Method

SESSION 8

Session Objectives

Recognize the use of method

Learn the concept of passing values to methods

Differentiate between user defined methods and standard methods

Differentiate between passing-by-value and passing-by-reference

Functions/Methods

The **method** describes the mechanisms that actually perform its tasks. The method hides from its user the complex tasks that it performs, just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster.

It break up programs into smaller modules

Each module is a separate program

Modules are connected to the main program

Methods (Example)

Program to accept the details of an employee to calculate the salary

Fields are:

- Employee number
- Employee name
- Grade
- Basic Salary
- Standard allowance
- Standard deductions

Salary slip contains:

- Employee number
- Employee name
- Grade
- Basic Salary
- Allowances
- Net salary
- Deductions

Methods (Example Contd.)

Following calculations should be made:

- Net salary = (Basic + Allowances) - Deductions
- Allowances = House Rental Allowance(HRA) + Standard Allowance
- Deduction = Standard deductions + Provident Fund
- Provident fund is deducted at the rate of 6% of the Basic salary
- HRA and the allowance are based on the grade

Grade	HRA
1	18% of Basic salary
2	15% of Basic salary
3	Nil

Methods (Example Contd.)

The above problem, if written in single module, will be too complex

This program is split into two smaller programs called:

- Provident Fund Calculation
- HRA Calculation

Methods (Example Contd.)

For calculating the salary of an employee:

Start
emp_no=0
emp_name
grade=0
basic=0
std_allow=0
std_ded=0
net_sal=0
allow=0
deduct=0
HRA=0
PF=0
reply = ‘y’

(Contd…)

Methods (Example Contd.)

```
while reply='y'
        do
                Accept emp_no, emp_name, grade, basic, std_all, std_ded
                PF = PF_CAL()
                HRA = HRA_CAL()
                allow = HRA + std_allow
                deduct = std_ded + PF
                net_sal = (basic + allow) - deduct
                display emp_no, emp_name, grade, basic, allow, deduct, net_sal
                display "Do you wish to continue(y/n)?"
                Accept reply
        enddo

End
```

Methods (Example Contd.)

Functions **PF_CAL()** and **HRA_CAL()**

```
PF_CAL ()
Start
if ( grade <> 3 )
      PF = .06 * basic
else
      PF = 0
endif
Return PF
```

```
HRA_CAL ()
Start
if ( grade = 1 )
      HRA = .18 * basic
else
      if ( grade = 2 )
            HRA = .15 * basic
      else
            HRA = 0
      endif
endif
Return HRA
```

Advantages of Methods

- Maintains code efficiently
- Eases in understanding the code
- Eliminates redundancy of code
- Makes code reusable

Parameters of a Methods

Function to add two values and display their sum:

```
Start
Add(2, 5)
End

Add(x is an integer, y is an integer)
Start
Declare sum as an integer
sum = x + y
Display sum
return
```

- Add(x is an integer, y is an integer) accepts two parameters, x and y as integers
- Add(2, 5) is function call with 2 and 5 as the values passed

Parameters of a Methods (Contd.)

The parameters of a function should be passed in the right number and right order

Example:

```
Start
Declare num as an integer
num = 10
Add(2, 5)
Add('9', 4, 6)        } Errors
Add(2, 5, num)
End
Add(x is integer, y is integer, z is integer)
Start
sum is an integer
sum = x + y + z
Display sum
return
```

Parameters of a Methods (Contd.)

Add(2, 5) results in an error as the function Add() is declared with 3 parameters but the function call passes only 2 values

Add(‘9’, 4, 6) results in an error because ‘9’ is a character

Add(2, 5, num) does not result in an error because num is a variable of the type integer

Return Value

Values returned by the function to the main program

A function can pass only one value back to the calling program

Return Value (Example)

To find the sum of two numbers and return the sum to the main program

```
Start
sum is an integer
sum = Add(2,5)
Display sum
End

Add(x is an integer, y is an integer) returns integer
Start
var1 is an integer
var1 = x + y
return var1
```

