FUNDAMENTALS OF PROGRAMMING WITH C#

MODULAR PROGRAMMING

Liu Fan
isslf@nus.edu.sg

Total:56

Objectives

- Write a program that organizes the instructions into methods
- Able to make a call across classes and methods

Agenda

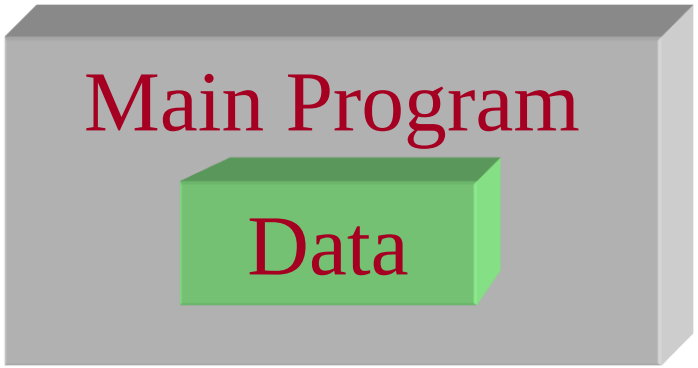
- Top Down Design
- Methods
  - Definition
  - Creation
  - Using
- Variables and their scope

Top Down Design

- Top Down Design is a formal process of decomposing a problem into simpler problems and then solving the simpler problems.
- This process is carried out repeatedly until the problems are so small that the solution is obvious.
- All languages provide facility for programmers to organise their programs into modules and also to integrate these.
- C# achieves this with the aid of User-Defined functions (methods)
- C# also use this for its object orientation

Unstructured Programming

- How do we learn programming?
  - I did it this way... (perhaps you too!)
    - Usually, people start learning programming by writing small and simple programs consisting only of one main program. Here "main program" stands for a sequence of commands or statements which modify data which is global throughout the whole program.



Drawbacks: Unstructured

- This programming technique can only be used in a very small program.

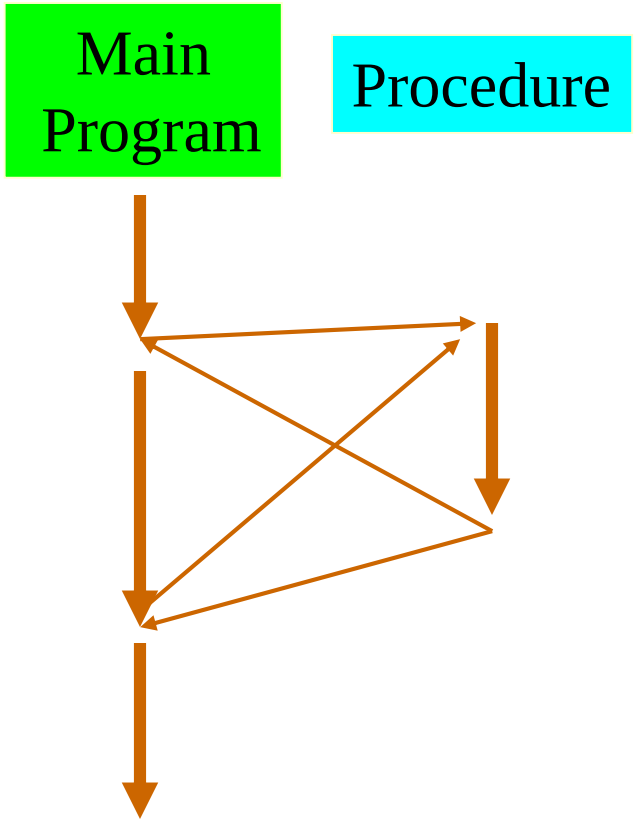
- For example, if the same statement sequence is needed at different locations within the program, the sequence must be copied. If an error needed to be modified, every copy needs to be modified.

- This has lead to the idea to extract these sequences(procedure), name them and offering a technique to call and return from these procedures.

Procedural Programming

- With procedural programming, you are able to combine sequences of calling statements into one single place.

- A procedure call is used to invoke the procedure. After the sequence is processed, flow of control proceeds right after the position where the call was made.



Procedures

- With parameters and sub-procedures (procedures of procedures) , programs can now be written more structured and error free.

- For example, if a procedure is correct, every time it is used it produces correct results.

- Consequently, in cases of errors you can narrow your search to those places which are not proven to be correct.

