Advanced Programming Methods
Lecture 4

Contents

- Structs
- Operator Overloading
- Delegates
- Implicitly Typed Local Variables
- Lambda Expressions
- Extension methods
- Events (Observer pattern)

Structs

- A struct is similar to a class, with the following key differences:
  - A struct is a value type, whereas a class is a reference type.
  - A struct does not support inheritance (other than implicitly deriving from object).
- A struct can have all the members a class can, except the following:
  - A default constructor
  - A finalizer
  - Virtual members

```
public struct Complex
{
  double re, im;
  public Complex (double re, double im) {this.re = re; this.im = im;}
}
...
Complex c1 = new Complex ( );       // c1.re and c1.im will be 0.0
Complex c2 = new Complex (1, 1);   // c2.re and c2.im will be 1.0
```

Struct

Remarks

A default constructor that cannot be overridden implicitly exists. It performs a bitwise-zeroing of its fields.

In a struct constructor, every field must be explicitly initialized.

There cannot be field initializers in a struct.

```
public struct Point
{
  int x = 1;
  int y;
  public Point( ) {}  //error
  public Point(int x) {this.x = x;} //error
 }
```

Nested Types

A *nested type (class or struct)* is declared within the scope of another type.

```
class List{
   public class Node{…}
}
```

Remarks

A nested type can access only the enclosing `static` members (even `private`).

It can be declared with the full range of access modifiers (not just `public` or `internal`).

The default visibility for a nested type is `private`.

Accessing a nested type from outside the enclosing type requires qualification with the enclosing type's name.

```
List.Node n;
```

Operator Overloading

Overloadable symbolic operators

<table>
<tr><td>+ (unary)</td><td>- (unary)</td><td>!</td><td>~</td><td>++</td></tr>
<tr><td>--</td><td>+</td><td>-</td><td>*</td><td>/</td></tr>
<tr><td>%</td><td>&</td><td>|</td><td>^</td><td><<</td></tr>
<tr><td>>></td><td>==</td><td>!=</td><td>></td><td><</td></tr>
<tr><td>>=</td><td><=</td><td></td><td></td><td></td></tr>
</table>

The following operators are also overloadable:

Implicit and explicit conversions (with the **`implicit`** and **`explicit`** keywords)

The literals **`true`** and **`false`**

The following operators are indirectly overloaded:

The compound assignment operators (e.g., **`+=, /=)`** are implicitly overridden by overriding the noncompound operators (e.g., **`+`**, **`=`**).

The conditional operators **`&&`** and **`||`** are implicitly overridden by overriding the bitwise operators **`&`** and **`|`**.

Operator Functions

An operator is overloaded by declaring an *operator function*. An operator function has the following rules:

- The name of the function is specified with the `operator` keyword followed by an operator symbol.
- The operator function must be marked `static`.
- The parameters of the operator function represent the operands.
- The return type of an operator function represents the result of an expression.
- At least one of the operands must be the type in which the operator function is declared.

Operator overloading

```
class Complex    {
  double re, im;
  public Complex(double re, double im){this.re=re; this.im = im;}
  public static Complex operator +(Complex a, Complex b) {
    return new Complex(a.re + b.re, a.im + b.im);
  }
  public  static bool operator ==(Complex a, Complex b) {
    return (a.re == b.re) && (a.im == b.im);
  }
  public static bool operator !=(Complex a, Complex b){
    return !(a==b);
  }
  //…
}
```

Operator overloading

Remarks

The C# compiler enforces operators that are logical pairs to both be defined. These operators are (`== !=`), (`< >`), and (`<= >=`).

If you overload (`==`) and (`!=`), you need to override the `Equals` and `GetHashCode` methods defined on `Object`. The C# compiler will give a warning if you do not override them.

If you overload (`< >`) and (`<= >=`), you should implement `IComparable` and `IComparable<T>`.

Delegates

Delegates are references to methods.

They are similar to function pointers from C++.

Delegates are similar to object references, but they are used to reference methods instead of objects.

A delegate has three properties:

The type or signature of the method that the delegate can point to.

The delegate reference which can be used to reference a method.

The actual method referenced by the delegate.

Delegates declaration, using `delegate` keyword:

```
delegate <return type> DelegateName(<list of parameters>);
```

Example:

```
delegate int ArithmeticMethod(int a, int b);
```

User defined delegates are subclasses of `System.Delegate` class. The class is automatically generated by the compiler, it cannot be explicitly created by the user.

Delegates

Initialization and usage

```
delegate String StringEncoder(String text);
class Test{
 public void Main(string args[]){
   String text="Ana are mere";
   EncoderUtils se=new EncoderUtils();
   StringEncoder enc1=ToLower;
   StringEncoder enc2=new StringEncoder(se.encodeA);
   StringEncoder enc3=new StringEncoder(se.encodeB);
   Console.WriteLine("ToLower ={0}", enc1(text));
   Console.WriteLine("encodeA ={0}", enc2(text));
   //…
 }
 static String ToLower(String text){
   return text.ToLower();
 }
}
class EncoderUtils{
 public String encodeA(String text){ … }
 public String encodeB(String txt){ … }
 public int encodeC(String txt){ … }
}
```

Multicast Delegates

All delegate instances have multicast capability.

