Web Application Development

2- Introduction to C# Programming Language

Dr. Abdullah Al-Hamdani

2- Introduction to C# Programming Language

- C# Basics:
  - Data Types, Comments, operations, I/O, and statements, functions, Strings
- Selection and Repetition Statements
- Arrays
- Classes and methods
- Exceptions
- Namespaces
- Collections and Generics
- Windows Applications

First C# Program

```
1  // Fig. 3.1: Welcome1.cs
2  // Text-displaying app.
3  using System;
4
5  class Welcome1
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       Console.WriteLine("Welcome to C# Programming!");
11    } // end Main
12 } // end class Welcome1
```

```
Welcome to C# Programming!
```

Escape Characters

```
1  // Fig. 3.10: Welcome2.cs
2  // Displaying one line of text with multiple statements.
3  using System;
4
5  class Welcome2
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       Console.Write("Welcome to ");
11       Console.WriteLine("C# Programming!");
12    } // end Main
13 } // end class Welcome2
```

```
Welcome to C# Programming!
```

C# Comments

- Same as in C++
  - // This is a comment
  - /*
    This is a
    multiline comment
    */
- XML-based Comments:
  - Use special tag names and used to automatically generate XML-based documentations about C# applications
  - can be used by tools like Visual Studio IntelliSense or documentation generators (e.g., DocFX, Sandcastle) to provide rich, structured documentation for your code
  - Starts with three slashes

```
/// <summary>
/// This application provides web pages
/// for student activities.
/// </summary>
```

Generating XML Documentation

```
/// <summary> The Hello class prints a greeting
/// on the screen
/// </summary>
class XML_Comments
{
  /// <remarks> We use console-based I/O.
  /// For more information about WriteLine, see
  /// <seealso cref="System.Console.WriteLine"/>
  /// </remarks>
  public static void Main( )
  {
    Console.WriteLine("This is XML Comments");
  }
}
```

Common Data Types

C# Type Name	VB Type Name	.Net Type Name
byte	Byte	Byte
short	Short	Int16
int	Integer	Int32
long	Long	Int64
float	Single	Single
double	Double	Double
decimal	Decimal	Decimal
char	Char	Char
string	String	String
bool	Boolean	Boolean
*	Date	DateTime
*	*	TimeSpan
object	Object	Object
dynamic		Type is according to the value

Declarations in C#

- Same as in C++ and Java

```
int x;
string name,s2;
name="Ali";
object abc;
object obj = new object();
public string name;
```

- Literal values
  - M (decimal), D (double), F (float), L (long)
  - E.g. double x = 1.3D;
- All-purpose var key

```
var myDecimal = 14.5M;
```

Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp2 {
   internal class Program    {
      static void Main(string[] args){
         short w = 0;
         int x = 5;
         Console.WriteLine($"Type for w is {w.GetType()}");
         Console.WriteLine($"Type for x is {x.GetType()}");
            dynamic m = 0L; Console.WriteLine($"Type for m is {m.GetType()}");
            m = "zzzzz"; Console.WriteLine($"Type for m is {m.GetType()}");
            m = 0.1F; Console.WriteLine($"Type for m F is {m.GetType()}");
            m = 0.1M; Console.WriteLine($"Type for m M is {m.GetType()}");
            m = 0.1D; Console.WriteLine($"Type for d D is {m.GetType()}");
         }
   }
}
```

Statements

- **Operators, assignment and initialization**
  - Same as in C++ and Java
  - E.g. `x++;` → `x= x +1`
    `++k;` → `k = k +1`
    `x--;` → `x = x -1;`
    `w+=3;` → `w = w + 3;`
    `m*=2;` → `m = m * 2;`
- **Strings and Escape character**
  - Same as in C++ and Java `\" \n \t \\`
    - E.g. `path = "c:\\myApp\\myfiles";`
  - String as methods similar to that in C++ and Java
    - e.g. `s.Substring(2,4);` and `s.ToUpper();`
  - Turn off C# and Java escaping by preceding a string with @
    - E.g. `path = @"c:\myApp\myfiles";`

First C# Program

```
1  // Fig. 3.11: Welcome3.cs
2  // Displaying multiple lines with a single statement.
3  using System;
4
5  class Welcome3
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       Console.WriteLine("Welcome\nto\nC#\nProgramming!");
11    } // end Main
12 } // end class Welcome3
```

```
Welcome
to
C#
Programming!
```

Formatting Output

- The .NET platform supports a style of string formatting slightly akin to the printf() statement of C and Python.
  - Numerous occurrences of tokens such as {0} and {1} embedded within various string literals.

```
//Prints: 9, Number 9, Number 9
Console.WriteLine("{0}, Number {0}, Number {0}", 9);

// Prints: 20, 10, 30
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
```

Formatting Numeric Data

String Format Character	Meaning in Life
C or c	Used to format currency. By default, the flag will prefix the local cultural symbol (a dollar sign [$] for U.S. English).
D or d	Used to format decimal numbers. This flag may also specify the minimum number of digits used to pad the value.
E or e	Used for exponential notation. Casing controls whether the exponential constant is uppercase (E) or lowercase (e).
F or f	Used for fixed-point formatting. This flag may also specify the minimum number of digits used to pad the value.
G or g Stands	Used for *general*. This character can be used to format a number to fixe or exponential format.
N or n	Used for basic numerical formatting (with commas).
X or x	Used for hexadecimal formatting. If you use an uppercase X, your hex format will also contain uppercase characters.

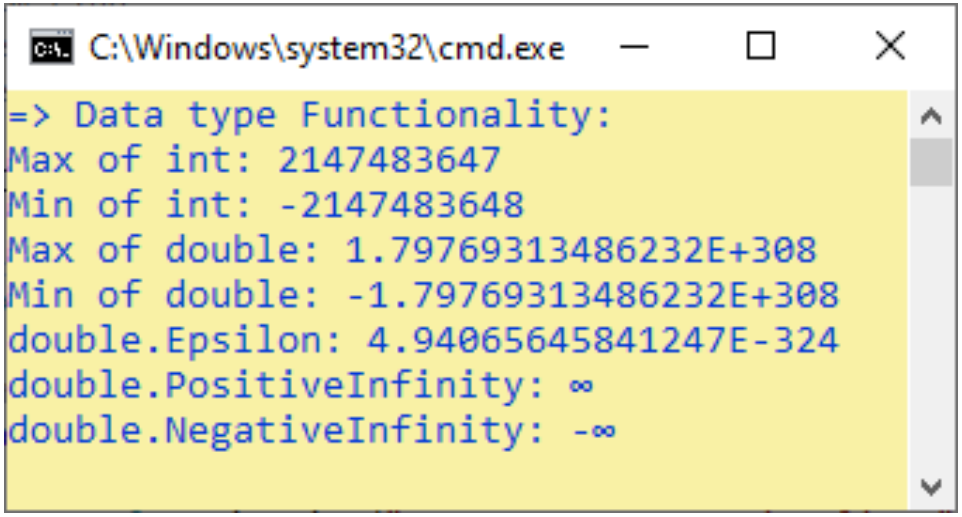
```
// Now make use of some format tags.
static void FormatNumericalData(){
    Console.WriteLine("The value 99999 in various formats:");
    Console.WriteLine("c format: {0:c}", 99999);
    Console.WriteLine("d9 format: {0:d9}", 99999);
    Console.WriteLine("f3 format: {0:f3}", 99999);
    Console.WriteLine("n format: {0:n}", 99999);
    // Notice that upper- or lowercasing for hex
    // determines if letters are upper- or lowercase.
    Console.WriteLine("E format: {0:E}", 99999);
    Console.WriteLine("e format: {0:e}", 99999);
    Console.WriteLine("X format: {0:X}", 99999);
    Console.WriteLine("x format: {0:x}", 99999);
}
```

```
The value 99999 in various formats:

c format: $99,999.00
d9 format: 000099999
f3 format: 99999.000
n format: 99,999.00
E format: 9.999900E+004
e format: 9.999900e+004
X format: 1869F
x format: 1869f
```

Members of Numerical Data Types

```
Console.WriteLine("=> Data type Functionality:");
Console.WriteLine("Max of int: {0}", int.MaxValue);
Console.WriteLine("Min of int: {0}", int.MinValue);
Console.WriteLine("Max of double: {0}", double.MaxValue);
Console.WriteLine("Min of double: {0}", double.MinValue);
Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
Console.WriteLine("double.PositiveInfinity: {0}", double.PositiveInfinity);
Console.WriteLine("double.NegativeInfinity: {0}", double.NegativeInfinity);
```



Formatting Output

- The $ special character identifies a string literal as an interpolated string.
  - When an interpolated string is resolved to a result string, items with interpolation expressions are replaced by the string representations of the expression results.

