Flow control

Classes, Object and Methods

Objective

- After working through this session you should:
  - Understand Methods, Parameters and Return values.
  - Understand how to define a Class and create and delete an Object;
  - Understand the relationship between Classes, Objects and Methods;

Outline

- Methods / Functions/ Sub Tasks/ Programs
  - Defining
  - Parameters, defining and calling (input instructions)
  - Return types (void for NO return)
  - Returning values
- Class fundamentals
- Creating objects

Method in General

Purpose:

- perform a specific task
- perform a related group of sub tasks
- simplify a program via modularisation
- reuse code

Defining A Method

- Method is a simple task, method or function, in C#
  - Methods are declared within a class or struct by specifying the access level (default is public)
  - Method has the return type, which matches with the return value
    - string type for Console.ReadLine(), int type for Int32.Convert( ) or bool type for Int32.TryParse(…)
    - void type if a method has no return value, e.g., Main(), Console.WriteLine()
  - The name of the method specify it’s functionality or performed task
  - It contains a code block, which has a statement or a series of statements. They must contribute to complete the task.
  - and the method parameter list inside the pair of bracket (….)
  - The compulsory task in C# is public static void Main(string[] args)

Defining A Method Example

```
using System; // W7la.cs
public class Program
{       public static void Main(string[] args)
        {  // declare data variables
           string Name, Address, Phone;
           // get input name
           Console.Write("Enter Name: ");
           do
           {     Name = Console.ReadLine();
        } while( Name == "");
            // get input address
           Console.Write("Enter Address: ");
           do
           {     Address = Console.ReadLine();
         } while( Address == "");
            // get input phone
           Console.Write("Enter Phone: ");
           do
           {     Phone = Console.ReadLine();
                      } while( Phone == "");
           // display details
           Console.WriteLine("Person   : {0}", Name);
           Console.WriteLine("Address : {0}", Address);
           Console.WriteLine("Phone    : {0}", Phone);
        }
}
```

Compare to the version in
- W7lb.cs
- W7lc.cs

Method Parameters, defining and calling

- Parameters provide ***values to methods***.
- In computing, there are two kinds of parameters:
  - actual parameters
  - formal parameters

"Enter Name: "

- **Defining** Parameters

```
  static string GetString(string prompt) // formal parameter
 static void Display(string Name, string Address, string Phone)
```

- **Calling** Parameters

"Helen Smith"

"124 Albert Street"

"9111 2222"

```
string Name = GetString("Enter Name: "); // actual parameter
Display("Helen Smith", "123 Albert Street", "9111 2222")
```

Parameter

Each formal parameter:

- is a variable
- is specified in the method header
- has a modifier (optional – out / ref /val / params), type and name
- has local scope
- has actual initialised value provided to a method *when it is called*
- created when the method starts, and destroyed when the method ends

Function Return Types example

4.00000

1.000000

```
private double f( double x )  // x is formal parameter– system auto casting int to double
{
      return(  3*x*x + 2*x – 1   ); // calculate 3x^2+2x -1 expression bases on x value
}
```

4.000000

4.000000

```
private void displayPoints_1()
{      for( int x = 0; x <= 10; x++)
       {      Console.Write( x );
              Console.Write("\t");
              Console.WriteLine(  f (x )  ); // x is actual parameter from 0 to 10
       }
}
```

Function Return types

```
using System;
// using System.Windows.Forms;
public class w7ld
{
      private static double f ( double x )  {    return( 3*x*x + 2*x -1 );      }

      private static void displayPoints_1()
      { string strMsg = "";
        for( int x = 0; x <= 10; x++)  strMsg += String.Format( "{0}\t{1}\n", x, f(x) );
        Console.WriteLine(strMsg); // MessageBox.Show(strMsg);
      }

     public static void Main()
     {int x = 100;
        Console.WriteLine("The x value in the Main program: " + x);
        //MessageBox.Show ("The x value in the Main program: " + x.ToString());
        displayPoints_1();
     }
}
```

Parameter list example

```
static void DisplayPoints_2( double x1, double x2, double
        step )
{
        for( double x = x1;   x <= x2;   x = x + step )
                        Console.Write("x = {0}, f = {1}\n",   x,
        f(x));
}


  public static void Main()
  {
      DisplayPoints_2 (1, 10, 2) ; // display x 1, 3, 5, 7, 9
  } // w7le.cs
```

string format parameter list

```
public void displayPoints_2(double x1, double x2, double step )
{
        for( double x = x1;    x <= x2;    x = x + step )
                Console.Write("x = {1}, f = {0}\n",   f(x),   x  );
}
```

Note:

1. the special string format
2. the list of values
   - where {0} is the value of the 0th element, i.e., x
   - where {1} is the value of the 1st element, i.e., f(x)