`void` Methods and Value-Returning Methods

A `void` method is one that simply performs a task and then terminates.

```
Console.WriteLine("Hi!");
```

A value-returning method not only performs a task, but also sends a value back to the code that called it.

```
int number = Covert.ToInt32("700");
```

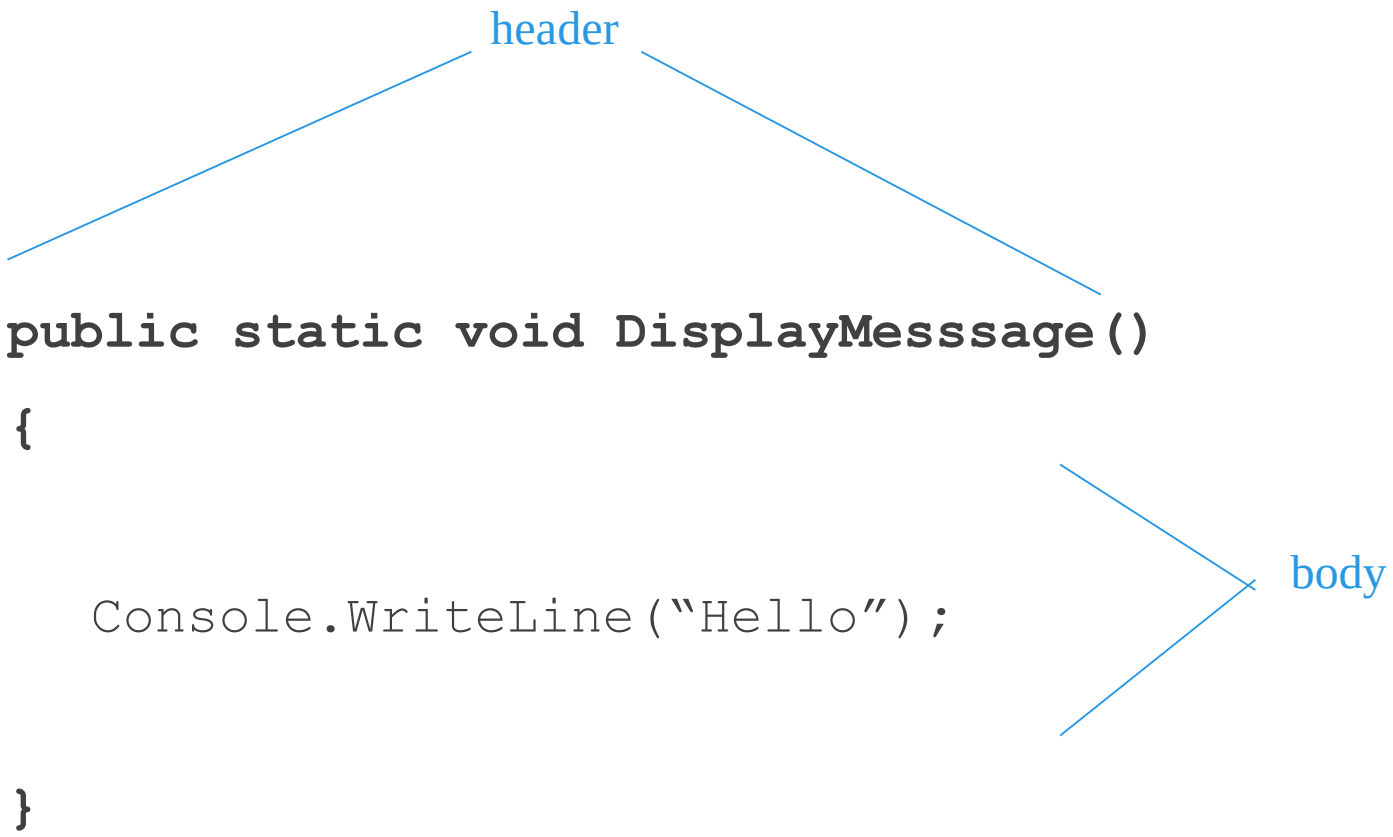
Defining a `void` Method

To create a method, you must write a definition, which consists of a header and a body.