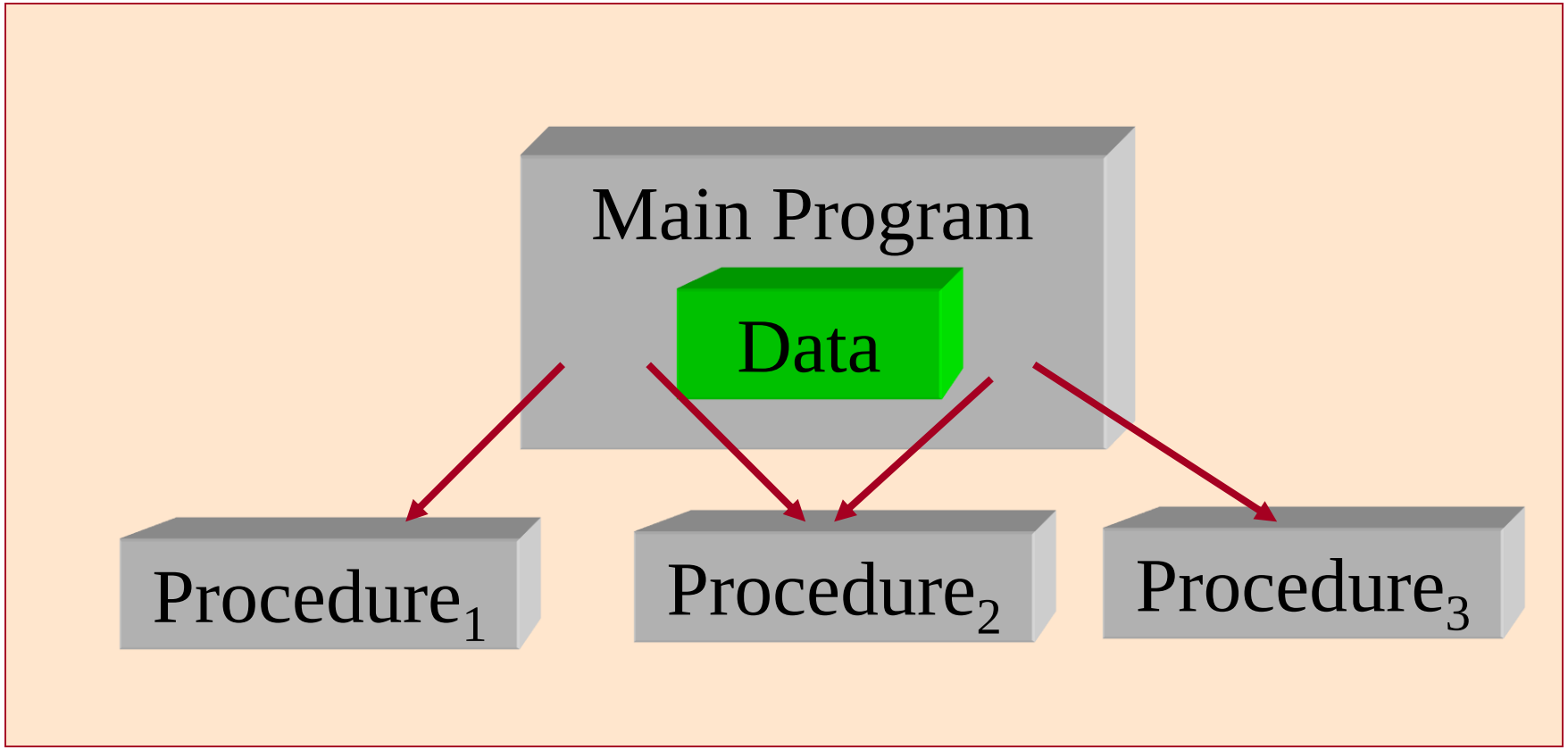
Procedure Program view

- Now a program can be viewed as a sequence of procedure calls.

- The main program is responsible to pass data to the individual calls, the data is processed by the procedures and the resulting data is presented.

- Thus, the flow of data can be illustrated as a hierarchical graph, a tree.

