Event Driven Programming

Programming with Event Driven in C#

Chalew Tesfaye
Department of Computer Science
Debre Berhan University
2017

Objectives

- After the end this lesson the student will be able to
  - ✓ Understand C# language fundamentals
    - > Data type, variables and constants, …
  - ✓ Write a C# program statement
  - ✓ Develop OO program with C#

Lesson Outline

- Arrays and Collections
- Object oriented programming
  - ✓ Methods
  - ✓ Indexer, delegates, events and operators
  - ✓ Classes
  - ✓ Inheritance
  - ✓ Interface and generics

Methods, events and delegates

Subroutines in Computer Programming

Methods

- A method is a kind of building block that solves a small problem
  - ✓ A piece of code that has a name and can be called from the other code
  - ✓ Can take parameters and return a value
  - ✓ Can be public or private
- Methods allow programmers to construct large programs from simple pieces
- Methods are also known as functions, procedures, and subroutines

Why to Use Methods?

- More manageable programming
  - ✓ Split large problems into small pieces
  - ✓ Better organization of the program
  - ✓ Improve code readability
  - ✓ Improve code understandability
- Avoiding repeating code
  - ✓ Improve code maintainability
- Code reusability
  - ✓ Using existing methods several times

Declaring and Creating methods

- Each Method has
  - ✓ Name
  - ✓ Access modifier
  - ✓ Return type
  - ✓ Parameters/arguments
  - ✓ A body /statements

Syntax

```
access_modifier return_type name(parametrs){
    statements;
}
```

example

```
public double CalculateGpa(double totalGradePoint, int totalCredit){
    return totalGradePoint/totalCredit;
}
```

Calling Methods

- To call a method, simply use:
  - ✓ The method's name
  - ✓ Pass value
  - ✓ Accept return value if any
- Example
  - ✓ `double cgpa = calculateGpa(92.23, 36);`

Optional Parameters

- C# 4.0 supports optional parameters with default values:

```
void PrintNumbers(int start=0; int end=100)
{
  for (int i=start; i<=end; i++)
  {
    Console.Write("{0} ", i);
  }
}
```

- The above method can be called in several ways:

```
PrintNumbers(5, 10);
PrintNumbers(15);
PrintNumbers();
```

- If you define an optional parameter,
  - ✓ every parameters after that parameters must be defined as optional
- If you pass a value to an optional parameter by position,
  - ✓ you must pass a value to all parameters before that parameter

Named Parameters

- You can also pass value by name
  - ✓ code the parameter name followed by a colon followed by the value/argument name

```
void PrintNumbers(int start=0; int end=100)
{
  for (int i=start; i<=end; i++)
  {
    Console.Write("{0} ", i);
  }
}
PrintNumbers(end: 40, start: 35);
```

Variable Parameter Lists

- Function that take variable number of parameters
- Use keyword params

```
static int addNumbers(params int[] nums)
{
    int total = 0;

    foreach (int x in nums)
    {
        total += x;
    }
    return total;
}
```

Passing parameters by value and reference

- When calling a method, each argument can be passed by value or by reference,
  - ✓ difference in how they are handled in memory
- Pass by value
  - ✓ Original value will not be changed by the calling method
- Pass by reference
  - ✓ Original value can be changed by the calling method
  - ✓ to pass by reference use
    - > ref or
    - > out keyword
      - No need to initialize the argument, assign value to it within the calling method
      - allows a return value to be passed back to via a parameter

Passing Parameter -> Example

```
void PrintSum(int start; int end, int ref sum)
{
    for (int i=start; i<=end; i++)
    {
        sum+=I;
     }
}
//
int sum =0;
PrintSum(start, end, ref sum);
Console.WriteLine("Sum {0}", sum);
```

Passing Parameter -> Example

```
static void SquareAndRoot(double num, out double sq, out double sqrt)
{
        sq = num * num;
        sqrt = Math.Sqrt(num);
}
double n = 9.0;
double theSquare, theRoot;
SquareAndRoot(n, out theSquare, out theRoot);
Console.WriteLine("The square of {0} is {1} and its square root is {2}", n,
theSquare, theRoot);
```