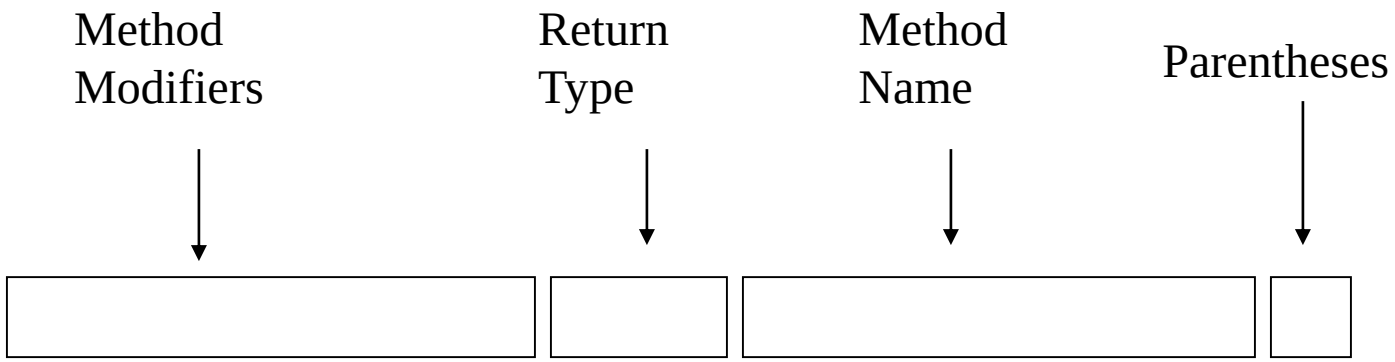
The method header, which appears at the beginning of a method definition, lists several important things about the method, including the method's name.

The method body is a collection of statements that are performed when the method is executed.

Two Parts of Method Declaration



Parts of a Method Header




```
public static   void  DisplayMessage ()
{
        Console.WriteLine("Hello");
}
```

Parts of a Method Header

Method modifiers

- `public`—method is publicly available to code outside the class
- `static`—method belongs to a class, not a specific object.

Return type—void or the data type from a value-returning method

Method name—name that is descriptive of what the method does

Parentheses—contain nothing or a list of one or more variable declarations if the method is capable of receiving arguments.

Calling a Method

A method executes when it is called.

The Main method is automatically called when a program starts, but other methods are executed by method call statements.

```
DisplayMessage();
```

Notice that the method modifiers and the void return type are not written in the method call statement. Those are only written in the method header.

Passing Arguments to a Method

Values that are sent into a method are called arguments.

```
Console.WriteLine("Hello");
number = Convert.ToInt32(str);
```

The data type of an argument in a method call must correspond to the variable declaration in the parentheses of the method declaration. The parameter is the variable that holds the value being passed into a method.

By using parameter variables in your method declarations, you can design your own methods that accept data this way.

Passing 5 to the **displayValue** Method

```
DisplayValue(5);
```

The argument 5 is copied into the parameter variable **num**.

```
public static void DisplayValue(int num)
{
 Console.WriteLine("The value is {0}", num);
}
```

The method will display **The value is 5**

Argument and Parameter Data Type Compatibility

When you pass an argument to a method, be sure that the argument's data type is compatible with the parameter variable's data type.

C# will automatically perform widening conversions, but narrowing conversions will cause a compiler error.

```
double d = 1.0;

DisplayValue(d);
```

Error! Can't convert `double` to `int`

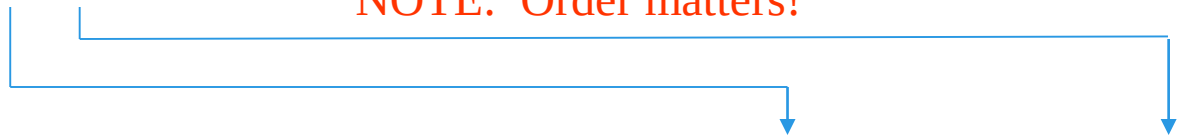
Passing Multiple Arguments

The argument 5 is copied into the **num1** parameter.