Procedure Program view



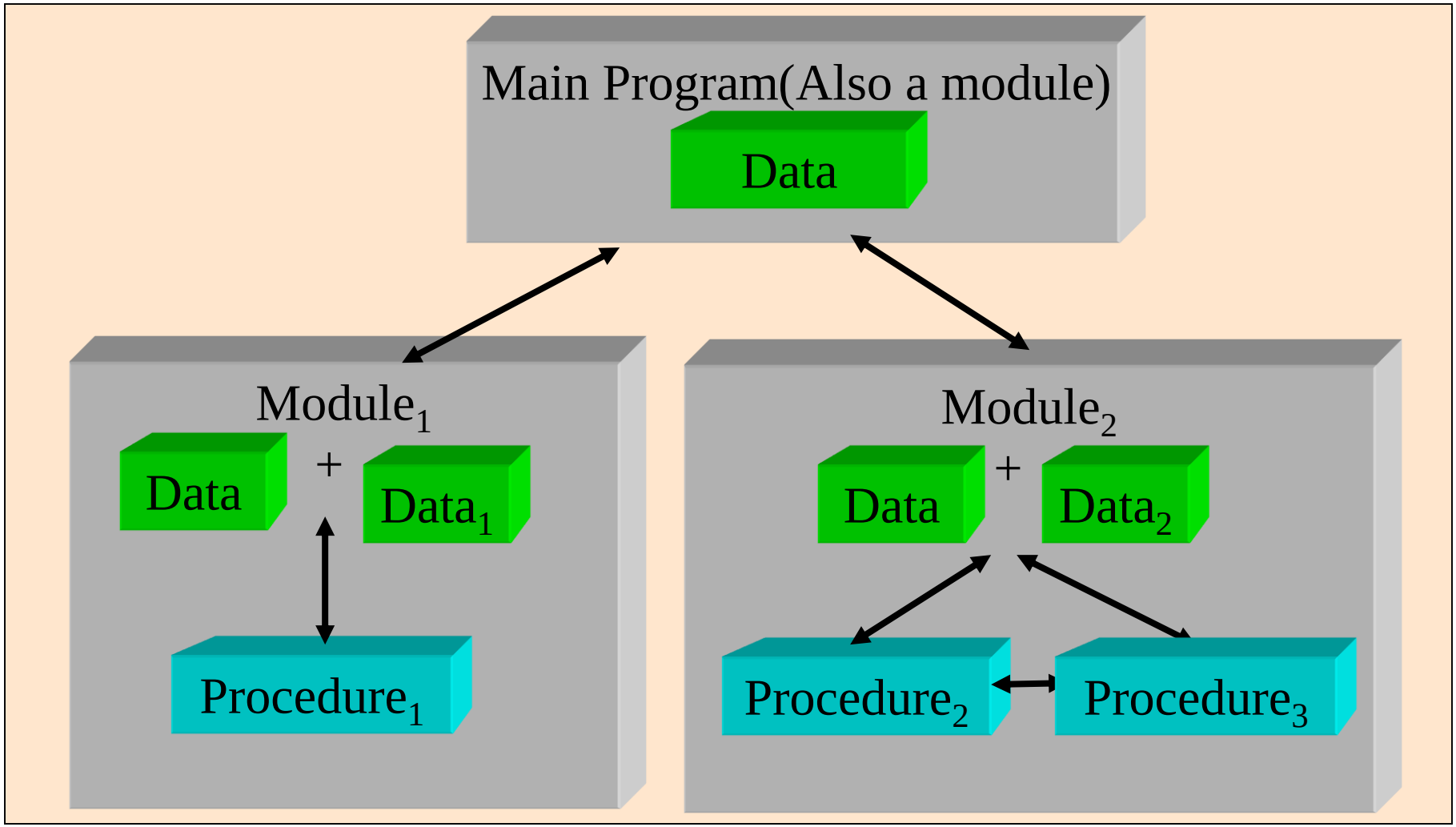
Modular Programming

- With modular programming, procedures of a common functionality are grouped together into separate modules.

- A program therefore no longer consists of only one single part. It is now divided into several smaller parts which interact through procedure calls and which form the whole program.

- The main program coordinates calls to procedures in separate modules and hands over appropriate data as parameters.

Modular Program View



Modular Programming

- Each module can have its own data. This allows each module to manage an internal state which is modified by calls to procedures of this module.

- However, there is only one state per module and each module exists at most once in the whole program.

Methods

- Methods may be thought as Object Oriented (OO) term for Functions in traditional Languages
  - The term Function is derived from a mathematical function.
  - Example:

    $f(x) = 2x^2 + 3x + 5$
  - We can obtain a single value of the above function for any given value of x.
  - Thus:

    f(5) => 70

    f(3) => 32 and so on

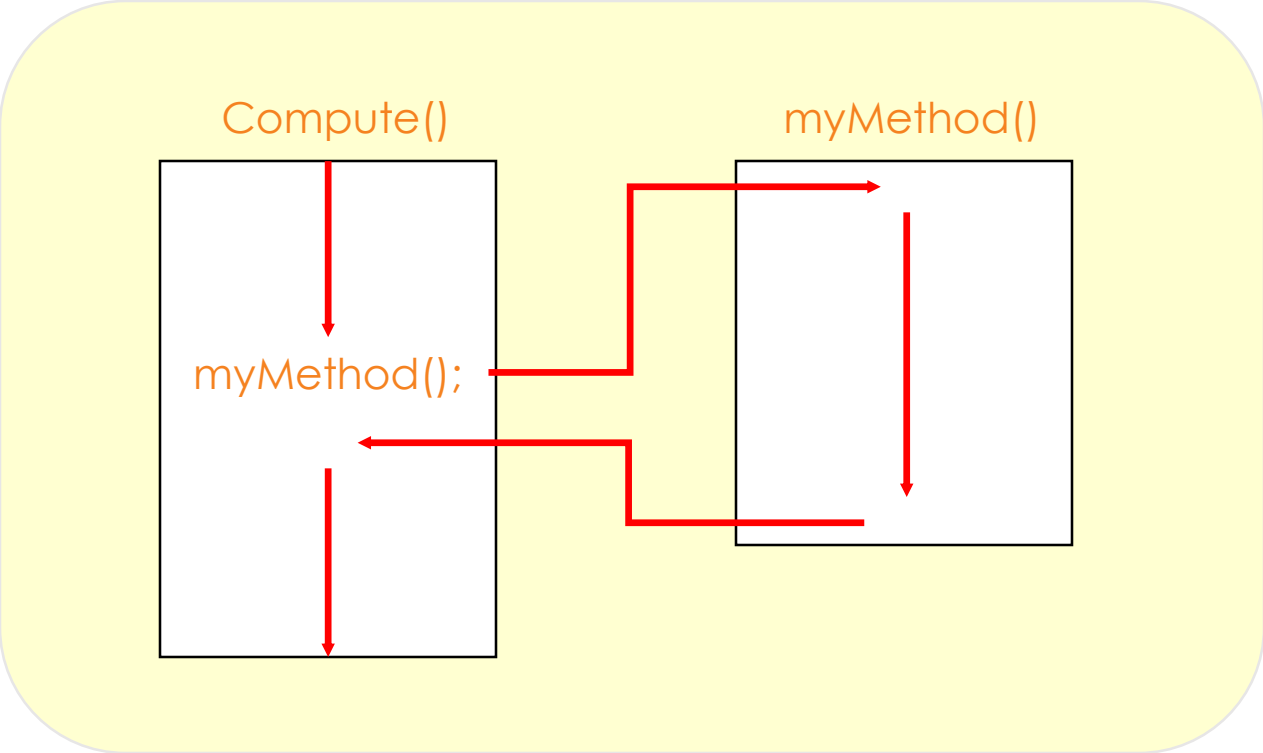
  - Thus if we require this function to be computed several times in our program, we need not re code the expression if we define this as a 'method'. We only need to call it every time we need the answer.

