C# INTERMEDIATE

METHODS

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Intermediate\_9\_Methods

{

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// NOTES

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// AGENDA

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Signature Methods

// Method Overloading

// Params Modifier

// Ref Modifier

// Out Modifier

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// SIGNATURE OF A METHOD

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// The Signature of a Method consists of its:

// Name

// Number of Parameters

// Type of Parameters

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// METHOD OVERLOADING

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method Overloading is when methods have the same name but different signatures.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// PARAM MODIFIERS

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// QUESTION? What happens when you want to create a Method that has a varying number of inputs.

// Such as if you want to add a varying number of numbers to add together.

// SOLUTION 1: You can use an Array of integers as an input (int[] numbers)

/\*

public class Calculator

{

public static int Add(int[] numbers)

{

// foreach Add function code

}

}

var result = Calculator.Add(new int[] { 1, 2, 3, 4 });

\*/

// SOLUTION 2: Use a Param Modifier.

/\*

public class Calculator

{

public static int Add()

{

// foreach Add function code

}

}

\*\*\* Call the Add function in either of the below ways:

\*\*\* TRADITIONAL WAY

var result = Calculator.Add( new int[] { 1, 2, 3, 4 });

\*\*\* NON-TRADITIONAL WAY

var result = Calculator.Add( 1, 2, 3, 4 );

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// REF MODIFIERS

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// The Ref Modifier instead of passing a copy of the variable value into a function

// passes a reference to that variable value into the function and when the

// function completes its coding, the variable itself has its value changed.

/\*

public class Weirdo

{

public void DoAWeirdThing(ref int a)

{

a += 2;

}

}

var a = 1;

Weirdo.DoAWeirdThing(a);

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// OUT MODIFIERS

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// When a Method has an 'OUT' parameter in it, then you also need to specify the 'out' keyword

// in the argument when you call that method.

// In that sense the Out Modifier is a little similar to the Ref Modifier.

// One reason that you may want to use the Out Modifier is that you can specify

// Sever Out return values in a Method, meaning that you can create a Method that can

// return multiple values.

// However, another approach to returning multiple values is to create a Method that

// returns a class and that class has properties that are the same as the 'out' return

// values.

// SYNTAX OF OUT MODIFIER

/\*

public class MyClass

{

public void MyMethod(out int result)

{

result = 1;

}

}

int a;

myClass.MyMethod(out a);

\*/

class Program

{

public class MyClass

{

public static void MyMethod(int score, out char Grade, out string success)

{

Grade = Convert.ToChar("N");

success = "TBD";

if(score >= 0 && score < 50)

{

Grade = Convert.ToChar("F");

success = "Fail";

}

else if(score >= 50 && score < 80)

{

Grade = Convert.ToChar("P");

success = "Pass";

}

else if(score >= 80 && score <= 100)

{

Grade = Convert.ToChar("A");

success = "Distinction";

}

else

{

Grade = Convert.ToChar("N");

success = "Score Not Valid";

}

}

}

static void Main(string[] args)

{

/\*

int score = 85;

char Grade = Convert.ToChar("N");

string success = string.Empty;

MyClass.MyMethod(score, out Grade, out success);

Console.WriteLine("A Score of " + score + ", has resulted in the following Results:" +

Environment.NewLine + "Grade = " + Grade + Environment.NewLine + "Results = " + success);

\*/

// The below code results in an exception

// var number = int.Parse("abc");

// In the below, we are using the OUT MODIFIER for a TryParse method

// You need to declare an integer variable first which is what will be

// used as the reference number and is the variable that stores the

// potentially transformed string into an integer.

// The input variable is the string "abc". This is the string that

// the TryParse method will try to convert to an integer and return

// a True or False depending on whether or not it was successful.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// PARSE AND TRY PARSE

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// The Parse method allows you to try to convert a string into

// another data type.

// If the Parse method is successful then it will return a

// value of the Data Type you are trying to convert to.

// If unsuccessful, it throws an exception.

// The TryParse method includes an OUT MODIFIER. If the

// the Conversion was successful, then the Method returns

// a TRUE, if unsuccessful, then it returns a FALSE.

// The Benefit of TryParse is that it does not throw an Exception, but

// it does require you to declare a variable of the Data Type you

// are trying to convert to before you use the TryParse method.

//==========================================================================

// These 2 Statements are essentially the same.

//==========================================================================

try

{

int num = int.Parse("abc");

}

catch(Exception)

{

Console.WriteLine("Conversion failed");

}

//==========================================================================

int number;

var result = int.TryParse("abc", out number);

if(result)

{

Console.WriteLine(number);

}

else

{

Console.WriteLine("Conversion failed");

}

}

static void UsePoints()

{

try

{

var point = new Point(10, 20);

point.Move(new Point(40, 60));

Console.WriteLine("Point is at ({0}, {1})", point.X, point.Y);

point.Move(100, 200);

Console.WriteLine("Point is at ({0}, {1})", point.X, point.Y);

}

catch(Exception)

{

Console.WriteLine("An unexpected error occurred");

}

}

static void UseParams()

{

var calculator = new Calculator();

Console.WriteLine(calculator.Add(1, 2));

Console.WriteLine(calculator.Add(1, 2, 3));

Console.WriteLine(calculator.Add(1, 2, 3, 4));

Console.WriteLine(calculator.Add(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }));

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Intermediate\_9\_Methods

{

public class Calculator

{

// USING THE PARAMS MODIFIER

public int Add(params int[] numbers)

{

var sum = 0;

foreach(var number in numbers)

{

sum += number;

}

return sum;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Intermediate\_9\_Methods

{

public class Point

{

public int X;

public int Y;

public Point(int x, int y)

{

this.X = x;

this.Y = y;

}

public void Move(int x, int y)

{

this.X = x;

this.Y = y;

}

public void Move(Point newLocatin)

{

// including the if statement with an Exception handling

// is known as DEFENSIVE PROGRAMMING

if(newLocatin == null)

throw new ArgumentNullException("newLocation");

Move(newLocatin.X, newLocatin.Y);

}

}

}