The argument 10 is copied into the **num2** parameter.

```
ShowSum(5,10);
```

NOTE: Order matters!



```
public static void ShowSum(double num1, double num2)
{
  double sum; //to hold the sum
  sum = num1 + num2;
  Console.WriteLine("The sum is {0}", sum);
}
```

Local Variables

A local variable is declared inside a method and is not accessible to statements outside the method.

Different methods can have local variables with the same names because the methods cannot see each other's local variables.

A method's local variables exist only while the method is executing. When the method ends, the local variables and parameter variables are destroyed and any values stored are lost.

Local variables are not automatically initialized with a default value and must be given a value before they can be used

Optional Parameters

As of .Net Framework 4 onwards, methods can have **optional parameters** that allow the calling method to vary the number of arguments to pass.

An optional parameter specifies a **default value** that's assigned to the parameter if the optional argument is omitted.

You can create methods with one or more optional parameters. *All optional parameters must be placed to the right of the method's non-optional parameters*—that is, at the end of the parameter list.

public int *Power(* ***int*** *baseValue,* ***int*** *exponentValue =* ***2*** *)*

Named Parameters

Normally, when calling a method that has optional parameters, the argument values—in order—are assigned to the parameters from left to right in the parameter list.

```
public void SetTime( int hour = 0, int minute = 0, int second = 0 )
```

```
t.SetTime(); // sets the time to 12:00:00 AM
t.SetTime( 12 ); // sets the time to 12:00:00 PM
t.SetTime( 12, 30 ); // sets the time to 12:30:00 PM
t.SetTime( 12, 30, 22 ); // sets the time to 12:30:22 PM
```

Recursion

A **recursive method** is a method that calls itself, either directly or indirectly through another method.

As an example of recursion concepts, consider the factorial of a nonnegative integer *n*, written *n!* (and pronounced "*n* factorial"), which is the product

$$n * (n - 1) * (n - 2) * \ldots * 1$$

$5! = 5 * 4 * 3 * 2 * 1$
$5! = 5 * (4 * 3 * 2 * 1)$
$5! = 5 * (4!)$

Example

```
using System;
public class FactorialTest
{
   public static void Main(string[] args)
 {
        for ( long counter = 0; counter <= 10; counter++ )
                Console.WriteLine( "{0}! = {1}", counter, Factorial( counter ));
 }
public static long Factorial(long number)
   {
         if (number <= 1)
             return 1;
          else
             return number * Factorial(number - 1);
   }
}
```

Returning a Value from a Method

Data can be passed into a method by way of the parameter variables. Data may also be returned from a method, back to the statement that called it.

```
int num = Convert.ToInt32("700");
```

The string "700" is passed into the ToInt32 method.

The `int` value 700 is returned from the method and stored into the `num` variable.

Defining a Value-Returning Method

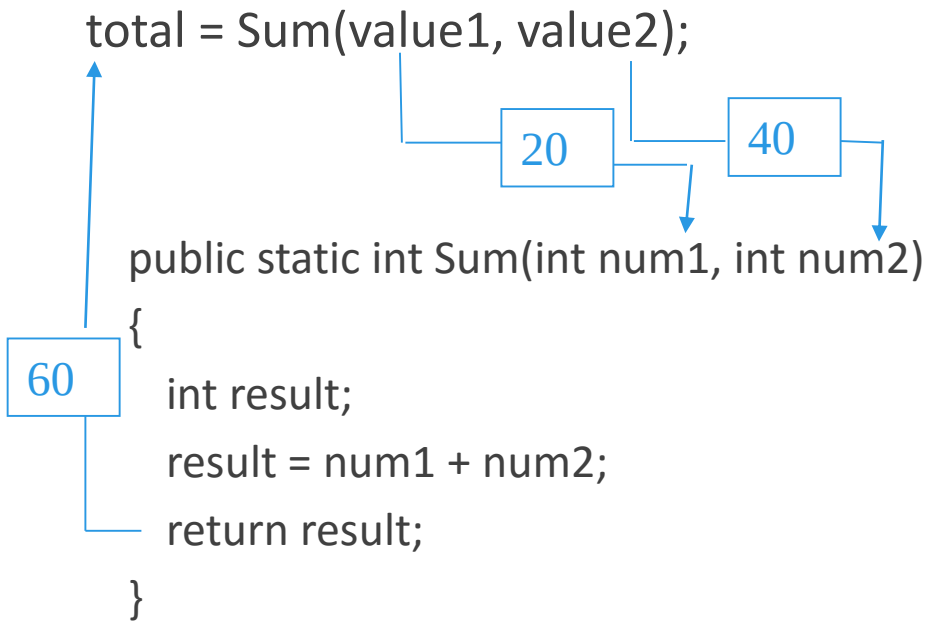
```
public static int Sum(int num1, int num2)
{
 int result;
 result = num1 + num2;
 return result;
}
```

