use any Math-library methods.***
- Incorporate this method into an app that reads integer values for base and exponent and performs the calculation with the `IntegerPower` method.

🛠 Do It Yourself!

- **Exercise: Coin Tossing:**
- Write an app that simulates coin tossing.
- The app should call a method `Flip` that takes no arguments and returns `false` for tails and `true` for heads.
- Display the result.

- Upgrade the app by using loop which keeps asking the user whether they want to flip a coin again.
- Count the number of times each side of the coin appears.

🛠 Do It Yourself!

- Exercise: **Perfect Numbers:**
- An integer number is said to be a perfect number if its factors, including 1 (but not the number itself), sum to the number.
- For example, 6 is a perfect number, because 6 = 1 + 2 + 3.
- Write method `Perfect` that determines whether parameter value is a perfect number.
- Use this method in an app that determines and displays all the perfect numbers between 2 and 1000.

- Enhance the app so that it displays the factors of each perfect number to confirm that the number is indeed perfect.

🛠 Do It Yourself!

- **Exercise: Prime Numbers:**
- An integer is said to be prime if it’s greater than 1 and divisible by only 1 and itself.
- For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.
- Write a method that determines whether a number is prime.
- Use this method in an app that displays all the prime numbers less than 10,000.

Enumeration in C#

Introduction to Enums in C#

- An enumeration is a value type that contains a list of named constants, such as the `Color` type.
- The enumeration type is defined by using the `enum` keyword.

```
public enum Color
{
        Red,
        Green,
        Blue
}
```

Introduction to Enums in C#

- You can declare variables of `enum` types, such as the variable `c`, and assign a value from the enumeration by setting one of the named constants prefixed with the name of the `enum` type.

```
Color c = Color.Red;
Console.WriteLine(c);
```

```
Output:

Red
```

Introduction to Enums in C#

- In C#, `enum` is a value type data type.
- The `enum` is used to declare a list of named integer constants.
- It can be defined using the `enum` keyword directly inside a namespace, or class.
- The `enum` is used to give a name to each constant so that the constant integer can be referred using its name.
- By default, the first member of an `enum` has the value `0` and the value of each successive `enum` member is increased by `1`.
- You can also set your own custom integer values to `enum` members.
- `enum` allows us to create a new data type.

Why And When To Use Enums?

- Use `enum`s when you have values that you know aren't going to change, like months, days, colors, deck of cards, etc.

Introduction to Enums in C#

- By default, the first item of an enum has the value 0.
- The second has the value 1, and so on.

- To get the integer value from an item, you must explicitly convert the item to an int.

```
enum Day
{
        Sunday,         // 0
        Monday,         // 1
        Tuesday,        // 2
        Wednesday,      // 3
        Thursday,       // 4
        Friday,         // 5
        Saturday        // 6
}

static void Main(string[] args)
{
        Console.WriteLine((int)Day.Sunday);
        Console.WriteLine((int)Day.Monday);
}
```

```
Output:

0
1
```

Introduction to Enums in C#

- By default, the type behind the `enum` type is an `int`.
- The underlying type can be changed to other integral types (`byte`, `short`, `int`, `long` with signed and unsigned variants).

```
enum Day : short

{
        Sunday,                 // 0
        Monday,                 // 1
        Tuesday,                // 2
        Wednesday,              // 3
        Thursday,               // 4
        Friday,                 // 5
        Saturday                // 6
}
```

Introduction to Enums in C#

- We can specify custom constant values for the members of the `enum`.

```
enum Day

{
        Sunday = 10,              // 10
        Monday,                   // 11
        Tuesday,                  // 12
        Wednesday,                // 13
        Thursday,                 // 14
        Friday,                   // 15
        Saturday                  // 16
}
```

Introduction to Enums in C#

- You can also assign random integer values to `enum` members.

```
enum Day
{
      Sunday = 10,          // 10
      Monday = 2,           // 2
      Tuesday = 8,          // 8
      Wednesday = 15,       // 15
      Thursday = 4,         // 4
      Friday = 7,           // 7
      Saturday = 12         // 12
}
```

Day of the Week Example

```
enum Day
{
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}

class Program
{
    static void Main(string[] args)
    {
        Console.Write("Enter the day number (0-6): ");
        int dayNum = int.Parse(Console.ReadLine());

        Day d = (Day)dayNum; // type-cast int to enum Day

        Console.WriteLine("It is " + d);
    }
}
```

Output:

```
Enter the day number (0-6): 4
It is Thursday
```

Playing Cards Example

```
enum CardValue
{
      Two = 2,
      Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King, Ace
}

enum Suit
{
      Heart, Spade, Club, Diamond
}

class Program
{
      static void Main(string[] args)
      {
            Random rnd = new Random();
            CardValue cardValue = (CardValue)rnd.Next(2, 15);
            Suit suit = (Suit)rnd.Next(4);

            Console.WriteLine(cardValue + " of " + suit);
      }
}
```

```
Output 1:

Seven of Club
```

```
Output 2:

Ace of Diamond
```

Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

References

Material has been taken from:

- Visual C# 2012: How to Program:
- https://learning.oreilly.com/library/view/visual-c-2012/9780133380170/ch07.html

- Professional C# 7 and .NET Core 2.0:
- https://learning.oreilly.com/library/view/professional-c-7/9781119449270/c03.xhtml