Recursive methods

- The Power of Calling a Method from Itself
  - ✓ Example

```
static decimal Factorial(decimal num)
{
    if (num == 0)
        return 1;
    else
        return num * Factorial(num - 1);
}
```

Events, delegates and Indexer

- Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications.
- Applications need to respond to events when they occur.
  - ✓ For example, interrupts.
- Events are used for inter-process communication.
- A **delegate** is a reference type variable that holds the reference to a method.
- The reference can be changed at runtime.
- An **indexer** allows an object to be indexed such as an array.
- When you define an indexer for a class, this class behaves similar to a **virtual array**.
- Reading assignment about Events, delegates and Indexer

Delegates => Example

```
public delegate int NumberFunction (int x);
NumberFunction f = Square;
Console.WriteLine("result of the delegate is {0}", f(5));
// now change the delgate
f = Cube;
Console.WriteLine("result of the delegate is {0}", f(5));

static int Square(int num)
{
    return num * num;
}
static int Cube(int num)
{
     return num * num * num;
}
```

Events => Example

```
public delegate void myEventHandler(string
newValue);

class EventExample
{
      private string theValue;
      public event myEventHandler valueChanged;
      public string Val
      {
         set {
            this.theValue = value;
            this.valueChanged(theValue);
         }
      }
   }
```

```
EventExample myEvt = new EventExample();
myEvt.valueChanged += new
myEventHandler(myEvt_valueChanged);
string str;
do
{
    str = Console.ReadLine();
     if (!str.Equals("exit"))
          myEvt.Val = str;
 } while (!str.Equals("exit"));
static void myEvt_valueChanged(string newValue)
 {
   Console.WriteLine("The value changed to {0}",
newValue);
  }
```

Object Oriented Programming

Modeling Real-world Entities with Objects

Objects

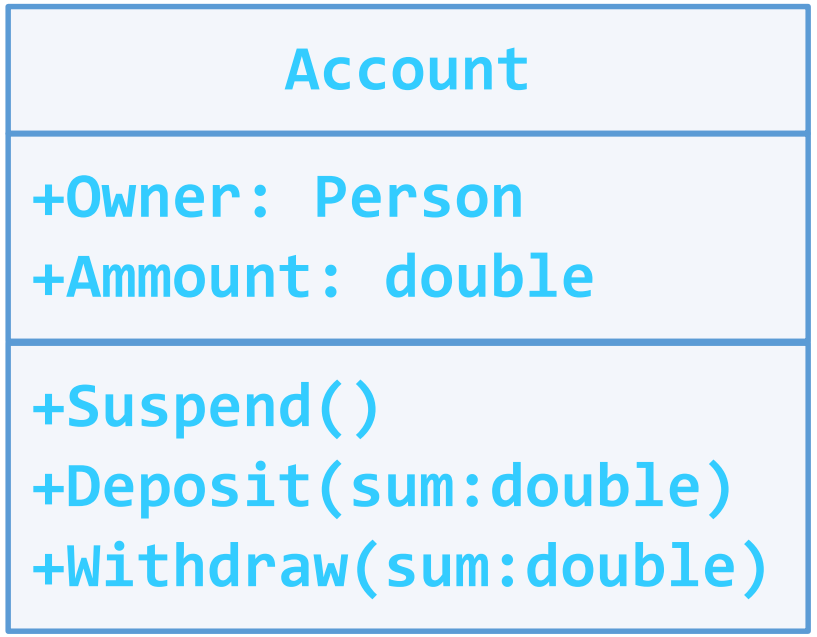
- Software objects model real-world objects or abstract concepts
  - ✓ Examples:
    - > bank, account, customer, dog, bicycle, queue
- Real-world objects have states and behaviors
  - ✓ Account' states:
    - > holder, balance, type
  - ✓ Account' behaviors:
    - > withdraw, deposit, suspend
- How do software objects implement real-world objects?
  - ✓ Use variables/data to implement states
  - ✓ Use methods/functions to implement behaviors
- An object is a software bundle of variables and related methods

Class

- Classes act as templates from which an instance of an object is created at run time.
- Classes define the properties of the object and the methods used to control the object's behavior.
- By default the class definition encapsulates, or hides, the data inside it.
- Key concept of object oriented programming.
- The outside world can see and use the data only by calling the build-in functions; called “methods”
- Methods and variables declared inside a class are called members of that class.