Return type

The return statement causes the method to end execution and it returns a value back to the statement that called the method.

This expression must be of the same data type as the return type

Calling a Value-Returning Method



Methods

User-defined Methods

- Are written by programmers keeping the requirements in mind

Standard methods

- Are packaged with the tool
- Perform often-required actions
- Examples:
  - System.Console.WriteLine(“Methods”)
  - MessageBox.Show(“message”) in C#

Function Libraries

Collection of functions

Contains functions that deal with a particular issue. For example, a **Math** function library will have functions that deal with mathematical functions such as square root, average, exponent etc.

Once incorporated into the application, the statements for the function need not be reproduced in the program

Directly the function with the correct number and type of parameters can be called

A group of functions used very often are put in a function library

Base Class Library

- It is a mass of huge pre-written code that we can easily incorporate, and use in our applications.
- Base Class Library is shared among all the .net supported languages.
- Classes in BCL are categorized into namespaces based on their functionality.

Base Class Library (2)

- Most commonly used namespaces:

Namespace	Assembly	What it contains?
System	mscorlib.dll System.Web.dll	Much of the functionality of the BCL is contained within this namespace. Contains various other namespaces within it, some of them are as listed below.
System.Array	mscorlib.dll	Contains classes for manipulating arrays
System.Threading	mscorlib.dll System.dll	Contains classes for Multi-Threading
System.Math	mscorlib.dll	Contains classes for performing mathematical functions
System.IO	mscorlib.dll System.dll	Contains classes for reading and writing to files and streams
System.Reflection	mscorlib.dll	Contains classes for reading metadata from assemblies
System.Net	System.dll	Contains classes for Internet access and socket programming

System.Array namespace (1)

- Provides us with classes and methods for manipulating arrays.

```
using System;

class Test
{
static void Main()
      {
      int[] MyArray= {6,4,7,3,5,1,2};

      Console.WriteLine("Contents of Array before sorting:\n");
      DispMe(MyArray); //Displays the Contents of the Array

      Array.Sort(MyArray); //Sorts the Array

      Console.WriteLine("\n\nContents of Array after sorting:\n");
      DispMe(MyArray); //Displays the Contents after sorting
      }

public static void DispMe(Array pArray)
      {
      foreach(int t in pArray)
            {
            Console.WriteLine(t);
            }
      }

}
```

System.Array namespace (2)

- Output:

```
Contents of Array before sorting:

6
4
7
3
5
1
2

Contents of Array after sorting:

1
2
3
4
5
6
7
```

System.Array namespace (3)

- Other methods of the System.Array class:

Method	Syntax	Activities
Clear	Void Array.Clear(*ArrayName, int index, int length*);	Sets a range of elements in the array to zero or null.
Copy	void Array.Copy(Source array, Source index, Destination array, Destination index, number of elements to be copied)	Copies a range of elements from an array starting at the specified source index and pastes them to another array starting at the specified destination index.
LastIndexOf	int Array.LastIndexOf(*array, searchElement*)	Returns the index of the last occurrence of the value in the Array.
Reverse	void Array.Reverse(*arrayName*)	Reverses the order of the elements in an **Array**.