```
string name = “Salem";
int age = 34;
Console.WriteLine($"Your name is {name}.");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
   //Your name is Salem.
   // Salem is 34 years old.
```

- The @ special character serves as a verbatim/literal identifier.
  - Quote escape sequence ("") is not interpreted literally;

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";
Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
// c:\documents\files\u0066.txt
// c:\documents\files\u0066.txt
```

Example of String interpolation

```
1  // Fig. 3.13: Welcome4.cs
2  // Inserting content into a string with string interpolation.
3  using System;
4
5  class Welcome4
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       string person = "Paul"; // variable that stores the string "Paul"
11       Console.WriteLine($"Welcome to C# Programming, {person}!");
12    } // end Main
13 } // end class Welcome4
```

```
Welcome to C# Programming, Paul!
```

Type Casting and Conversion

- All types of input data are received from keyboard as string.
- Therefore, there is a need to convert them to the corresponding type.
- Examples:

```
double d = 5.4321;

int n = (int)d;

string s = n.ToString();

int n = int.Parse("54321");

string snum = "54.321";

double d = double.Parse(snum);
```

Output/output Statements

- **Display on Screen**
  - Console.WriteLine(……);
  - Console.Write(……);

- **Read from keyboard**
  - Console.ReadLine()
  - Convert to numbers (e.g. int or double)
    - int.Parse(…)

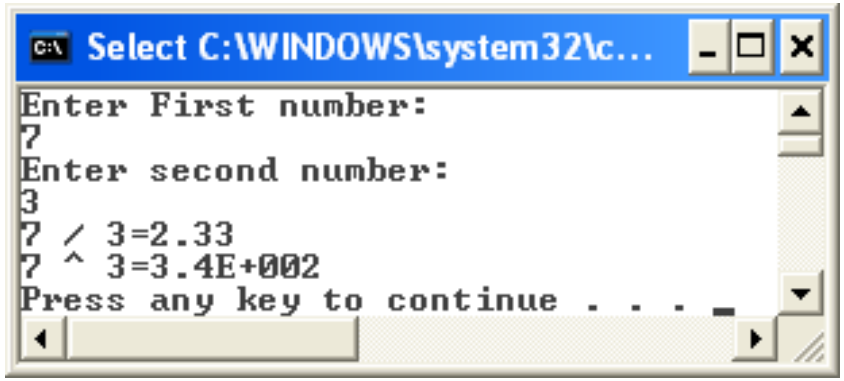
    e.g., x = int.Parse(Console.ReadLine());

```
1  // Fig. 3.14: Addition.cs
2  // Displaying the sum of two numbers input from the keyboard.
3  using System;
4
5  class Addition
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       int number1; // declare first number to add
11       int number2; // declare second number to add
12       int sum; // declare sum of number1 and number2
13
14       Console.Write("Enter first integer: "); // prompt user
15       // read first number from user
16       number1 = int.Parse(Console.ReadLine());
18       Console.Write("Enter second integer: "); // prompt user
19       // read second number from user
20       number2 = int.Parse(Console.ReadLine());
21
22       sum = number1 + number2; // add numbers
23
24       Console.WriteLine($"Sum is {sum}"); // display sum
25    } // end Main
26 } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Example

```
using System;
namespace ConsoleApplication1 {
  class myProgram {
   static void Main(string[] args){
        int x, y;
        double r;
        //Display Message on Screen
        Console.WriteLine("Enter First number:");
        //Read Input from keyboard
        x = int.Parse(Console.ReadLine());
        Console.WriteLine("Enter second number:");
        y = int.Parse(Console.ReadLine());
        r = (double)x / y;
        //Display with format
        Console.WriteLine( x+ " / " + y +"=" + r.ToString("0.00"));
        r = Math.Pow(x, y);
        Console.Out.WriteLine(x + " ^ " + y + "=" + r.ToString("E02"));
     }
   }
}
```



Predefined Functions (Math library)

- Math Class is part of .Net Framework
- Examples: double n;

```
n = Math.Sqrt(81);                  // n= 9.0
n = Math.Round(42.889) ;            //  n = 42.0
n = Math.Round(42.889,1) ;          //  n = 42.9
n = Math.Ceil(42.889) ;             //  n = 43
n = Math.Floor(42.889) ;            //  n = 42
n = Math.Round(42.889,1) ;          //  n = 42.9
n = Math.Abs(-10);                  //  n = 10.0
n = Math.Pow(2,3) ;                 //  n = 8.0
n = Math.PI;                        // n = 3.14 ….
n = Math.Log(2.712828)              //n=1.0
n = Math.Max(3,7)                   //n=7.0
n = Math.Min(3,7)                   //n=3.0
```

Exercise

1. **Write a C# program that reads an angle in degrees and convert it to radius using the following formula**

   Radius = degree * PI / 180

   - **Compute the sin, cos and tan for the angle.**

2. **Write a C# program that reads two numbers and compute**
   - **Maximum value, Minimum and average**

Enumerations

- enum is a custom data type of name-value pairs.
- An enumerator is a class or structure that implements a .NET interface named IEnumerable.
- enum is widely used in .Net Framework
  - Commonly used to make code more readable, maintainable, and type-safe compared to using raw numeric or string values.
- Examples:

```
// Declare the enumeration:
public enum MessageSize { Small , Medium , Large }
// Equivalent to
public enum MessageSize { Small = 0, Medium = 1, Large = 2}
// Create a field or property:
MessageSize msgSize;
// Assign a value:
msgSize = MessageSize.Medium;
```

Selection Statements

- Logical Operators and expressions
  - same as in C++ and Java
- If/else
  - same as in C++ and Java
- Switch
  - same as in C++ and Java

Nested if...else Statements

```
if (studentGrade >= 90)
{
   Console.WriteLine("A");
}
else if (studentGrade >= 80)
{
   Console.WriteLine("B");
}
else if (studentGrade >= 70)
{
   Console.WriteLine("C");
}
else if (studentGrade >= 60)
{
   Console.WriteLine("D");
}
else
{
   Console.WriteLine("F");
}
```

Example – Nested If

```
if (a == 1)
    Label1.Text = "one";
else if (a >= 2)
    Label1.Text = "Two or More";
else
    Label1.Text = "Less than One";
```

Exercise

- Write a program that read an integer number between 10 and 500 and perform the following
  - Display if it is less than or equal to 250
  - Displays if it is even or odd
  - Display if it divisible by 5 or not
- If the number is less than 10 or more than 500 the program should display an either message.

Exercise

- Write a C# program that the values for a,b & c and computes the quadratic equation using the following formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Let root= $B^2 - 4\ a\ c$
- If root > Zero then there are two solutions
- If root = ZERO then there is only solution
- If root < ZERO then there is no solution.

Case Statement

- Can be used to evaluate a single value or expression with type int, bool, char, string OR enumeration
- string color = "blue";

```
switch (color) {
    case "red":
        Label1.Text = "Red";
        break;
    case "blue":
        Label1.Text = "Blue";
        break;
    default:
        Label1.Text = "Not Red Nor Blue";
        break;
}
```

Case Statement

- `enum MessageSize {Small, Medium, Large};`
- `MessageSize msgSize = MessageSize.Small;`
- ```
  switch (msgSize) {
      case MessageSize.Small: Label1.Text = "S";
      break;
      case MessageSize.Medium: Label1.Text = "M";
      break;
      case MessageSize.Large: Label1.Text = "L";
      break;
      default: Label1.Text = "---";
      break;
  }
  ```
  ```

Example - using enum

```
enum EmpType
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 9
}
```

```
class Program
{
   static void Main(string[] args){
    // Make an EmpType variable.
    EmpType emp = EmpType.Contractor;
     switch (emp) //Check employee type
     {
          case EmpType.Manager:              Console.WriteLine("Manager");break;
          case EmpType.Grunt:      Console.WriteLine("Grunt"); break;
          case EmpType.Contractor:           Console.WriteLine("Contractor");break;
          case EmpType.VicePresident:        Console.WriteLine("VP"); break;
          default:                           Console.WriteLine("Invalid type");
     }
}
```

