SWE344

Internet Protocols and Client-Server Programming

Module 2b: C# Programming Essentials

Dr. El-Sayed El-Alfy alfy@kfupm.edu.sa

Mr. Bashir M. Ghandi bmghandi@ccse.kfupm.edu.sa

Computer Science Department

King Fahd University of Petroleum and Minerals

Objectives

- Learn more about how C# programs are organized
- Learn how to declare Methods and Classes
- Learn how Inheritance and Polymorphism are achieved
- Learn how to declare and implement Interfaces
- Learn about structures (struct) and Enumerators (enum)
- Learn how to raise and handle exceptions
- Learn how to do basic file IO using Streams.

Agenda

- OOP: Methods, and Classes
- OOP: Inheritance and Polymorphism

Organizing Types

- C# provides full object-oriented technology, including inheritance, polymorphism, and encapsulation
- A C# program is a collection of types
  - Defined in source files, organized by namespaces, and complied into assemblies (.exe or .dll files).
  - These organizational units generally overlap
    - a source file can contain many namespaces and a namespace can span several source files.
    - an assembly can contain several namespaces and a namespace can spread across several assemblies.
  - For simplicity, unless you have too many classes in a namespace, put related classes into a single namespace, in a single source file and compile it into a single assembly.
- In C#, the concept of a class, an interface, inheritance and polymorphism are very similar to what is known in Java.
- We shall concentrate on explaining the differences in these concepts between the two languages

Classes

- A class defines a template from which objects are created
- A class is declared using the class keyword

```
[access-modifiers] class class-name{
   class-body
}
```

- A class in C# can contain
  - Fields, constructors, methods and inner classes (helper classes)
  - Properties, events, indexers and operators
- Fields and methods may either be instance (default) or static

Example

```
1.   using System;
2.   namespace Banking {
3.       public class BankAccount {
4.          const double charityRate = 2.5;
5.          static int count;
6.          string name;
7.          int accountNumber;
8.          double balance;
9.          public BankAccount(string name) {
10.              this.name = name;
11.              accountNumber = ++count;
12.         }
13.         public BankAccount(string name, double amount) : this(name){
14.              balance = amount;
15.         }
16.         public void Deposit(double amount) {
17.              if (amount > 0)
18.                 balance += amount;
19.         }
20.         public void Withdraw(double amount) {
21.              if (balance >= amount)
22.                 balance -= amount;
23.         }
```

Example …

```
24.         public double GetBalance() {
25.                 return balance;
26.         }
27.         public double GetAnnaulCharity() {
28.                 double charity = balance * charityRate /100;
29.                 balance -= charity;
30.                 return charity;
31.         }
32.         public static void PrintCustomerCount() {
33.                 Console.WriteLine("Number of Customers = "+count);
34.         }
35.         public override String ToString() {
36.                 return "Acc #:"+accountNumber + ":"+name + ": "+balance;
37.         }
38.     }
```

Example …

```
39.      class TestAccount {
40.         public static void Main() {
41.               BankAccount acc1 = new BankAccount("Sami", 2000);
42.               BankAccount acc2 = new BankAccount("Omar");
43.                acc1.Deposit(3000);
44.               acc1.Withdraw(4000);
45.               Console.WriteLine(acc1);
46.               acc2.Deposit(5000);
47.               acc2.Withdraw(2000);
48.               Console.WriteLine(acc2);
49.               BankAccount.PrintCustomerCount();
50.           }
51.      } // end of TestAccount class
52.  } // end of namespace
53.
```

```
Acc #:1 Name:Sami Balance: 1000
Acc #:2 Name:Omar Balance: 3000
Number of Customers = 2
Press any key to continue . . .
```

Class Members

- Fields
  - A field is a member variable used to hold a value (represents an attribute).
  - You can apply several modifiers to a field, depending on how you want it to be used such as
    - const -- specifies that the value of the field or the local variable cannot be modified (A **const** field can only be initialized at the declaration of the field)
    - static -- declares a member that belongs to the type itself rather than to a specific object.
    - readonly – declares a field that can only be assigned values as part of the declaration or in a constructor in the same class; **readonly** fields can have different values depending on the constructor used; while a **const** field is a compile-time constant, the **readonly** field can be used for runtime constants

Class Members

- Fields …
  - Examples

```
public const double x = 1.0, y = 2.0, z = 3.0;
public static const int c1 = 5.0;
public static const int c2 = c1 + 100;
```