Methods

- A method declaration specifies the code that will be executed when the method is called (or invoked)
- When a method is invoked, the flow of control jumps to the method and executes its code
- When completed, the flow returns to the calling statement (i.e. the place where the method is called) and continues to the next line
- The invocation may or may not return a value, depending on the method definition

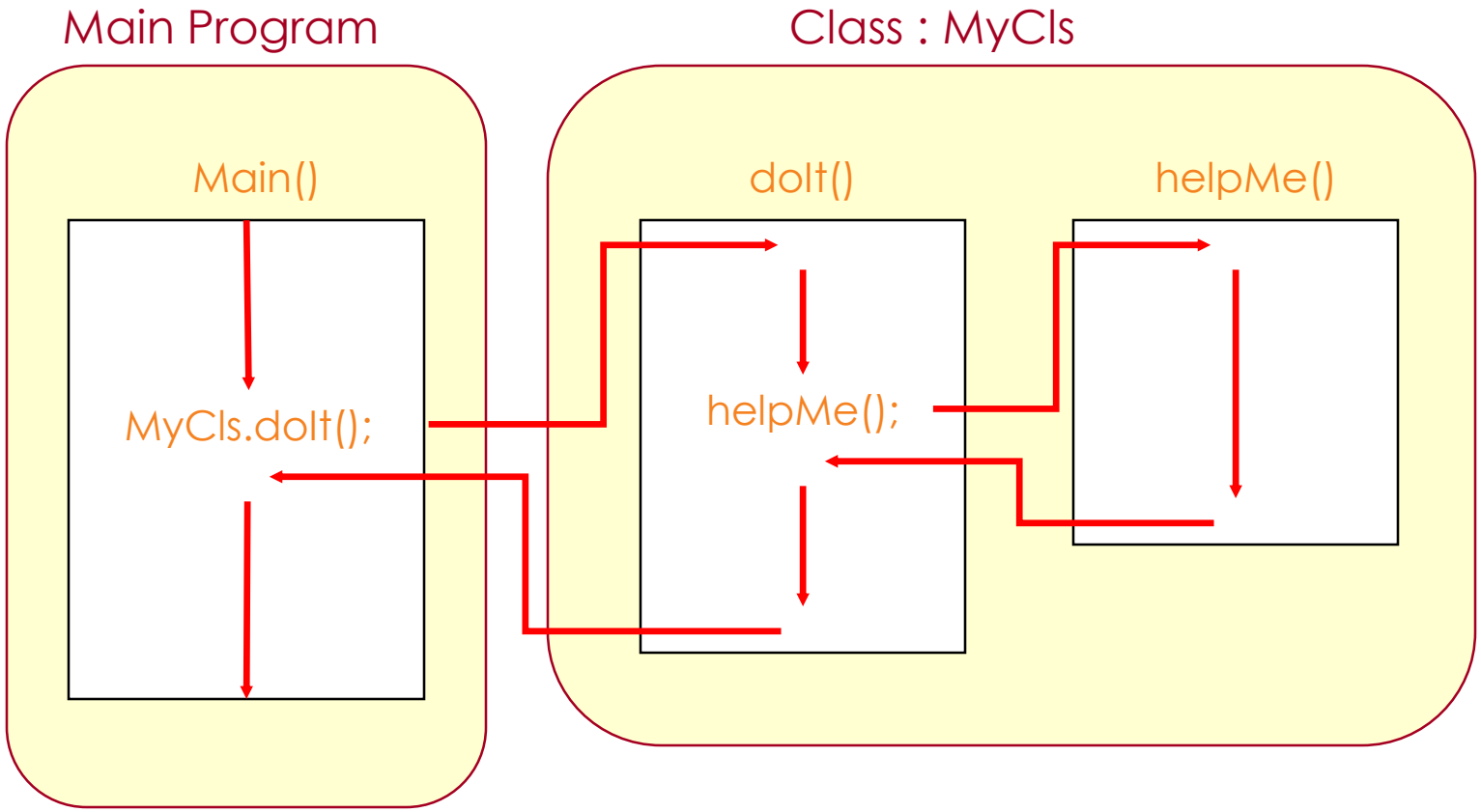
Method Control Flow

- The called method could be within the same class, in which case only the method name is needed to invoke it



Method Control Flow

- The called method can also be part of another class or object



Class Library vs User-Defined Methods

- We had already used Class Library Methods (Functions) like:
  - Math.Sqrt()
  - String.ToUpper()
  - Convert.ToInt()
- The use of User-Defined method is similar
- Standard Class Library Methods are provided by Microsoft while the User-Defined Methods are written by you.
- Like the Library Methods, the User Defined Method has:
  - Name
  - Arguments
  - Return Value (Type)
  - Argument Type
  - In addition, you define the activities in the function while coding the function.

Defining Our Own Methods

- Methods consists of three key aspects:
  - The Header
  - The Body
  - The 'return' statement (usually the last statement in the method).
- Methods are written inside a class
  - For example, the "Main( )" that we have been coding is a method.

```
static DataType MethodName ( DTypeN  Arg1,  DTypeN  Arg2 …)
{
    // Program statements
    return value;
}
```

Note:

We confine ourselves to what are called as "static" methods only during the Fundamentals of Programming module.

Methods Explained: Header

- static
  - We would use this modifier now as an essential prefix to all our methods
  - Meaning and alternatives would be explained in OOP
- DataType
  - This gives the type of value the method returns.
  - You may use any data type that is appropriate for this method.
  - If there is no return value then the keyword "void" has to be used for DataType.
- Method Name:
  - This is the identifier to the method.
  - You may coin a useful name to depict the action that the method performs
  - Naming validity Method is governed by the same rules as with variable name.
- DTypeN ArgN
  - Arguments that are passed to the method by the calling program.
  - ArgN represents a variable name where the passed in values are stored
  - DTypeN is the data type of the variable ArgN.
  - There can be any number of arguments. Each argument is separated by a comma
  - If there are no arguments then no variables are mentioned in parenthesis. However, the open/close parenthesis is essential and should not be omitted.