Repetitions

- Same as in Java
  - For Loop
  - while loop
  - do-while
- foreach

For Loop

```
for (<declation and initialization>;<condition>;<increment>)
{
    statement1;
    statement2; ……
}
```

- Display numbers from 1 to 10

```
for(int i=1; i<= 10;i++)
    Console.WriteLine(i)
```

- Display numbers from 10 to 1

```
for(int i=10; i>= 1;i--)
    Console.WriteLine(i)
```

- Display even numbers from 0 to 10

```
for(int i=0; i <= 10;i+=2)
    Console.WriteLine(i)
```

For Loop

- Write C# Program that reads 10 numbers and compute their summation

```
class Program {
   static void Main(string[] args){
        int total = 0;
        int num;
        for(int i = 0; i < 5; i++)
        {
                Console.WriteLine("Enter number");
                num = int.Parse(Console.ReadLine());
                total = total + num;
        }
        Console.WriteLine($"The sum is {total}");
  }
}
```

Reading n numbers

```
using System;

namespace ConsoleApp2
{
   internal class Program    {
      static void Main(string[] args) {

         int num,n, total=0;
         try          {
            Console.WriteLine("Enter n ");
            n = int.Parse(Console.ReadLine());
            int[] nums = new int[n];
            for (int i = 0; i < n; i++)              {
               Console.WriteLine($"Enter num {i+1} :  ");
               nums[i] = int.Parse(Console.ReadLine());
               total += nums[i];
            }

            Console.WriteLine($"n is {nums.Length} and the total is {total}");

         }
         catch(Exception e) { Console.WriteLine($"Error is {e.ToString()}"); }
      }
   }
}
```

while Loop

```
<declation and initialization>;
while (<condition>)
{
    statement1;
    statement2; ……
    <increment>;
}
```

- Display numbers from 1 to 10

```
int i=1;
while(i<=10)
{
    Console.WriteLine(i)
    i++;
}
```

- Display numbers from 1 to 10

```
int i=10;
while(i>=1){
    Console.WriteLine(i)
    i--;
}
```

- Display even numbers from 0 to 10

```
int i=0;
while(i<=10)
{
    Console.WriteLine(i)
    i=i+2;
}
```

while Loop

- Write C# Program that reads 10 numbers and compute their summation

```
class Program {
   static void Main(string[] args){
        int total = 0,num;
        int i = 0;
        for(i < 5){
                Console.WriteLine("Enter number");
                num = int.Parse(Console.ReadLine());
                total = total + num;
                 i++;
        }
        Console.WriteLine($"The sum is {total}");
  }
}
```

Do-while Loop

```
<declation and initialization>;
  do
  {
       statement1;
       statement2; ……
       <increment>;
  }
  while (<condition>);
```

- Display numbers from 1 to 10

```
 int i=1;
 do
 {
      Console.WriteLine(i)
      i++;
 }
 while(i<=10);
```

- Display numbers from 10 to 1

```
int i=10;
 do{
      Console.WriteLine(i)
      i--;
 }
 while(i>=1);
```

- Display even numbers from 0 to 10

```
int i=0;
 do
 {
      Console.WriteLine(i)
      i=i+2;
 }
 while(i<=10);
```

Do-while Loop

- Write C# Program that reads 10 numbers and compute their summation

```
class Program {
    static void Main(string[] args){
          int total = 0,num;
          int i = 0;
          do {
                  Console.WriteLine("Enter number");
                  num = int.Parse(Console.ReadLine());
                  total = total + num;
                   i++;
          }
          whie(i < 5);
          Console.WriteLine($"The sum is {total}");
   }
}
```

Exercise

- Write a C# program that reads the number of students (n) and reads read the result for each students as a string “Pass” or “Fail”

- The program needs to perform the following
  - Computes the number of students who passed the exam
  - Computes the number of failed students
  - Compute the maximum of the numbers
  - Compute the lowest values

Arrays

- 1D arrays (starts from index 0)
  - `string[] course = new string[3];`
    `course[0] = "Web";`
    `course[1] = "Server";`
    `course[2] = "Programing";`
  - `int[] arr = new int[5]{1, 2, 3, 4, 5};`
  - `int[] arr = {1, 2, 3, 4, 5};`
  - `char[] delimeters = new char[]{' ', '\r', '\n'};`
  - Array size
    - `Size = arr.Length; //Size = 5;`
- 2D arrays
  - `int[,] mat = new int[2, 3]; //All elements set to ZERO`
    `mat[1,2] = 5;`
  - `//2 by 3 Matrix`
    `int[,] Matrix = {{1,2,3},{4,5,6}};`

Manipulating Arrays

- Array Declaraction

```
int[] arr ={5, 7, 9, 10, 3};
```

- Using while loop

```
int i=0;
while (i< arr.Length){
      Console.WriteLine(arr[i])
      i++;
}
```

- Using for loop

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

Foreach Loop

- Similar to for(... in …) in JavaScript
- Allows you to loop through the items in a set of data
- Display element of an array

```
int[] arr ={5, 7, 9, 10, 3};
foreach (int number in arr)
{
        Console.WriteLine(number);
 }
```

Exercise

- Read unknown numbers of numbers from the keyboard (separated by space) and compute their summation
  - Using a string to read all number

```
string line = Console.ReadLine();
```

  - Split the numbers as separate array of string

```
char [] sep = { ' ', ','};
string[] snum = line.Split(sep);
```

  - Create an array of integers (or double) with the same size

```
int[] num = new int[snum.Length];
```

  - Use loop to assign numbers to array

```
for (int i = 0; i < snum.Length; i++)
                num[i] = int.Parse(snum[i]);
```

Creating a Computed Size Array

- The array size does not need to be a compile-time constant
  - Any valid integer expression will work
  - Array size specified by compile-time integer constant:

```
long[ ] row = new long[4];
```

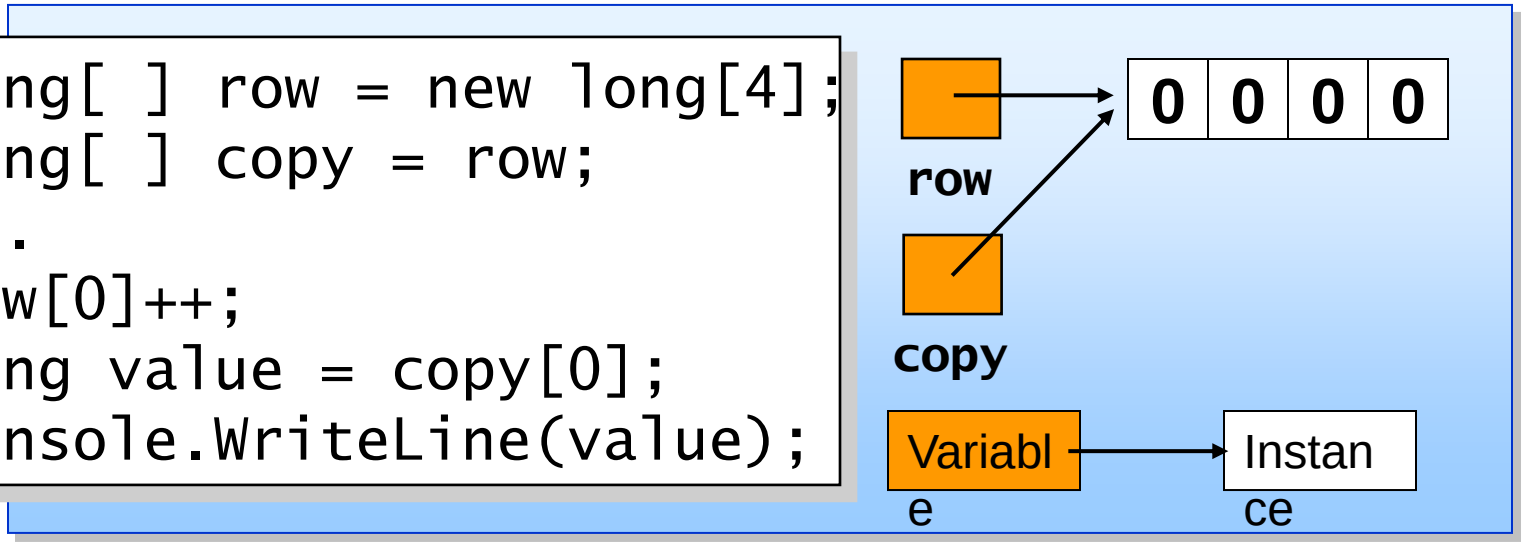
  - Array size specified by run-time integer value:

```
string s = Console.ReadLine();
int size = int.Parse(s);
long[ ] row = new long[size];
```