Objects vs Class

- An instance of a class is called an object.
- Classes provide the structure for objects
  - ✓ Define their prototype, act as template
- Classes define:
  - ✓ Set of attributes
    - > Represented by variables and properties
    - > Hold their state
  - ✓ Set of actions (behavior)
    - > Represented by methods
- A class defines the methods and types of data associated with an object



Classes in C#

- An object is a concrete instance of a particular class
- Creating an object from a class is called instantiation
- Objects have state
  - ✓ Set of values associated to their attributes
- Example:
  - ✓ Class: Account
  - ✓ Objects: Abebe's account, Kebede's account

Classes in C#

- Basic units that compose programs
- Implementation is encapsulated (hidden)
- Classes in C# can contain:
  - ✓ Access Modifiers
  - ✓ Fields (member variables)
  - ✓ Properties
  - ✓ Methods
  - ✓ Constructors
  - ✓ Inner types
  - ✓ Etc. (events, indexers, operators, …)

Classes in C#

- Classes in C# could have following members:
  - ✓ Fields, constants, methods, properties, indexers, events, operators, constructors, destructors
  - ✓ Inner types (inner classes, structures, interfaces, delegates, ...)
- Members can have access modifiers (scope)
  - ✓ `public`, `private`, `protected`, `internal`
- Members can be
  - ✓ `static` (common) or specific for a given object

Classes in C# – Examples

- Example of classes:
  - ✓ `System.Console`
  - ✓ `System.String` (`string` in C#)
  - ✓ `System.Int32` (`int` in C#)
  - ✓ `System.Array`
  - ✓ `System.Math`
  - ✓ `System.Random`

Simple Class Definition

```
public class Cat : Animal
{
  private string name;
  private string owner;
  public Cat(string name, string owner)
  {
    this.name = name;
    this.owner = owner;
  }
  public string Name
  {
    get { return name; }
    set { name = value; }
  }
 public string Owner
  {
    get { return owner;}
    set { owner = value; }
  }
  public void SayMiau()
  {
    Console.WriteLine("Miauuuuuuu!");
  }
}
```

Access Modifiers

- Class members can have access modifiers
  - ✓ Used to restrict the classes able to access them
  - ✓ Supports the OOP principle "encapsulation"
- Class members can be:
  - ✓ `public` – accessible from any class
  - ✓ `protected` – accessible from the class itself and all its descendent classes
  - ✓ `private` – accessible from the class itself only
  - ✓ `internal` – accessible from the current assembly (used by default)

Fields and Properties

- Fields are data members of a class
- Can be variables and constants
- Accessing a field doesn't invoke any actions of the object
- Example:
  - `String.Empty` (the `""` string)
- Constant fields can be only read
- Variable fields can be read and modified
- Usually properties are used instead of directly accessing variable fields
  - // Accessing read-only field
  - String empty = String.Empty;
  - // Accessing constant field
  - int maxInt = Int32.MaxValue;

Properties

- Properties look like fields (have name and type), but they can contain code, executed when they are accessed
- Usually used to control access to data fields (wrappers), but can contain more complex logic
- Can have two components (and at least one of them) called accessors
  - ✓ get for reading their value
  - ✓ set for changing their value
- According to the implemented accessors properties can be:
  - ✓ Read-only (get accessor only)
  - ✓ Read and write (both get and set accessors)
  - ✓ Write-only (set accessor only)
- Example of read-only property:
  - ✓ `String.Length`

The Role of Properties

- Expose object's data to the outside world
- Control how the data is manipulated
- Properties can be:
  - ✓ Read-only
  - ✓ Write-only
  - ✓ Read and write
- Give good level of abstraction
- Make writing code easier
- Properties should have:
  - ✓ Access modifier (`public`, `protected`, etc.)
  - ✓ Return type
  - ✓ Unique name
  - ✓ `Get` and / or `Set` part
  - ✓ Can contain code processing data in specific way