System.Math namespace

- Class Math provides a collection of methods that enable you to perform common mathematical calculations

Method	Description	Example
`Abs( x )`	absolute value of *x*	`Abs( 23.7 )` is 23.7 `Abs( 0.0 )` is 0.0 `Abs( -23.7 )` is 23.7
`Ceiling( x )`	rounds *x* to the smallest integer not less than *x*	`Ceiling( 9.2 )` is 10.0 `Ceiling( -9.8 )` is -9.0
`Cos( x )`	trigonometric cosine of *x* (*x* in radians)	`Cos( 0.0 )` is 1.0
`Exp( x )`	exponential method e^x	`Exp( 1.0 )` is 2.71828 `Exp( 2.0 )` is 7.38906
`Floor( x )`	rounds *x* to the largest integer not greater than *x*	`Floor( 9.2 )` is 9.0 `Floor( -9.8 )` is -10.0
`Log( x )`	natural logarithm of *x* (base *e*)	`Log( Math.E )` is 1.0 `Log( Math.E * Math.E )` is 2.0
`Max( x, y )`	larger value of *x* and *y*	`Max( 2.3, 12.7 )` is 12.7 `Max( -2.3, -12.7 )` is -2.3
`Min( x, y )`	smaller value of *x* and *y*	`Min( 2.3, 12.7 )` is 2.3 `Min( -2.3, -12.7 )` is -12.7
`Pow( x, y )`	*x* raised to the power *y* (i.e., x^y)	`Pow( 2.0, 7.0 )` is 128.0 `Pow( 9.0, 0.5 )` is 3.0
`Sin( x )`	trigonometric sine of *x* (*x* in radians)	`Sin( 0.0 )` is 0.0
`Sqrt( x )`	square root of *x*	`Sqrt( 900.0 )` is 30.0
`Tan( x )`	trigonometric tangent of *x* (*x* in radians)	`Tan( 0.0 )` is 0.0

System.IO namespace (1)

- It provides us classes to operate on files and directories.

```
using System;
using System.IO;

class Test
{
static void Main(string[] args)
      {
      FileInfo[] F;

      DirectoryInfo DI=new DirectoryInfo(".");
      F = DI.GetFiles("*.cs");

      foreach(FileInfo t in F)
            {
            Console.WriteLine(t);
            }
      }
}
```

- Output:

```
Scooby.cs
Properties.cs
MailChecker.cs
MyEditor.cs
```

System.IO namespace (2)

```
using System;
using System.IO;

class Test
{
static void Main(string[] args)
      {
      Console.WriteLine(@"Creatin' Directory C:\Scooby ...");

      Directory.CreateDirectory(@"c:\Scooby");
      DateTime CreationDate = Directory.GetCreationTime(@"c:\Scooby");

      Console.WriteLine("Directory Created on : " +
CreationDate.ToString());
      }
}
```

```
Creatin' Directory C:\Scooby…
Directory Created on : 19-Nov-2019  10:49:59 PM
```

Passing Arguments: Pass-by-Value vs. Pass-by-Reference

Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**.

When an argument is passed by value (the default in C#), a *copy* of its value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller.

This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems.

When an argument is passed by reference, the caller gives the method the ability to access and modify the caller's original variable.

Pass-by-Values

To accept a number and change its value in the function:

Start
num is an integer
Display “Enter a number”
Accept num
Display the value in num before change
change(pass num by value)
Display the value in num after change
End

change(var is an integer)
Start
var = 10
Display the value in var
return

Example 2 Cont…..

change(pass num by value)

- Variable num is passed to the function by value

change(var is an integer)

- Value of num is copied onto the variable var

The statement ‘Display the value in var’ displays the value of var to be 10

The statement ‘Display the value in num after change’ displays the original value of num i.e. the accepted number from the user

Pass-by-Reference

- To pass a variable by reference, C# provides keywords **ref** and **out**.
- Applying the ref keyword to a parameter declaration allows you to pass a variable to a method by reference—the called method will be able to modify the original variable in the caller.
- The ref keyword is used for variables that already have been initialized in the calling method. Normally, when a method call contains an uninitialized variable as an argument, the compiler generates an error.
- Preceding a parameter with keyword out creates an **output parameter**. This indicates to the compiler that the argument will be passed into the called method by reference and that the called method will assign a value to the original variable in the caller.
- If the method does not assign a value to the output parameter in every possible path of execution, the compiler generates an error. This also prevents the compiler from generating an error message for an uninitialized variable that's passed as an argument to a method.
- A method can return only one value to its caller via a return statement, but can return many values by specifying multiple output (ref and/or out) parameters.