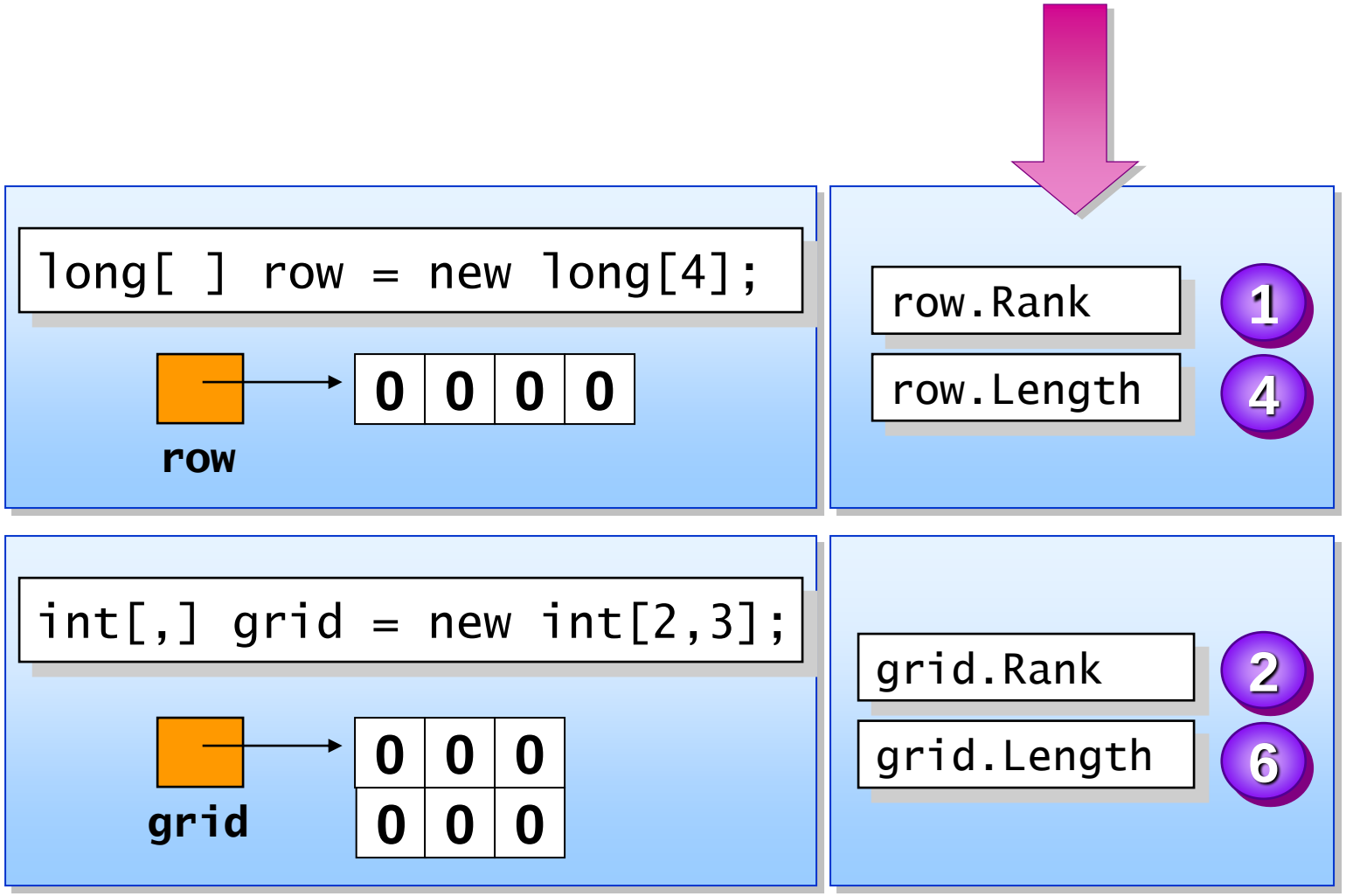
Copying Array Variables

- Copying an array variable copies the array address only
  - It does not copy the array instance
  - Two array variables can refer to the same array instance

```
long[ ] row = new long[4];
long[ ] copy = row;
...
row[0]++;
long value = copy[0];
Console.WriteLine(value);
```

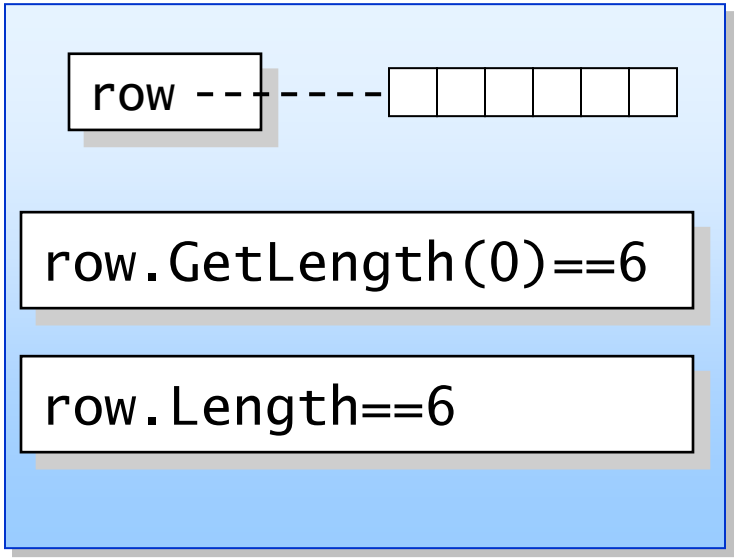


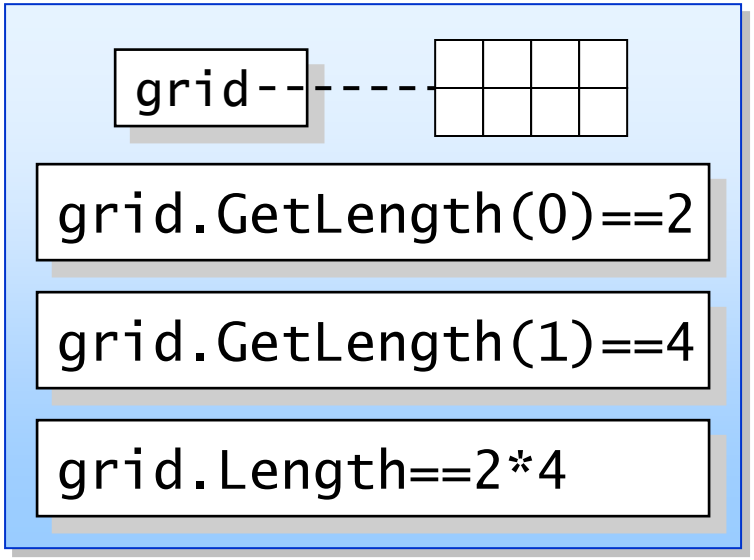
Array Properties



Checking Array Bounds

- All array access attempts are bounds checked
  - A bad index throws an IndexOutOfRangeException
  - Use the Length property and the GetLength method





Dynamic Arrays

- C# arrays do not support re-dimensioning.
  - This means that once you create an array, you can't change its size.
  - Instead, you would need to create a new array with the new size and copy values from the old array to the new, which would be a tedious process.
- However, if you need a dynamic array-like list, you can use one of the collection classes provided to all .NET languages through the .NET class library.
- One of the simplest collection classes that .NET offers is the ArrayList, which always allows dynamic resizing.