Methods Explained: Body

- Body
  - The Body of the method is enclosed inside a pair of curly braces i.e., { … }
  - The body can contain any statement that we are familiar with.
  - The body statements would represent the program logic and can be thought of as a program in itself.
  - The body should contain a return statement.

Methods Explained: return

- The return statement
  - The return statement marks the logical end to the program (method).
  - The return statement can be anywhere in the program
    - However there are compiler checks which takes precaution to ensure that at least one return statement would be hit during execution. For this reason it is advisable to safely place the return statement at the physical end of the method.
  - The return statement would return a value to the calling program.
    - The value returned would be placed after the keyword 'return'
    - The value can be either a constant, variable or expression.
    - The value should be of the same data type as mentioned in the Method Header.
    - If the method header data type is mentioned as 'void' then you should have a return without the value. The word 'void' need not be stated in return statement.

Methods: Organisation

- The following is the structure of a class having the Main() method and two user defined methods.

```
class MyClass
{
        static void Main()
        {
          // Program statement
        }

        static DataType Method1 (DType1 arg1, DType2 arg2 … )
        {
          // Program statements
        }

        static DataType Method2 (DType1 arg1, DType2 arg2 … )
        {
          // Program statements
        }
}
```

Methods: Example

- Write a method which when invoked with an integer argument, would return as string with a value "Even" if the integer passed is an even number else it would return "Odd".

- Approach 1:

```
static string NumberType(int num)
{
  if ((num % 2)==0)
    return "Even";
  else
    return "Odd";
}
```

Methods: Example

- Approach 2: The same method can be rewritten as:

```
static string NumberType(int num)
{
  string s = "";
  if ((num % 2)==0)
     s ="Even";
  else
     s = "Odd";
  return s;
}
```

Methods Example: Discussion

- Merits of First Approach
  - The First approach has the advantage of executing fewer statements for the obtaining the same result. Hence the first approach is computationally more efficient than the second.

- Disadvantage of First Approach
  - The first approach is not readably structured since there are more than one point of exits.
  - Some times the compiler may not detect that a "return" would be hit. Hence it might generate a compiler error
  - This example is straight forward so compiler detects both the returns and hence no error is generated.