  - Default field values

Type	default value
All numeric types	0
bool	false
char	'\0'
string or object reference	null

Class Members …

- Methods
  - A method is a group of declarations and other statements that perform a specific task (define the behavior of the class instances)
  - Defining a method

```
[access-modifiers] return-type method-name([para-type param-name, …..]){
    method-body
}
```

  - If a method returns a value, it must have a return statement
  - If a method does not return a value, the return statement is optional and the return type must be void
  - You can define local variable inside the method
  - If a parameter or local variable has the same name as a field name, the field name is hidden
    - To access the field name use this keyword (a reference to the current object)

Class Members …

- Methods …
  - The keyword this is used for two main purposes:
    - Resolving name conflict between instance variables and method or constructor parameters.
    - Calling another constructor from a constructor in the same class. However, we note that the call is placed in the header of the calling constructor
    - Example

```
public BankAccount(string name, double amount):
 this(name){
      balance = amount;
}
```

Class Members …

- Passing parameters
  - By value – changes made to the parameter inside the method are not affecting the actual variable in the method call
    - The object reference is passed as a value but it can be used to the content of the object
  - By reference – use ref before the parameter in the method signature and call; in this case changes to the parameter affects the variable in the method call
    - Try to minimize using call by reference
  - When passing parameters by value or by reference, the variables that are passed must be assigned values before the method is called
- Out parameters
  - A parameter can be declared to be out in the method signature and call meaning it is used to return a value (similar to passing by reference except that the variable is not initialized before passing it to the method)

Class Members …

- Calling a method
  - A class (static) method is called by

```
Class-Name.Method-Name(arguments)
```

  - An instance method is called

```
Object-Reference.Method-Name(arguments)
```

- Method overloading
  - Define methods in a class that have the same name but different parameters (different signatures)

Class Members …

- Constructors
  - A method that has the same name as the class name (usually used to initialize fields using parameters)
  - A constructor does not have a return type
  - If no constructor is defined, there will a default one
  - You can define multiple overloaded constructors that accept different parameters
  - You can define constructors that allow copying the fields from one object to another (copy constructors)
    - Example

```
Student(Student x){
  name  = x.name;
  quiz1 = s.quiz1;
  quiz2 = s.quiz2;
}
```

Class Members …

- Destructor
  - Can be used to do something immediately before removing an object by the garbage collector, e.g. closing an opened file
  - Has the same name as the class preceded by ~
  - Does not take any parameter and does not have a return type
  - Example

```
~Student(){
    // things to be done
}
```

Class Members ...

- Properties
  - Are the normal get and set methods we have in Java but in C# the set and get operations are unified into a single unit.
  - They are sometimes called smart fields because they're actually methods that look like fields to the class's clients
    - They behave exactly like methods. They are inherited by subclasses and they can be hidden or overridden. They can have any of the modifiers that a normal method can have.
  - Allow the client a greater degree of abstraction because it doesn't have to know whether it's accessing the field directly or whether an accessor method is being called.
  - To define a property, you must have at least one of get or set blocks.
  - Notice that compiler automatically defines a variable, value, in the set block to receive the set argument.
  - Private fields and properties promote encapsulation

Class Members ...

- Properties…

```
1.  class BankAccount {
2.      private double balance;
3.      //....
4.
5.      public double Balance {      // define a property
6.        get{ return balance; }
7.        set{ balance = value; }    // value is implicit parameter
8.      }
9.   }
10. //...
11. // create a bank account
12. BankAccount acc = new BankAccount();
13. acc.Balance = 12000.0;   // implicit call to set
14. double z = acc.Balance;   // implicit call to get
```

Class Members ...

- Using Access Modifiers
  - To achieve encapsulation, a type may hide itself from other types or other assemblies by adding one of the following access modifies:

public	Members marked public are visible to any method of any class. Default for interfaces and enums.
private	Members in class A that are marked private are accessible only to methods of class A. Default for classes (and structs)
Protected	Members in class A that are marked protected are accessible to methods of class A and also to methods of classes derived from class A.
internal	Members in class A that are marked internal are accessible to methods of any class in A's assembly.
protected internal	Members in class A that are marked protected internal are accessible to methods of class A, to methods of classes derived from class A, and also to any class in A's assembly. This is effectively protected OR internal.