ArrayList

- **Using an ArrayList in C#**

```
// Create an ArrayList object. It's a collection, not an array,
// so the syntax is slightly different.
ArrayList dynamicList = new ArrayList();
// Add several strings to the list.
// The ArrayList is not strongly typed, so you can add any data type
// although it's simplest if you store just one type of object
// in any given collection.
dynamicList.Add("one");
dynamicList.Add("two");
dynamicList.Add("three");
// Retrieve the first string. Notice that the object must be converted to a
// string, because there's no way for .NET to be certain what it is.
string item = (string) dynamicList[0];
// OR
string item = Convert.ToString(dynamicList[0]);
```

List Collection

- **List Class**
  - **Used to maintain data for a specific type**

```
List<type/class> list1 = new List<type/class>() [{…..}];
                                                    //optional
```

  - **Example:**

```
List<string> list1 = new List<string>() {"A","B","C"};
List1.Add("X");
List1.Remove("A");
if(list1.contains("B")) …….;
For(int i=0;i<list1.Count;i++) Console.WriteLine(list1[i]);
Foreach(string x in list1) Console.WriteLine(x);

List<int> Intlist = new List<int>();
IntList.Add(5);
```

Exercise

- Write a C# program that uses a dynamic list for friend names.

- The program should repeatedly display the following menu and perform the corresponding actions
  - Add a new friend
  - Remove a friend by name
  - Remove the i^{th} friend
  - Print the names of all friend
  - Print the names in alphabetical order
  - Stop the program

Program

```
using System;
using System.Collections.Generic;
namespace ConsoleApp1{
  internal class Program  {
    static void Main(string[] args)  {
     List<String> friends = new List<String>() ;
      int ch = 0;
      while ((ch=menu())!=4)
      {
        switch (ch) {
          case 1:
            Console.WriteLine("Enter name:");
            String f=Console.ReadLine();
            friends.Add(f);
            break;
          case 2:
            Console.WriteLine("Enter index");
            int ind=int.Parse(Console.ReadLine());
            friends.RemoveAt(ind);
            break;
          case 3:
            foreach(String  s in friends)
              Console.WriteLine(s);
            break;
        }
      }
    }
```

```
 static int menu()
     {
       int ch = 0;
       do
       {
         Console.WriteLine("1-Add friend 2-Remove friend 3-
display 4-Stop.");
         try
         {
           ch = int.Parse(Console.ReadLine());
         }
         catch { Console.WriteLine("Invalid choose"); }
       }
       while (ch>0 &&ch>5);
       return ch;
     }
   }
}
```

Date and Time

- **DateTime Data Type**
  - **Initialize to the current date and time**

    ```
    DateTime myDate1 = DateTime.Now;
    DateTime myDate 2= DateTime.Now.AddHours(10);
    ```

  - **Members to get current date and time**
    - **Year, Date, Day, Hour, Minute, Seconds ….**
  - **Methods to set date and time**
    - **myDate1.AddDay(1); myDate1.AddHours(2);**
- **TimeSpan**
  - **Represents a time intervals**
    - **TimeSpan timeDiff = myDate2 – myDate1;**

Example

```
using System;
namespace ConsoleApp1
{
   internal class Program
   {
      static void Main(string[] args) {
         DateTime t1 = DateTime.Now;
         Console.WriteLine("Enter your name");
         string name=Console.ReadLine();

         DateTime t2 = DateTime.Now;
         TimeSpan dif = t2 - t1;
         Console.WriteLine($"Dear {name}, you took {dif.TotalSeconds} "+
                  " seconds to enter your name..");

      }
   }
}
```

Exercise

- Write a C# program that requests from the user to enter his/her birthdate as three separate integers using the following format

  dd mm yyyy

- The program needs to compute the age and the day of the week for the birth day.

Defining Methods

- **Main is a method**
  - **Use the same syntax for defining your own methods**

```
using System;

class ExampleClass
{
    static void ExampleMethod( )
    {
        Console.WriteLine("Example method");
    }
    static void Main(…… )
    {
        // ...
       ExampleMethod( );
    }
}
```

Static Class Members

- You can use static methods without creating an object
- E.g.

```
DateTime myDate = DateTime.Now;
myDate = myDate.AddDays(1);
```

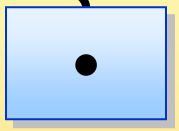
Example – Defining methods

```
1  // Fig. 7.10: SquareTest.cs
2  // Square method used to demonstrate the method
3  // call stack and activation records.
4  using System;
5
6  class Program
7  {
8     static void Main()
9     {
10       int x = 10; // value to square (local variable in main)
11       Console.WriteLine($"x squared: {Square(x)}");
12    }
13
14    // returns the square of an integer
15    static int Square(int y) // y is a local variable
16    {
17       return y * y; // calculate square of y and return result
18    }
19 }
```

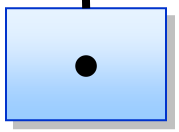
```
x squared: 100
```

Using References as Method Parameters

- References can be used as parameters
  - When passed by value, data being referenced may be changed



```
static void Swap(int x, int y)
{
  int temp=x; x=y; y=temp;
}
```



```
int a=5,b=7;
Swap(a,b);
Console.WriteLine("a=" + a+" , b=" + b);
```

Passing Parameters By Reference

- Methods access the parameter values directly.
  - If a method changes the value of a pass-by-reference parameter, the original object is also modified
- Use ref keywords in both function call and definition
- Example
  - Function definition

```
static void swap(ref int x, ref int y)
{ int temp = x; x=y; y=temp; }
```

  - Function call

```
int  a = 7, b = 3;
swap(ref a, ref b); // Function call in the main method
Console.WriteLine("  a = "+ a + " and b= " + b);
```

Output Parameter

- Commonly used as a way to return multiple pieces of information from a single method.
- Precede the parameter declaration by the keyword **out**
- Call code can submit uninitialized variables
- Example

```
private  void ProcessNumber(int n, out int nSquare, out int nCube)
{ nSquare = n*n;   nCube = n* n * n; }

int num = 5,myC,myS;
ProcessNumber(num, out myS, out myC);
```

Example - ref and out parameters

```
using System;
namespace ConsoleApp1 {
   internal class Program    {
      static void Main(string[] args) {
        int a=10,b=20;
        Console.WriteLine($"a={a} and b={b}");
         swap(ref a,ref b);
         Console.WriteLine($"a={a} and b={b}");
         int m1, m2;
         double av;
         maxMinAvg(a, b, out m1, out m2, out av);
         Console.WriteLine($"min is {m1}, max={m2} and avg={av}");
      }
      static void swap(ref int x, ref int y) {
         int t = x; x=y; y=t;
         Console.WriteLine($"x={x} and y={y}");
      }
      static void maxMinAvg(int x, int y,out int min,out int max,out double avg) {
         min = Math.Min(x,y);
         max= Math.Max(x,y);
         avg = (x + y) / 2;

      }
   }
}
```

Optional Parameters

- 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.

Optional Parameters

- For example, the method header

```
static int Power(int baseValue, int exponentValue = 2)
```

- Optionally, a second argument (for the `exponentValue` parameter) can be passed to `Power`.
- Consider the following calls to Power:
  - `Power()`
  - `Power(10)`
  - `Power(10, 2)`
- The first generates a compilation error because this method requires a minimum of one argument.
- The second is valid because one argument (`10`) is being passed—the optional `exponentValue` is not specified in the method call.
- The last call is also `valid`—`10` is passed as the required argument and 2 is passed as the optional argument.

```
1  // Fig. 7.15: CalculatePowers.cs
2  // Optional parameter demonstration with method Power.
3  using System;
4
5  class CalculatePowers
6  {
7     // call Power with and without optional arguments
8     static void Main()
9     {
10       Console.WriteLine($"Power(10) = {Power(10)}") ;
11       Console.WriteLine($"Power(2, 10) = {Power(2, 10)}");
12    }
13
14    // use iteration to calculate power
15    static int Power(int baseValue, int exponentValue = 2)
16    {
17       int result = 1;
18
19       for (int i = 1; i <= exponentValue; ++i)
20       {
21          result *= baseValue;
22       }
23
24       return result;
25    }
26 }
```

```
Power(10) = 100
Power(2, 10) = 1024
```

Named Parameters

- C# provides a feature called named parameters, which enable you to call methods that receive optional parameters by providing only the optional arguments you wish to specify.
- Explicitly specify the parameter's name and value—separated by a colon (:)—in the argument list of the method call.
- For example:

```
// sets the time to 12:00:22
t.SetTime( hour: 12, second: 22 );
```

Method Overloading

- Define a set of identically named methods that differ by the number (or type) of parameters, the method in question is said to be overloaded.

```
class Program {
  static void Main(string[] args){
    int sum=Add(2,3);
}
// Overloaded Add() method.
static int Add(int x, int y)
{ return x + y; }

static double Add(double x, double y)
{ return x + y; }

static long Add(long x, long y)
{ return x + y; }
}
```