Defining Properties – Example

```
public class Point
{
    private int xCoord;
    private int yCoord;
    public int XCoord
    {
        get { return xCoord; }
        set { xCoord = value; }
    }
    public int YCoord
    {
        get { return yCoord; }
        set { yCoord = value; }
    }
    // More code ...
}
```

Instance and Static Members

- Fields, properties and methods can be:
  - ✓ Instance (or object members)
  - ✓ Static (or class members)
- Instance members are specific for each object
  - ✓ Example: different dogs have different name
- Static members are common for all instances of a class
  - ✓ Example: `DateTime.MinValue` is shared between all instances of `DateTime`

Instance and Static Members – Examples

- Example of instance member
  - ✓ `String.Length`
    - > Each string object has different length
- Example of static member
  - ✓ `Console.ReadLine()`
    - > The console is only one (global for the program)
    - > Reading from the console does not require to create an instance of it

Static vs. Non-Static

- Static:
  - ✓ Associated with a type, not with an instance
- Non-Static:
  - ✓ The opposite, associated with an instance
- Static:
  - ✓ Initialized just before the type is used for the first time
- Non-Static:
  - ✓ Initialized when the constructor is called

Methods

- Methods manipulate the data of the object to which they belong or perform other tasks
- Examples:
  - ✓ `Console.WriteLine(…)`
  - ✓ `Console.ReadLine()`
  - ✓ `String.Substring(index, length)`
  - ✓ `Array.GetLength(index)`

Static Methods

- Static methods are common for all instances of a class (shared between all instances)
  - ✓ Returned value depends only on the passed parameters
  - ✓ No particular class instance is available
- Syntax:
  - ✓ The name of the class, followed by the name of the method, separated by dot
  - ✓ <class_name>.<method_name>(<parameters>)

Method Overloading

- Overloading
  - ✓ The same name different way of calling a method
- Example:

```
class Shape
{
    public double CalculateArea(double w, double l)
    {
        return w*l;
    }
    public double CalculateArea(double l)
    {
        return l*l;
    }
}
Shape area = new Shape();
double rectangle = area. CalculateArea(12.3, 10.5);
double square= area. CalculateArea(10.5);
```

Method Overriding

- Being able to change or augment the behavior of method in a class
- virtual – tells the compiler that this method can be overridden by derived class
  - ✓ public virtual type methodName()
- override – in the subclass, tells the compiler that this method is overriding the same named method in the base class
  - ✓ public override type methodName()
- base – in subclass, calls the base class method
  - ✓ base. methodName();

Overriding > Example

```
class baseClass
{
    public virtual void doSomething()
   {
       Console.WriteLine("This is the baseClass saying hi!");
   }
}
class subClass : baseClass
{
     public override void doSomething()
     {
          base.doSomething();
          Console.WriteLine("This is the subClass saying hi!");
      }
}
baseClass obj1 = new subClass();
obj1.doSomething();
```

Abstract Class

- Abstract classes cannot be instantiated by themselves
  - ✓ you must define a subclass (derived class) and instantiate that instead
- Abstract classes have abstract members that derived class must override in order to provide functionality

```
public abstract class ClassName
{
    public abstract type method(args);
}
```

abstract > Example

```
abstract class myBaseClass
{
   public abstract int myMethod(int arg1, int arg2);
}
class myDerivedClass : myBaseClass
{
    public override int myMethod(int arg1, int arg2)
    {
        return arg1 + arg2;
    }
}
```

Sealed Class

- Sealed classes are the opposite of abstract classes
  - ✓ Abstract classes force you to drive a subclass in order to use them
  - ✓ Sealed classes prevent further subclasses from being derived
- Individual methods can also be marked as sealed, which prevents them from being overridden in subclass

```
sealed class myExampleClass
{
    public static string myMethod(int arg1)
    {
        return String.Format("You sent me the number {0}", arg1);
    }
}
class mySubClass : myExampleClass
{
}
```

Interface vs. Implementation

- The public definitions comprise the interface for the class
  - ✓ A contract between the creator of the class and the users of the class.
  - ✓ Should never change.
- Implementation is private
  - ✓ Users cannot see.
  - ✓ Users cannot have dependencies.
  - ✓ Can be changed without affecting users.
- The interface defines the 'what‘ part of the syntactical contract and
- The deriving classes define the 'how‘ part of the syntactical contract.