Objects

- To declare a reference for a bank account

```
BankAccount acc1; // acc1 is null
```

- To create an object, use new operator and a constructor

```
acc1 = new BankAccount("Omar");
```

- Before trying to access an object's fields or methods through an object reference, the object reference must refer to a real object (i.e., is not null)

Inheritance

- To achieve code re-usability, a class can inherit from another class – in C#, only single inheritance is allowed.
- There is no “extends” keyword. Instead, a colon is used after the header of the derived class followed by base class identifier
  - A class can extend only one class but it can implement several interfaces
  - The super class and the interfaces are listed after the colon separated by commas.
  - If there is a super class being extended, then it must appear first in the list.
- The base keyword:
  - is used instead of the Java’s super, to refer to a superclass member.
  - is used to call the constructor of the base class from within a subclass. However, like this keyword, such a call should be in the heading of the calling constructor.

Example

```
1.  class BankAccount{
2.    private string num;
3.    private double balance;

4.    public BankAccount(string num, double balance){
5.        this.num = num ;
6.        this.balance = balance;
7.    }
8.    //...
9.  }
10. class SavingAccount:BankAccount {
11.    private double interest;
12.
13.    public SavingAccount(string num, double balance,
14.                         double interest): base(num, balance){
15.        this.intreset = interest;
16.    }
17.    //...
18. }
```

Inheritance …

- Overriding and hiding:
  - In C#, overriding is not allowed by default.
  - The base class must indicate that it is willing to allow its method to be overridden by declaring the method as virtual, abstract or override.
  - The subclass must also indicate that it is overriding the method by using the override keyword.
  - The effect of overriding is the same as in Java – Polymorphism. At run-time, a method call will be bound to the method of the actual object.
  - A subclass may also decide to hide an inherited method instead of overriding it by using the new keyword

Example

```
1.   using System;
2.   class A {
3.        public virtual void method() {
4.           Console.WriteLine(" In A");
5.        }
6.   }
7.   class B : A {
8.        public override void method() { // override inherited method
9.           Console.WriteLine("In B");
10.       }
11.  }
12.  class C : B {
13.       public new void method() {  // hide inherited method
14.          Console.WriteLine("In C");
15.       }
16.  }
17.  class Test {
18.       public static void Main() {
19.          C c = new C();  c.method();   // calls C's method
20.          B b = c; b.method();  //calls B's method
21.          A a = c; a.method();  //calls B's method
22.       }
23.  }
```

```
In C
In B
In B
Press any key to continue . . .
```

Casting Objects

- Upcast: casting an object of a derived class to the base class
- Downcast: casting an object of a base class to a derived class
- The *as* operator is used for type-conversion (down-casting).
  - Example

```
Student s = new GraduateStudent(…);
GraduateStudent gs;
gs = (GraduateStudent) s;
gs = s as GraduateStudent; // another pretty equivalent
```

  - The only difference is when the object in s is not compatible with GraduateStudent. In that case, the first statement throws InvalidCastException, while the second assigns null to gs.
- The *is* operator is like the instanceof operator in Java. It checks if an object is compatible with a type.
  - Example:

```
if (s is GraduateStudent)
    gs = s as GraduateStudent;
```

Abstract Classes

- A class that is declared using the abstract keyword
- It may have abstract methods as well as implemented methods
- An abstract method provides a method name and signature and must be implemented in all derived classes.
- Abstract classes establish a base for derived classes, but it is not legal to instantiate an object of an abstract class.
- An abstract class can be derived from another abstract class

```
1.   using System;

2.   public abstract class Shape {
3.       public String name() {
4.               return GetType().Name;
5.       }
6.       public abstract double Area();
7.       public abstract double Perimeter();

8.       public override String ToString() {
9.               return "ShapeType:"+name() + ":" + Perimeter() + ":" + Area();
10.      }
11.  }
```

Sealed Classes

- Similar to final classes in Java, classes marked sealed can not be used to derive other classes

```
sealed public class Student{
….
}
```

- You can also mark a method as sealed to prevent overriding it
- A method can be marked as sealed in a non-sealed class to prevent overriding it in a derived class
- Classes are most often marked sealed to prevent accidental inheritance.

Notes

- A type or type member cannot be declared to be more accessible than any type it uses.
  - For example, a class cannot be public if it extends an internal class.
  - A method cannot be protected if the type of one of its parameters is internal.
- Also access modifiers cannot be used when they conflict with the purpose of inheritance.
  - For example, an abstract method cannot be private.
  - Similarly, a sealed class cannot define new protected members since no class can benefit from them.