Local Functions (New)

- Another new feature introduced in C# 7 is the ability to create methods within methods, referred to officially as *local functions*.
- A local function is a function declared inside another function.

```
static int AddWrapper(int x, int y)
{
    //Do some validation here
    return Add();
    int Add()
   {
     return x + y;
   }
}
```

Exceptions

- Same as C++ using try…catch…finally

```
try{
   //statements
}
catch (Exception ee) {
   //handling exceptions
}
finally {
  //clean used memory and close connections
}
```

Example - Exceptions

```
//read values for  n2 and n2
try {
    n1 = int.Parse(Console.ReadLine());
    n2 = int.Parse(Console.ReadLine());
    result = n1 / n2;
}
catch (DivideByZeroException err) {
   Console.Writeline("Invalid operation: division by zero" );
}
catch (Exception ee) {
  Console.WriteLine("Other problems: " + ee.Message);
}
```

Classes

- **Simple Class**

```
public class className
{
  //Class code
      //Attributes
      // Properties
     // and methods
}
```

Example

- **Class definition**

```
public class Product{
  private string name;
  private decimal price;
  private string imageURL;
}
```

- **Creating Objects**

```
Product  p1 = new Product( );
```

- **Release the object from memory**

```
p1 = null;
```

Class Interactions

- Class can interact with each other using
  - **Properties:** allow to access and/or modify object data (attributes)
    - Similar to set and get methods in C++ classes
  - **Methods:** allow to perform action on an object
    - Similar methods in C++
  - **Events:** provide notification that something has happened (e.g. button is clicked)

Adding Class Properties

- Used to set and get the value for a given attribute
- Property name is the same name as the corresponding attribute with first letter in capital
- Two Accessors (properties can have both or one accessor)
  - get accessor is used to the value for an attribute
  - set accessor is used to set the value for a given attribute

```
public class Product
{
  private string name;
  private decimal price;
  private string imgUrl;
  public string Name {
    get   { return name;}
    set    { name = value;}
}
```

```
public decimal Price {
    get   { return price;}
    set    { price= value;}
}
public string ImgUrl  {
    get   { return imgUrl;}
    set    {imgUrl = value;}
}
}
```

Using Properties

- Product P1 = new Product( );
- P1.Name= “HP Printer”;
- P1.Price = 200;
- P1.ImgUrl = “http://www.hp.com/printers”;

- int ProdPrice = P1.Price;

Auto-implemented properties

- In some cases, property get and set accessors just assign a value to or retrieve a value without including any additional logic.
- An auto-implemented property by using the get and set keywords without providing any implementation.
- Auto-implemented properties are used to simplify the C# code.

```
public class Product
{
        public string Name   { get; set; }
        public decimal Price    {  get; set;}
        public string ImgUrl    {  get; set;}
}
```

Adding Constructors (similar to C++)

```
public class Product{
        private string name;
        private decimal price;
        private string imgUrl;
        public Product(string myN, decimal myP)
       {
           name= MyN;  price= myP
       }
}
```

- Create object using constructor

```
Product  p2 = new Product("Desktop",377.55M);
```

Exercise

- Define a class to main employee name and his/her salary
- Uses a dynamic list for employee information.
- The program should repeatedly display the following menu and perform the corresponding actions
  - Add a new employee
  - Remove the i^{th} friend
  - Print information about all employees
  - Print the salary for a given employee name
  - Stop the program

Namespaces

- Namespaces are used to organize different types in class library.

```
namespace MyCompany{
    namespace MyApp{
        public class Product
        {
            ....................
        }
    }
}
```

```
namespace MyCompany.MyApp
{
        public class Product
        {
            ....................
        }
}
```

- Using namespaces

```
using namespace MyComany.MyApp;
Product p = new Product();
```

- OR MyComany.MyApp.Product p = new MyComany.MyApp.Product();

Reading and Writing files

- Import input/output file using
  - `using System.IO;`
- Input files – StreamReader class
- Output files – StreamWriter class
- Use exceptions to handle problems in opening or creating files

Example – Writing to output file

- Writing a message to an output file

```
try{
    StreamWriter r = new StreamWriter(“message.txt");

    r.WriteLine("This is an output file.....");

    r.Close();
}
catch (Exception ee){
    Console.WriteLine("Problem in writing to file");
}
```

Example – Reading from input file

- Reading two numbers in each line from a text file

```
int x, y;
string s;
string[] w;
try{
 StreamReader fin = new StreamReader(@"C:\....\ConsoleApplication2\TextFile1.txt");
 while ((s=fin.ReadLine())!=null){
          w=s.Split(' ');
          x=int.Parse(w[0]);      y=int.Parse(w[1]);
          Console.WriteLine(x+"   "+y);
 }
 fin.Close();
}
catch (Exception ee){  //other exceptions e.g. FileNotFoundException
   Console.WriteLine("Problem in reading from file");
}
```

Accessing all folders/files

- Find all directories in a given path
  - `string[] dirs = Directory.GetDirectories(path);`
  - Use loop (foreach) to process each folder
- Get the current working directory
  - `String curDir=Directory.GetCurrentDirectory()`

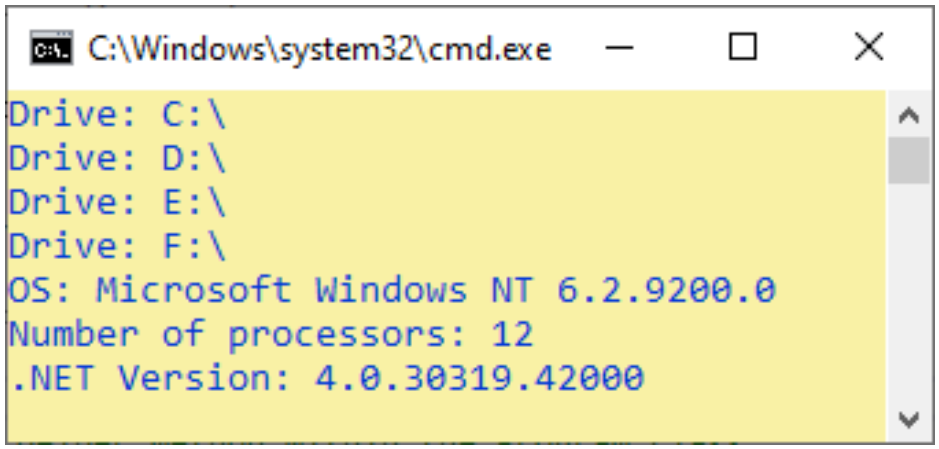
  OR
  - `string curDir= Environment.CurrentDirectory;`
- Set the current working directory
  - `Directory.SetCurrentDirectory(path);`
- Create new folder
  - `Directory.CreateDirectory(DirectoryName);`

Extracting Server Information

- `Environment.CurrentDirectory`
- `Environment.MachineName`
- `Environment.UserDomainName`
- `Environment.UserName`
- `Environment.OSVersion`
- `Directory.GetCurrentDirectory()`
- `Environment.GetLogicalDrives()`

System.Environment Class

```
static int Main(string[] args) {
  // Helper method within the Program class.
  ShowEnvironmentDetails();
  Console.ReadLine();
  return -1;
}
static void ShowEnvironmentDetails() {
  // Print drives on this machine, and other interesting details.
  foreach (string drive in Environment.GetLogicalDrives())
            Console.WriteLine("Drive: {0}", drive);
  Console.WriteLine("OS: {0}", Environment.OSVersion);
  Console.WriteLine("Number of processors: {0}", Environment.ProcessorCount);
  Console.WriteLine(".NET Version: {0}",  Environment.Version);
}
```



Accessing all files

- Find all files in a given path
  - `string[] f = Directory.GetFiles(path);`

- Find all files with a specific name/extension "regular expression"
  - `string[] f = Directory.GetFiles(path, "s*.pdf");`

Exercise – File contents

- List all files and folders in your computer using a specific path

- Display the contents for all files

Downloading Web Contents

- Import Net System.Net
- Download Web page content

```
WebClient c = new WebClient();
string web = c.DownloadString(url);
```

- Extract Web page contents
- Web Content Mining: Extract Web page URLs
- Web Crawling: Recursively extract contents for URL pages