Interface > Example

```
public interface IShape
{
    //interface members
    double CalculateArea(double w, double l);
}
//implementation
public class Rectangle: Ishape
{
    double CalculateArea(double w, double l)
    {
        double area = w*l;
        return area;
    }
}
```

Constructors

- is a method with the same name as the class.
- It is invoked when we call *new* to create an instance of a class.
- In C#, unlike C++, you must call new to create an object.
  - ✓ Just declaring a variable of a class type does not create an object.
- Example

```
class Student {
        private string name;
        private fatherName;
        public Student(string n, string fn){
                name = n;
                fatherName = fn;
        }
 }
```

- If you don't write a constructor for a class, the compiler creates a default constructor.
- The default constructor is public and has no arguments.

Multiple Constructors

- A class can have any number of constructors.
- All must have different signatures.
  - ✓ The pattern of types used as arguments
- This is called overloading a method.
  - ✓ Applies to all methods in C#.
  - ✓ Not just constructors.
- Different names for arguments don’t matter, Only the types.

Structures

- Structures are similar to classes
- Structures are usually used for storing data structures, without any other functionality
- Structures can have fields, properties, etc.
  - ✓ Using methods is not recommended
- Structures are value types, and classes are reference types
  Example of structure
  - ✓ `System.DateTime` – represents a date and time

Namespaces

- Organizing Classes Logically into Namespaces
- Namespaces are used to organize the source code into more logical and manageable way
- Namespaces can contain
  - ✓ Definitions of classes, structures, interfaces and other types and other namespaces
- Namespaces can contain other namespaces
- For example:
  - ✓ `System` namespace contains `Data` namespace
  - ✓ The name of the nested namespace is `System.Data`
- A full name of a class is the name of the class preceded by the name of its namespace
- Example:
  - ✓ `Array` class, defined in the `System` namespace
  - ✓ The full name of the class is `System.Array`

Including Namespaces

- The using directive in C#:
  - using <namespace_name>
- Allows using types in a namespace, without specifying their full name
- Example:
  - using System;
  - DateTime date;
  - instead of
  - System.DateTime date;

Generic Classes

- Parameterized Classes and Methods
- Generics allow defining parameterized classes that process data of unknown (generic) type
  - ✓ The class can be instantiated with several different particular types
  - ✓ Example: `List<T>` → `List<int>` / `List<string>` / `List<Student>`
- Generics are also known as "parameterized types" or "template types"
  - ✓ Similar to the templates in C++
  - ✓ Similar to the generics in Java

Generics – Example

```
public class GenericList<T>
{
  public void Add(T element) { … }
}
class GenericListExample
{
  static void Main()
  {
    // Declare a list of type int
    GenericList<int> intList =  new GenericList<int>();
    // Declare a list of type string
    GenericList<string> stringList = new GenericList<string>();
  }
}
```

Preprocessor Directives

- Preprocessor directives give instructions to the compiler
- Should be defined before any line of code, at the very beginning
- Define or undefined symbols
  - ✓ #define
  - ✓ #undef
- To test if the symbol is defined or not
  - ✓ #if
  - ✓ #else
  - ✓ #elif
  - ✓ #endif
- For documentation of codes
  - ✓ #region
  - ✓ #endregion

Preprocessor Directives -> Example

```
#define DEBUGCODE
#define PRE

 #region This is the main function

 #endregion

#if DEBUGCODE
        Console.WriteLine("This is only in debug code");
#else
        Console.WriteLine("This only gets written out in non-debug code");
#endif

#if PRE
        Console.WriteLine(“Preprocessor directives!");
#endif
```

For more information

- Brian Bagnall, et al. C# for Java Programmers. USA. Syngress Publishing, Inc.
- Svetlin Nakov *et al*. Fundamentals of Computer Programming With C#. Sofia, 2013
- Joel Murach, Anne Boehm. Murach C# 2012, Mike Murach & Associates Inc USA, 2013

QUESTION?