- Merits of Second Approach:
  - More structured since the "return" statement is at the end
  - You can be more sure that program flows through and no statements are skipped.

- Weighing the pros and cons, you are highly recommended to take the second approach of placing return at the end.

Example: Calling the Method

- Why call?
  - The method that we wrote is not a main program.
  - Hence methods are called from another method, in our case usually the Main method.
- How to call?
  - We confine at the moment to static methods only.
  - Like with the Class Library we can call our method by:

```
ClassName.MethodName(args)
```

  - Like with the Class Library we can use the method call:
    - For assigning the return value to a variable.
    - Using the return value as part of Expression.
    - Using the return value as argument to another method.
    - Methods that return void are usually written as independent statement.
  - If the method is called from within the same class, then prefixing the ClassName is optional. i.e, the following would work:

```
MethodName(args)
```

Full Example

```
class MyFirstProgram
{
   static void Main()
   {
      for (int i=1;i <= 10; i++)
         Console.WriteLine(i + "\t" + NumberType(i) );

   }

   static string NumberType(int num)
   {
      if ((num % 2) == 0)
        return "Even";
      else
        return "Odd";
   }
}
```

Method Example : Void

- In this example we demonstrate a method that performs an action without returning a value.
- The method demonstrated here takes a string argument and prints the string on the console by centering it on the screen (assuming a screen width of 80).

```
static void CentredPrint(string s)
{
   int TotLen = ((80 - s.Length)/2) + s.Length;
   string r = s.PadLeft(TotLen,' ');
   Console.WriteLine(r);
   return;
}
```

Full Example: Void Method

```
class MyProgram10
{
static void Main()
  {
    CentredPrint("Institute of Systems Science");
    CentredPrint("National University of Singapore");
  }

  static void CentredPrint(string s)
  {
            int TotLen = ((80 - s.Length)/2) + s.Length;
            string r = s.PadLeft(TotLen,' ');
            Console.WriteLine(r);
            return;
  }
}
```

```
Institute of Systems Science
National University of Singapore
```

Passing Arguments - By Value

- Arguments can be passed
  - By Value
  - By Reference

- By Value
  - We have been using this all along without really mentioning it.
    - Any Variable that is mentioned as argument to a method is by default pass by value.
    - Here the value contained in the variable is sent to the called method and used inside this method.
    - Any changes in the corresponding variable in the method does not impact the variable value of the calling module.

Passing "By Value": Example

```
static void Main()
{
        int x = 5, y = 6;
        Console.WriteLine("Main: " + x + "\t" + y);
        Method1(x,y);
        Console.WriteLine("Main: " + x + "\t" + y);
}

static void Method1(int x, int y)
{
        Console.WriteLine("Mthd: " + x + "\t" + y);
        x = 999; y = 444;
        Console.WriteLine("Mthd: " + x + "\t" + y);
}
```

```
Main: 5         6
Mthd: 5         6
Mthd: 999       444
Main: 5         6
```

Passing Arguments - By Reference

- By Reference
  - To pass variable by reference we need to use the ref keyword before the variable name:
    - At the time of defining the method AND
    - At every time the method is invoked (called).
  - Any Variable that is mentioned as ref argument passes the memory reference to the method and not merely the value.
  - This means that the memory location used in the calling module and the called method is the same.
  - Irrespective of the variable names used in the calling and called method it has the same memory location.
  - Any changes in the corresponding variable in the method impacts the variable value of the calling module.

Passing "By Ref": Example

```
static void Main()
{
        int x = 5, y = 6;
        Console.WriteLine("Main: " + x + "\t" + y);
        Method1(ref x,ref y);
        Console.WriteLine("Main: " + x + "\t" + y);
}

static void Method1(ref int x, ref int y)
{
        Console.WriteLine("Mthd: " + x + "\t" + y);
        x = 999; y = 444;
        Console.WriteLine("Mthd: " + x + "\t" + y);
}
```

```
Main: 5         6
Method1: 5              6
Method1: 999    444
Main: 999       444
```

Variables and their behaviour

- Variables are only visible in a certain area - this area is known as their **scope**.
  - A variable that is declared in a method is local to that method only. That is this variable cannot be used in any other method.
  - A class variable is common across all the methods and this name is best not re-declared inside methods. If declared inside methods, the variable declared in the method would be treated as a local variable and would not to impact the class variable in any way. Thus changes to a locally declared variable is not reflected in the class variable.

    Note: The rigour of this would be appreciated in OOP Modules.

Variables Scope

- C# variables scope can be divided into three categories as follows:
  - Class Level Scope
  - Method Level Scope
  - Block Level Scope

Class Level Scope

- Class Variables are those that are declared outside the methods but inside a class.
- These variables are common to all methods and can be accessed from within any method.
- These variables are also termed as the **field variables** or **class members**.

```
  using System;
// declaring a Class
class classOne
{
  // this is a class level variable, having class level scope
  static int myFieldVariable = 5;

  static void Main(string[] args)
  {
        // accessing class level variable
        Console.WriteLine(myFieldVariable);
  }
}// here class level scope ends
```

Method Variables

- Variables that are declared inside a method have method level scope. These are not accessible outside the method.
- However, these variables can be accessed by the nested code blocks inside a method.
- These variables are termed as the **local variables**.
- These variables don’t exist after method’s execution is over.