Parameter Passing Modifiers

Most languages have 2 ways to pass parameters:

1. call-by-value
2. call-by-reference

C# has 4 modifications to pass parameters:

1. value // call-by-value, and the default read only
2. ref // call-by-reference read-write
3. out // call-by-reference write only
4. params // call-by-reference single array list

value modifier

- The actual parameter *value is copied into its corresponding formal parameter*.
- The original value *in the calling method* cannot be changed by the called method

```
using System;
public class w7lf
{
    static void swap (string a,  string b)
    {
        string temp = a;
        a = b;
        b = temp;
        Console.WriteLine("At the end of swap a: {0} b: {1}", a, b);
    }

     public static void Main()
     {
          string a = "Hello", b = "World";
          swap(a, b);
          Console.WriteLine("In Main after calling swap a: {0} b: {1}", a, b);
     }
}
```

ref modifier

The actual parameter:

- is a variable (not a value)
- can be modified / changed by the called method (using the name of the formal parameter)

```
using System;
public class w7lg
{
    static void swap (ref string a, ref string b)
    {
        string temp = a;
        a = b;
        b = temp;
        Console.WriteLine("At the end of swap a: {0} b: {1}", a, b);
    }

     public static void Main()
     {
         string a = "Hello", b = "World";
         swap(ref a, ref b); // a store contains b and b store a result
         Console.WriteLine("In Main after calling swap a: {0} b: {1}", a, b);
         // swap("Helen Smith", "John Anderson"); // give errors
         // when release comment fix text can’t store update or swap result
     }
}
```

out modifier

Like a ref parameter:

- the actual parameter:
  - is a variable
  - can be modified by the called method
- **but** the value of the variable is **not passed** to the method (*only data comes* **‘out’** *to store* ***results*** *from the method*).

```
using System;
public class w7lh
{
    static void swap (out string a, out string b)
    { // string temp = a;  // illegal a is output – can write but not get value
        a = "10";
        b = "100";
        Console.WriteLine("At the end of swap a: {0} b: {1}", a, b);
    }

     public static void Main()
     {
          string a = "Hello", b = "World";
          swap(ref a, ref b); // a store contains b and b store a result
          Console.WriteLine("In Main after calling swap a: {0} b: {1}", a, b);
          // swap("Helen Smith", "John Anderson"); // give errors
          // when release comment fix text can’t store update or swap result
     }
}
```

params modifier

- allows an **unspecified number of values** to be sent to a method
- data sent must be a **1D array** (or a **list** of values)

```
using System;
public class w7li
{
    static void Display( params double[] data )
    {
        double previous = data[0];
        Console.Write("Receiving data to Display: ");
        foreach( double x in data) Console.Write( "{0} ", x );
        Console.WriteLine("\n");
        data[ 0 ] = 123;   // changes array[ 0 ] too
        Console.WriteLine("At the end of Display Method data[0] change
from {0} to {1}\n", previous, data[0]);
    }

    public static void Main()
    {
        testing(); // check in the next page
    }
```

W7li.cs the testing()

```
 static void testing( )
 {
    double[ ] array = { 2, 4, 6, 8 };
    Display( array );             // send an array
    Console.WriteLine("After calling the Display back to Testing Method");
    foreach(int x in array) Console.Write("{0} - ", x);

    Console.WriteLine("\nThe 1st element on the array is change value to 123");
    Console.WriteLine("Calling the Display with a group of integer instance values");
    Display( 1, 2, 3, 4, 5, 6, 7);  // send a list and can't display the change of 1 to 123 as 1 is passing by value
    Console.WriteLine("Calling the Display with 13 to display");
    int a = 13;
    Display(a); // a convert to double and use like passing by value
    Console.WriteLine("In Testing Method After Display a is {0}\nNOT Change as passing an int parameter
  (a=13) acts like passing by value", a);
 }
}
```

Valid Method Calls

- ensure that the **number** of **actual parameters** match the *number of* formal parameters
  (except for the params modifier)
- ensure that the **type** of an **actual parameter** matches that of its *corresponding* formal parameter

```
private void test( string s, int a, int b, int c )
{
      …
}

private void m( )
{
      test( "abc", 2, 5, 8);          // valid
      test( "abc", 2);                // invalid missing b and c
      test( 2, 5, "abc", 8);          // invalid wrong type for s and b
}
```