C# Collections and Generics

Collections

- **Two Types of Collections**
  - **System.Collections (non-generic collections)**
    - **to store and interact with bits of data used within an application.**
  - ***Generic Collections***
    - **New namespace was introduced in the base class libraries: System.Collections.Generic.**
- **Generic containers are often favored over their non-generics**
  - **typically provide greater type safety and performance benefits.**

Types of System.Collections

System.Collections Class	Meaning in Life
ArrayList	Represents a dynamically sized collection of objects listed in sequential order
BitArray	Manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0)
Hashtable	Represents a collection of key-value pairs that are organized based on the hash code of the key
Queue	Represents a standard first-in, first-out (FIFO) collection of objects
SortedList	Represents a collection of key-value pairs that are sorted by the keys and are accessible by key and by index
Stack	A last-in, first-out (LIFO) stack providing push and pop (and peek) functionality

Generic Collections

- Specify Type Parameters for Generic Classes/Structures

```
// This List<> can hold only Person objects.
List<Person> morePeople = new List<Person>();
morePeople.Add(new Person ("Frank", "Black", 50));
Console.WriteLine(morePeople[0]);

// This List<> can hold only integers.
List<int> moreInts = new List<int>();
moreInts.Add(10);
moreInts.Add(2);
int sum = moreInts[0] + moreInts[1];
```

- Advantages:
  - Provide better performance
    - No need for casting: no boxing or unboxing penalties when storing value types.
  - Type safe - can contain only the type of type you specify.

Generic Collections

Generic Class	Meaning in Life
Dictionary<TKey, TValue>	This represents a generic collection of keys and values.
LinkedList<T>	This represents a doubly linked list.
List<T>	This is a dynamically resizable sequential list of items.
Queue<T>	This is a generic implementation of a first-in, first-out list.
SortedDictionary<TKey, TValue>	This is a generic implementation of a sorted set of key-value pairs.
SortedSet<T>	This represents a collection of objects that is maintained in sorted order with no duplication.
Stack<T>	This is a generic implementation of a last-in, first-out list.

List<T> Collection

- **List Class**
  - **Used to maintain data for a specific type**

```
List<type/class> list1 = new List<type/class>() [{…..}];
                                                    //optional
```

  - **Example:**

```
List<string> list1 = new List<string>() {"A","B","C"};

List1.Add("X");
List1.Remove("A");
if(list1.contains("B")) …….

List<int> Intlist = new List<int>();
IntList.Add(5);
```

Example – Define Person Class

```
class Person
    {
        public string FirstName {get;set;}
        public string LastName { get; set; }
        public int Age { get; set; }

        public Person(string f, string l, int a)
        {
            FirstName = f; LastName = l; Age = a;
        }
        public override string ToString()
        {
            return "Name: " + FirstName + " " + LastName + " and Age:" + Age;
        }
    }
```

Example – List<T>

* Main Operations: Add(), Insert(), Remove(), RemoveAt(),…

```
static void Main(string[] args){
     // Make a List of Person objects
     List<Person> people = new List<Person>() {
                           new Person ("Ali", "Alabri", 47),
                           new Person ("Amal", "Aljabri", 45),
                           new Person ("Sultan", "Alrashdi", 9),
                           new Person ("Khamis", "Alsalmi", 8)};
     // Print out # of items in List.
     Console.WriteLine("Items in list: {0}", people.Count);
     // Enumerate over list.
     foreach (Person p in people)  Console.WriteLine(p.ToString());
     // Insert a new person.
     Console.WriteLine("\n->Inserting new person.");
     people.Insert(2, new Person ( "Mouza", "Alazri", 2 ));
     Console.WriteLine("Items in list: {0}", people.Count);
     // Copy data into a new array.
     Person[] arrayOfPeople = people.ToArray();
     foreach (Person p in arrayOfPeople)
                Console.WriteLine("First Names: {0}", p.FirstName);
 }
```

```
Items in list: 4
Name: Ali Alabri and Age:47
Name: Amal Aljabri and Age:45
Name: Sultan Alrashdi and Age:9
Name: Khamis Alsalmi and Age:8

->Inserting new person.
Items in list: 5
First Names: Ali
First Names: Amal
First Names: Mouza
First Names: Sultan
First Names: Khamis
Press any key to continue . . .
```

Stack<T>

- Represents a collection that maintains items using a last-in, first-out manner.
- Main Operations: Push(), Pop(), Peek()

```
static void Main(string[] args){
   Stack<Person> stackOfPeople = new Stack<Person>();
   stackOfPeople.Push(new Person ("Ali","Tabook", 47 ));
   stackOfPeople.Push(new Person ("Sultan","Altaei", 45 ));
   stackOfPeople.Push(new Person ("Maryam", "albalushi", 9 ));
   // Now look at the top item, pop it, and look again.
   Console.WriteLine("First person is: {0}", stackOfPeople.Peek());
   Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
   Console.WriteLine("\nFirst person is: {0}", stackOfPeople.Peek());
   Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
   Console.WriteLine("\nFirst person item is: {0}", stackOfPeople.Peek());
   Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
   try {
     Console.WriteLine("\nnFirst person is: {0}", stackOfPeople.Peek());
     Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
   }
   catch (InvalidOperationException ex) {
     Console.WriteLine("\nError! {0}", ex.Message);
  }
}
```

```
First person is: Name: Maryam albalushi and Age:9
Popped off Name: Maryam albalushi and Age:9

First person is: Name: Sultan Altaei and Age:45
Popped off Name: Sultan Altaei and Age:45

First person item is: Name: Ali Tabook and Age:47
Popped off Name: Ali Tabook and Age:47

Error! Stack empty.
```

Queue<T>

- Queues are containers that ensure items are accessed in a first-in, first-out manner.
- Main Operations: Enqueue(), Dequeue(), Peek()

```
static void Main(string[] args){
        // Make a Q with three people.
        Queue<Person> peopleQ = new Queue<Person>();
        peopleQ.Enqueue(new Person("Ali", "Tabook", 47));
        peopleQ.Enqueue(new Person("Sultan", "Altaei", 45));
        peopleQ.Enqueue(new Person("Maryam", "albalushi", 9));
        // Peek at first person in Q.
        Console.WriteLine("{0} is first in line!", peopleQ.Peek().FirstName);
        // Remove each person from Q.
        GetCoffee(peopleQ.Dequeue());
        GetCoffee(peopleQ.Dequeue());
        GetCoffee(peopleQ.Dequeue());
        // Try to de-Q again?
        try { GetCoffee(peopleQ.Dequeue()); }
        catch (InvalidOperationException e){ Console.WriteLine("Error! {0}", e.Message); }
}
static void GetCoffee(Person p){
        Console.WriteLine("{0} got coffee!", p.FirstName);
}
```

```
Ali is first in line!
Ali got coffee!
Sultan got coffee!
Maryam got coffee!
Error! Queue empty.
```

SortedSet<T>

- Automatically ensures that the items in the set are sorted when you insert or remove items.
- Pass as a constructor argument an object that implements the generic IComparer<T> interface to compare items.

```
class Program {
     // You must import System.Collections to access the ArrayList.
     static void Main(string[] args) {
        SortedSet<Person> setOfPeople = new SortedSet<Person>(new SortPeopleByAge())
              {  new Person ("Ali", "Tabook", 47), new Person ("Sultan", "Altaei", 45),
                 new Person ("Maryam", "albalushi", 9),new Person ("Khamis", "Alsalmi", 8) };
         // Note the items are sorted by age!
         foreach (Person p in setOfPeople) Console.WriteLine(p.ToString());
         Console.WriteLine();
         // Add a few new people, with various ages.
         setOfPeople.Add(new Person("Salem", "Juma", 8));
         setOfPeople.Add(new Person("Malek", "Juma", 8));
         // Still sorted by age!
         foreach (Person p in setOfPeople) Console.WriteLine(p.ToString());
     }  }
  //create comparer
  internal class SortPeopleByAge: IComparer<Person> {
     public int Compare(Person x, Person y) {
         //first by Age then Last name then First Name
         int result = x.Age.CompareTo(y.Age);
         if (result == 0) result = x.LastName.CompareTo(y.LastName);
         if (result == 0) result = x.FirstName.CompareTo(y.FirstName);
         return result;
     }  }
```

```
Name: Khamis Alsalmi and Age:8
Name: Maryam albalushi and Age:9
Name: Sultan Altaei and Age:45
Name: Ali Tabook and Age:47

Name: Khamis Alsalmi and Age:8
Name: Malek Juma and Age:8
Name: Salem Juma and Age:8
Name: Maryam albalushi and Age:9
Name: Sultan Altaei and Age:45
Name: Ali Tabook and Age:47
```