Method Level Scope

```
using System;
// declaring a Class
class classOne
{ // from here class level scope starts
  static int myFieldVariable = 5;
    // declaring a method
    static void Main()

    { // from here method level scope starts

 // accessing method level variable, it will give compile time error
as you are trying to access the local variable of Add()

        Console.WriteLine("The sum of 3 and 5 is:" +
Add(myFieldVariable,myLocalVariable));


    } // here method level scope ends

    // declaring a method
    public void Add(int x, int y)

    { // from here method level scope starts

        int myLocalVariable = 3;
        return x+y;

    } // here method level scope ends

} // here class level scope ends
```

Block Level Scope

- These variables are generally declared inside the for, while statement etc.
- These variables are also termed as the **loop variables** or **statement variables** as they have limited their scope up to the body of the statement in which it declared.
- Generally, a loop inside a method has three level of nested code blocks(i.e. class level, method level, loop level).
- The variable which is declared outside the loop is also accessible within the nested loops. It means a class level variable will be accessible to the methods and all loops. Method level variable will be accessible to loop and method inside that method.
- A variable which is declared inside a loop body will not be visible to the outside of loop body.

Block Level Scope

```
using System;

//example to illustrate class level scope, method level scope
namespace Day6
{
    class Program4
    {
        static int myFieldVariable = 5;
        static void Main(string[] args)
        {
            int myLocalVariable = 3;
            Console.WriteLine("The sum of 3 and 5 is:" + Add(myFieldVariable, myLocalVariable));
            for (int j =0; j<5; j++)
            {
                Console.WriteLine(j);
            }

            // this will give error as block level variable cann't be accessed outside the block
            // Console.WriteLine(j);
        }
        static int Add(int x, int y)
        {
            return x + y;
        }
    }
}
```

Delegate

- Delegate is a datatype that represent a method of a particular signature.
- Delegate allows a method with a particular signature to be stored like a variable and passed as method parameter

Delegate

- Example

```
delegate int IntOps(int n);
```

- Means that we are defining a new datatype called IntOps that represent method that accept int and return an int
- Once we declare this, we can create a method that uses this delegate type

```
void ApplyOperation(int[] arr, IntOps ops)
```

Delegate

- We can use the parameter just like a normal method in our method

```
static void ApplyOperation(int[] arr, IntOps ops){

    for (int i=0; i<arr.Length; i++) {
        arr[i] = ops(arr[i]);
    }

}
```

Delegate

- Calling the method that accepts a delegate
- Uses an existing method in a class.

```
static void Main() {
    int[] A = new int[] {1, 2, 3};
    IntOps myOp = Add10;

    ApplyOperation(A, myOp);
    //A should contain {11, 12, 13} now.
}

static int Add10(int x) {
    return x + 10;
}
```

This method match the signature of the IntOps delegate

Delegate

- Calling the method that accepts a delegate
- Uses anonymous method.

```
static void Main() {
   int[] A = new int[] {1, 2, 3};
   IntOps myOp = delegate(int a)
       {
           return a + 10;
       };
   ApplyOperation(A, myOp);
   //A should contain {11, 12, 13} now.
}
```

Anonymous method instead of a named method belong to a class.

Delegate

- Calling the method that accepts a delegate
- Uses lambda expression.

```
static void Main() {
    int[] A = new int[] {1, 2, 3};
    IntOps myOp = (int a) => { return a + 10; };
    ApplyOperation(A, myOp);
    //A should contain {11, 12, 13} now.
}
```

Lambda expression which means the same thing as the anonymous method

Structured Programming in Context

Cohesion and Coupling

- The design concepts of cohesion and coupling becomes relevant during the current discussion. The programmatic implications are taken for discussion
- Cohesion
  - The extent to which the activities within the module is closely related.
    - The degree of interaction within a module--the glue that holds a module together
- Coupling
  - The extent to which two modules are tied to each other.
    - The degree of interaction between modules--the wiring connecting the modules
- Aim for loose coupling and tight cohesion.

Business Case

- Organise your Business Programs discussed during the course of this module into sub-programs and comment on the cohesion and coupling aspects of the way you have organised your program.