Polymorphism – Function overloading

```
using System;
public class w7lj
{
    static int Add(int a, int b)   { Console.WriteLine("Adding integer {0} and {1}", a, b); return a+b; }
    static float Add(float a, float b)  { Console.WriteLine("Adding float {0} and {1}", a, b); return a+b; }
    static double Add(double a, double b)  { Console.WriteLine("Adding double {0} and {1}", a, b); return a+b; }
    static decimal Add(decimal a, decimal b) { Console.WriteLine("Adding decimal {0} and {1}", a, b); return a+b; }
    static string Add(string a, string b) { Console.WriteLine("Adding string {0} and {1}", a, b); return a+" "+b; }

    public static void Main()
    {
        Console.WriteLine( Add (1 ,2) );
        Console.WriteLine( Add (1.25f, 1.85f) );
        Console.WriteLine( Add (1.25, 1.85) );
        Console.WriteLine( Add (1.2m, 1.85m) );
        Console.WriteLine( Add ( "one", "two") );
    }
}
```

Class fundamentals

- A *class* is a program structure that defines an abstract data type
  - Must create the class first
  - Could have instance variables (private data attributes) of the class
  - Could have properties variable(public get or/and set method for private data attributes)
  - Could have functions/methods or operations or event procedure for GUI (e.g., click, focus, etc.)
- Object is an instance of a class. It has all data and method(s) which are defined by the class declaration.
  - Integer have an attribute as numeric type,
    - no decimal place, could be positive or negative,
    - Minimum value is -2147483648 and Maximum value is 2147483647
  - Integer have operator+, operator -, operator *, operator /, operator % , ToString(), etc.

Declare the class

```
 class ClassName
{
         private type _VariableName;     // private data attribute in SIT232
       ....
          public type VariableName { get; set; } // property
          ....
         public ClassName() { } // parameter-less constructor
        public ClassName( type variable,... ) // custom parameter list    constructor
        {      // set or store all parameters to the private Variable attributes
                VariableName  = variable;
                ....
        }

        ...// other methods , functions or  event procedures


}
```

Constructor

- **create one object (the major task)**
- **initialise data members**
- **perform one or more (sub) tasks to achieve the (major) task**
- a class may have **several constructors**
- For example, create a GUI with 2 Names, Persons, Buttons and a Label
  - create a Form object
  - create 2 Buttons object
  - create a Label object
  - add the Button and Label objects to the Form object

Constructor Format

```
access-level class_name( parameters )
{
 statements;
}
```

- access-level is **usually public**
  - public (for SIT102)
  - private (for SIT232)
  - protected (for SIT232)
  - internal (ignore for SIT102)
- **no type**
- the name is the **class name**
- 0 or more **parameters**

Calling Constructors

- require the **new operator**
- require the **class name**
- require any **parameters**

```
Banana b1, b2;
b1 = new Banana(); //constructor with parameter-less
b1.name = "B1";
b2 = new Banana("B2");
//constructor with 1 parameter B2 name
```

Destructor

- **destroy an object (the major task)**
- perform any **house keeping**
- a class has **only 1 destructor**
- Calling Destructors **in C#**
  - destructors **cannot be called by the programmer**
  - destructors are **invoked automatically:**
    - by the **garbage collector** (when an object is not being referenced)
    - and when the **program exits**

  - **ObjectName = null;**

Name Class example

```
using System;
public class Name
{
    //property
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Name()
    {
        FirstName = "Unknown First Name";
        LastName = "Unknown Last Name";
    }

    public Name(string NewLastName, string NewFirstName)
    {
        FirstName = NewFirstName;
        LastName = NewLastName;
    }
}
```

Program Class Example

```
public class Program
{    // no data or properties
     public static void GetData(string prompt, out string data)
     {   do
         {    Console.Write(prompt);
              data = Console.ReadLine();
         } while( data == "");
     }

     public static void Main()
     {
           // declare 2 string variables as test 3
           string LastName, FirstName;

          GetData("Enter First Name: ", out FirstName);
          GetData("Enter Last Name: ", out LastName);
          Console.WriteLine("After Get input data the name {0}, {1}", LastName.ToUpper(), FirstName);
```

Program Class Example using Name

```
       // declare aName object which create by using the public Name() constructor
        Name aName = new Name(); // FirstName and LastName will be Unknown

        Console.WriteLine("After Create the aName {0}, {1}", aName.LastName.ToUpper(),
     aName.FirstName);

        // declare aName object which create by using the public Name(string, string) constructor
       // use the FirstName and LastName variables in Main to create the anotherName object
        Name anotherName = new Name(FirstName, LastName);

        Console.WriteLine("After Create the another {0}, {1}", anotherName.LastName.ToUpper(),
     anotherName.FirstName);

        // notify the garbage collection to release the object
        aName = null;
        anotherName = null;
    }
}
```

Another example

Specifications:

- We need to keep a ***list of products*** that lets us track the products they have purchased and sold.
- Each product has product code, product description, and current in stock quantity, the purchase cost, the mark up rate.
- The new product must be added first before record the purchase or sale transaction. The new product adding must search to make sure no duplicated item is added.
- The Purchase and sale need to obtain the product code, and search for the recorded product to update its quantity.
- There is an option to display all products in the Inventory System.