Dictionary<TKey, TValue>

- Rather than obtaining an item from a List<T> using a numerical identifier (integer index), Dictionary<Tkey,Value> uses the unique text key as subscript.

```
static void Main(string[] args) {
        // Populate using Add() method
        Dictionary<string, Person> peopleA = new Dictionary<string, Person>();
        peopleA.Add("P1", new Person ("Ali", "Tabook", 47));
        peopleA.Add("P2", new Person ("Sultan", "Altaei", 45));
        peopleA.Add("P3", new Person("Maryam", "albalushi", 9));
        peopleA.Add("P4", new Person ("Khamis", "Alsalmi", 8));
        // Get Homer.
        Person p = peopleA["P2"];
        Console.WriteLine(p.ToString());
        Console.WriteLine();

        // Populate with initialization syntax.
        Dictionary<string, Person> peopleB = new Dictionary<string, Person>()
        {
        { "A1", new Person ("Ali", "Tabook", 47) },
        { "A2", new Person ("Sultan", "Altaei", 45) },
        { "A3", new Person("Maryam", "albalushi", 9)}
        };
        // Get Lisa.
        Person p2 = peopleB["A1"];
        Console.WriteLine(p2.ToString());
}
```

Name: Sultan Altaei and Age:45

Name: Ali Tabook and Age:47

Custom Generic Methods

- Function overloading is a useful feature in OOP, one problem is that you can easily end up with many methods that essentially do the same thing for different classes
- For example, assume you need to build some methods that can switch two pieces of data using a simple swap function.

- Swap for integer type

```
// Swap two integers.
static void Swap(ref int a, ref int b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

swap for Person class

```
// Swap two Person objects.
static void Swap(ref Person a, ref Person b)
{
  Person temp = a;
  a = b;
  b = temp;
}
```

Custom Generic Methods

- Whenever you have a group of overloaded methods that differ only by incoming arguments, generics could make your life easier.
- Consider the following generic Swap<T> method that can swap any two Ts:

```
// This method will swap any two items using type parameter <T>.
  static void Swap<T>(ref T a, ref T b) {
      Console.WriteLine("You sent the Swap() method a {0}", typeof(T));
      T temp = a; a = b; b = temp;
  }
  static void Main(string[] args) {
      //Swap two integers
      int a = 10, b = 90;
      Console.WriteLine("Before swap: {0}, {1}", a, b);
      Swap<int>(ref a, ref b);
      Console.WriteLine("After swap: {0}, {1}", a, b);
      Console.WriteLine();
      // Swap two strings.
      string s1 = "Hello", s2 = "There";
      Console.WriteLine("Before swap: {0} {1}!", s1, s2);
      Swap<string>(ref s1, ref s2);
      Console.WriteLine("After swap: {0} {1}!", s1, s2);
  } }
```

```
Before swap: 10, 90
You sent the Swap() method a System.Int32
After swap: 90, 10

Before swap: Hello There!
You sent the Swap() method a System.String
After swap: There Hello!
```

Custom Generic Structures and Classes

- Similar to generic methods, you can construct generic structures and generic classes.

- Generic Structures and Generic Classes
  - Identical Process: replace all occurrences of the specific type(s) in class or structure with generic type <T>
  - For example - create Point<T> types

```
// Point using ints.
Point<int> p = new Point<int>(10, 10);
// Point using double.
Point<double> p2 = new Point<double>(5.4, 3.3);
```

Example - Generic Structure

```
public struct Point<T> {
    // Generic state date.
    private T xPos;
    private T yPos;
    // Generic constructor.
    public Point(T xVal, T yVal) {
        xPos = xVal;
        yPos = yVal;
    }
    // Generic properties.
    public T X {
        get { return xPos; }
        set { xPos = value; }
    }
    public T Y {
      get { return yPos; }
      set { yPos = value; }
    }
    //ToString function
    public override string ToString() => $"[{xPOs}, {yPos}]";
    // Reset fields to the default value of the type parameter.
    public void ResetPoint() {
        xPos = default(T);
        yPos = default(T);
    }
}

static void Main(string[] args) {
        // Point using ints.
        Point<int> p = new Point<int>(10, 10);
        Console.WriteLine("p.ToString()={0}", p.ToString());
        // Point using double.
        Point<double> p2 = new Point<double>(5.4, 3.3);
        Console.WriteLine("p2.ToString()={0}", p2.ToString());
}
```

Exercise

- Define classes for ticket reservations
- Specify different types of interactions between the classes and perform operations.
- Use all type of collections to perform the ticket reservation operations.
- Use other built-in methods in collections
- Create generic class/struct/methods

Windows Form

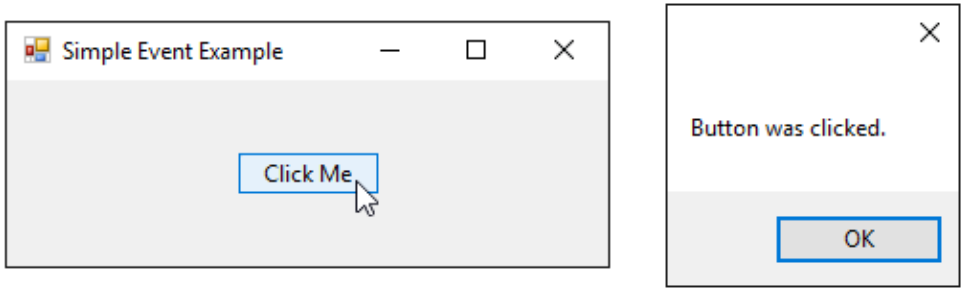
Windows Form

- GUI controls are objects that can display information on the screen or enable users to interact with an app.
- Several common GUI controls – same as Web Controls

Control	Description
`Label`	Displays *images* or *uneditable text*.
`TextBox`	Enables the user to *enter data via the keyboard*. It also can be used to *display editable or uneditable text*.
`Button`	Triggers an *event* when clicked with the mouse.
`CheckBox`	Specifies an option that can be *selected* (checked) or *unselected* (not checked).
`ComboBox`	Provides a *drop-down list* of items from which the user can make a *selection* either by clicking an item in the list or by typing in a box.
`ListBox`	Provides a *list* of items from which the user can make a *selection* by clicking one or more items.
`Panel`	A *container* in which controls can be placed and organized.
`NumericUpDown`	Enables the user to select from a *range* of numeric input values.

Event Handling

- GUIs are event driven.
- When the user interacts with a GUI component, the event drives the program to perform a task.
- A method that performs a task in response to an event is called an event handler.




```
1  // Fig. 14.5: SimpleEventExampleForm.cs
2  // Simple event handling example.
3  using System;
4  using System.Windows.Forms;
5
6  namespace SimpleEventExample
7  {
8     // Form that shows a simple event handler
9     public partial class SimpleEventExampleForm : Form
10    {
11       // default constructor
12       public SimpleEventExampleForm()
13       {
14          InitializeComponent();
15       }
16
17       // handles click event of Button clickButton
18       private void clickButton_Click(object sender, EventArgs e)
19       {
20          MessageBox.Show("Button was clicked.");
21       }
22    }
23 }
```

```
1  // Fig. 14.36: InterestCalculatorForm.cs
2  // Demonstrating the NumericUpDown control.
3  using System;
4  using System.Windows.Forms;
5
6  namespace NumericUpDownTest
7  {
8     public partial class InterestCalculatorFor
9     {
10       // default constructor
11       public InterestCalculatorForm()
12       {
13          InitializeComponent();
14       }
15
16       private void calculateButton_Click(object sender, EventArgs e)
17       {
18          // retrieve user input
19          decimal principal = decimal.Parse(principalTextBox.Text);
20          double rate = double.Parse(interestTextBox.Text);
21          int year = (int) yearUpDown.Value;
22
23          // set output header
24          string output = "Year\tAmount on Deposit\r\n";
25
26          // calculate amount after each year and append to output
27          for (int yearCounter = 1; yearCounter <= year; ++yearCounter)
28          {
29             decimal amount =  principal *
30               ((decimal) Math.Pow((1 + rate / 100), yearCounter));
31             output += $"{yearCounter}\t{amount:C}\r\n";
32          }
33
34          displayTextBox.Text = output; // display result
35       }
36    }
37 }
```