APPENDIX

Summary

- Methods is used to organize programs into modular and reusable set of instructions
- Methods allow bigger program to be broken into smaller well-defined programs
- You can call methods that belong to another class, as long as the method is public
- Delegate allow you to pass a method with a particular signature as an argument to another method or assign it into a variable

Passing by value and by ref vs. value type and reference type

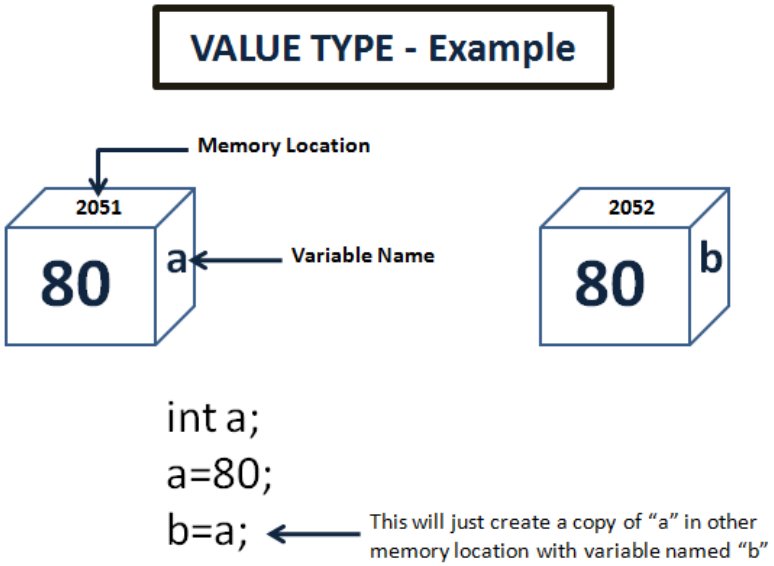
	By Value	By Reference
Update the value of a value-type variable (int, double)	No	Yes
Update the value of an element in a reference-type variable (arrays, objects, string) e.g. changing the value of one of the element of an array	Yes	Yes
Replace the entire content of a reference-type variable. e.g. replacing the array with a bigger array	No	Yes

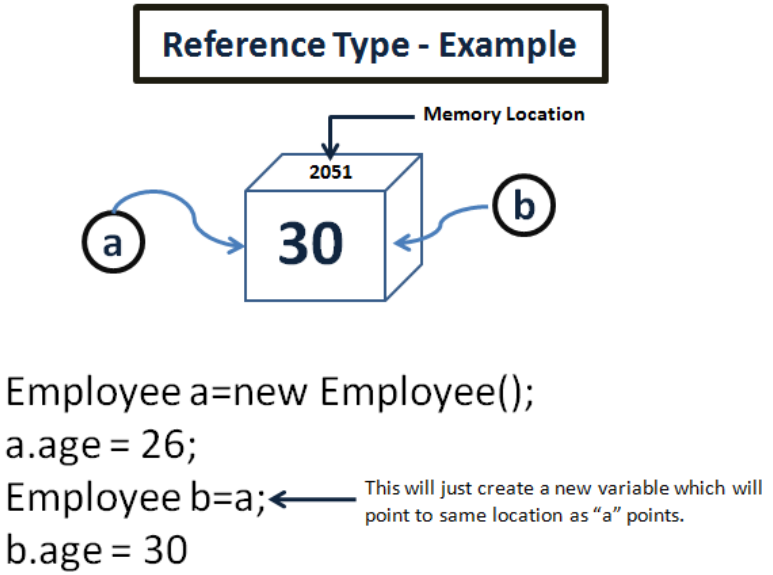
Revision on Types and Variables

- There are two kinds of variable of types in C#: *value types and reference types*.
  - Variables of *value types* directly contain their data whereas variables of *reference types* store references to their data (object).
- With *reference types*, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable.
- With *value types*, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other (except in the case of *ref* and *out* parameter variable)

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types-and-variables

Revision on Types and Variables





Value Type stores its contents in memory allocated on the stack. When you created a Value Type, a single space in memory is allocated to store the value and that variable directly holds a value. If you assign it to another variable, the value is copied directly and both variables work independently.

Reference Types are used by a reference which holds a reference (address) to the object but not the object itself. Because reference types represent the address of the variable rather than the data itself, assigning a reference variable to another doesn't copy the data. Instead it creates a second copy of the reference, which refers to the same location of the heap as the original value. Reference Type variables are stored in a different area of memory called the heap.

https://manojbhoir.wordpress.com/2015/09/29/value-type-and-reference-types/

Revision on Types and Variables

- Value types
  - Simple types
    - Signed integral: short, int, long
    - Unsigned integral: byte, ushort, uint, ulong
    - Unicode characters: char
    - IEEE binary floating-point: float, double
    - High-precision decimal floating-point: decimal
  - Boolean: bool
  - Enum types
    - User-defined types of the form enum E {...}
  - Struct types
    - User-defined types of the form struct S {...}
  - Nullable value types
    - Extensions of all other value types with a null value

Revision on Types and Variables

- Reference types
  - Class types
    - Ultimate base class of all other types: object
    - Unicode strings: string
    - User-defined types of the form class C {...}
  - Interface types
    - User-defined types of the form interface I {...}
  - Array types
    - Single- and multi-dimensional, for example, int[] and int[,]
  - Delegate types
    - User-defined types of the form delegate int D(...)