Ref Vs Out

Ref	Out
The parameter or argument must be initialized first before it is passed to ref.	It is not compulsory to initialize a parameter or argument before it is passed to an out.
It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method.	A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method.
Passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter.	Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method.
It is not compulsory to initialize a parameter value before using it in a calling method.	A parameter value must be initialized within the calling method before its use.

Pass-by-Reference (Example)

To accept a number and change its value in the function:

Start
num is an integer
Display “Enter a number”
Accept num
Display the value of num before change
change(pass num by reference)
Display the value of num after change
End

change(var is an integer)
Start
var = 10
Display the value in var
return

Pass-by-Reference (Example Contd.)

change(pass num by reference)

- Function change accepts a reference of the variable num

change(var is an integer)

- Variable var is a reference to the variable num

The statement ‘Display the value in var’ displays the value of var to be 10

The statement ‘Display the value in num after change’ will also display 10 as the variable var is a reference to the variable num

Example

```
using System;
class ReferenceAndOutputParameters {
 public static void Main( string[] args ) {
        int y = 5;
        int z;
        Console.WriteLine( "Original value of y: {0}", y );
        Console.WriteLine( "Original value of z: uninitialized\n" );
        SquareRef( ref y ); // must use keyword ref
        SquareOut( out z ); // must use keyword out
        Console.WriteLine( "Value of y after SquareRef: {0}", y );
        Console.WriteLine( "Value of z after SquareOut: {0}\n", z );
        Square( y );
        Square( z );
        Console.WriteLine( "Value of y after Square: {0}", y );
        Console.WriteLine( "Value of z after Square: {0}", z );
}
```

```
static void SquareRef(ref int x )
{
        x = x * x;
}
static void SquareOut(out int x)
{
        x = 6;
        x = x * x;
}
static void Square(int x )
{
        x = x * x;
}
}
```

Output:

```
Original value of y: 5
Original value of z: uninitialized
Value of y after SquareRef: 25
Value of z after SquareOut: 36
Value of y after Square: 25
Value of z after Square: 36
```

Method Overloading

Methods of the same name can be declared in the same class, as long as they have different sets of parameters (determined by the number, types and order of the parameters). This is called **method overloading**

Method Overloading

- Three ways of overloading methods:
  - Specifying different number of parameters
  - Specifying different types of parameters
  - Specifying different sequence of parameters

```
…
public class semester
{
private int duration;

public semester()
      {
      duration = 144;
      }

public semester(int d)
      {
      duration = d;
      }

public setDuration(int newDuration)
      {
      duration = newDuration;
      }
}
…
```

Method Overloading (2)

- By specifying different number of parameters:

```
…
public void Admission(int t, string s)
      {
      //Admission 1 Implementation
      //Method takes two parameters
      }
…
public void Admission(int t, string s, int a)
      {
      //Admission 2 Implementation
      //Method takes three parameters
      }
..
public Admission()
      {
      //Admission 3 Implementation
      //Method takes in no parameters
      }
…
```

Method Overloading (3)

- By specifying different types of parameters:

```
...
public void Admission(short t, string s)
      {
      //Admission 1 Implementation
      //Method takes two parameters, but of type Short & String
      }
...
public void Admission(int t, string s)
      {
      //Admission 2 Implementation
      //Method takes three parameters, but of type int & string
      }

...
public Admission(bool b, int t)
      {
      //Admission 3 Implementation
      //Method takes in no parameters, but of type bool & int
      }
...
```