A delegate instance can reference not just a single target method, but also a list of target methods. The `+=` operator combines delegate instances, and `-=` operator removes delegates instances.

```
String text="Ana are mere";
EncoderUtils se=new EncoderUtils();
StringEncoder enc=ToLower;
enc+=new StringEncoder(se.encodeA);
enc+=new StringEncoder(se.encodeB);
enc(text);   //all three methods are called, in the
//order they were added
enc-=ToLower;
enc(text);   //only two methods are called
```

A multicast delegate inherits from `System.MulticastDelegate` (that inherits from `System.Delegate`)

Multicast Delegates

If a multicast delegate has a non void return type, the caller receives the return value from the last method to be invoked. The preceding methods are still called, but their return values are discarded.

C# compiles `+=` and `-=` operations made on a delegate to the static `Combine` and `Remove` methods of the `System.Delegate` class.

When a delegate instance is assigned an instance method, the delegate instance must maintain a reference not only to the method, but also to the instance of that method. The `Target` property of the `System.Delegate` class represents this instance. If the method is `static`, the result is `null.`

Implicitly Typed Local Variables

Starting with C# 3.0 it is possible to declare and initialize a local variable without explicitly specifying the type.

If the compiler is able to infer the type from the initialization expression, the keyword **`var`** can be used in place of the type declaration.

```
var x = 5;
var y = "hello";
var z = new System.Text.StringBuilder();
```

It is equivalent to:

```
int x = 5;
String y = "hello";
System.Text.StringBuilder z = new System.Text.StringBuilder();
```

Implicitly Typed Local Variables

Implicitly typed variables are statically typed:

```
var x = 5;
x = "hello";    // Compile-time error; x is of type int
```

- `var` can decrease code readability when the type cannot be deduced just by looking at the variable declaration.

```
Random r = new Random();
var x = r.Next();      //int
```

Lambda Expressions

- A *lambda expression* is an unnamed method written in place of a delegate instance.
- They were introduced in C# 3.0.
- The compiler immediately converts the lambda expression to either:
  - A delegate instance.
  - An expression tree, of type Expression<TDelegate>, representing the code inside the lambda expression in a traversable object model. This allows the lambda expression to be interpreted later at runtime.

Example

```
    delegate int Transformer (int i);

Transformer sqr = x => x * x;      //lambda expression
Console.WriteLine (sqr(3)); // 9
```

Lambda Expressions

- A lambda expression has the following form:

  `(parameters) => expression-or-statement-block`

- The parentheses can be omitted if and only if there is exactly one parameter of an inferable type.
- Each parameter of the lambda expression corresponds to a delegate parameter, and the type of the expression (which may be `void`) corresponds to the return type of the delegate.
- A lambda expression's code can be a statement block instead of an expression.

  ```
  x => { return x * x; };
  ```

- When the compiler cannot infer the type of the lambda parameter contextually, you must specify the type explicitly:

  ```
  (int x) => x * x
  ```

Extension Methods

- *Extension methods* allow an existing type to be extended with new methods without altering the definition of the original type. They were added in C# 3.0.
- An extension method is a static method of a static class, where the `this` modifier is applied to the first parameter. The type of the first parameter will be the type that is extended.

```
public static class StringHelper
{
    public static bool IsCapitalized (this string s)
    {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
    }
}
```

Call:

```
    String location="Cluj";
    Console.WriteLine (location.IsCapitalized());
```

Extension Methods

- An extension method call, when compiled, is translated back into an ordinary static method call:

```
Console.WriteLine (StringHelper.IsCapitalized (location));
```

- The translation works as follows:

```
arg0.Method (arg1, arg2, ...); // Extension method call
StaticClass.Method (arg0, arg1, arg2, ...); // Static method
```

- Remarks:
  - An extension method cannot be accessed unless the namespace is in scope. You have to use the `using` directive.
  - Any compatible instance method (having the same signature) will always take precedence over an extension method. The extension method can still be called using its normal static syntax.
  - If two extension methods have the same signature, the extension method must be called as an ordinary static method to disambiguate the method to call.

Events

Events are a language feature that formalizes the Publisher/Subscriber (Observer) pattern.

An *event* is a wrapper for a delegate that exposes just the subset of delegate features required for the publisher/subscriber model.

The main purpose of events is to prevent subscribers from interfering with each other.

To declare an event member, the `event` keyword is put in front of a delegate member.

Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Publisher(Subject)-Subscriber(Observer) relationship:

A publisher is one who publish data and notifies it to the list of subscribers who have subscribed for the same to that publisher.

Example:

A simple example is Newspaper.

Whenever a new edition is published by the publisher,it will be circulated among subscribers whom have subscribed to publisher.

Events

1. Define a public delegate

```
public delegate void DelegateEvent(Object sender, EventArgs args);
```

The first parameter is usually the originator of the event, and the second parameter usually holds any additional data to be passed to the event handler.

2. Define a class that generates or raises the event (`Publisher`). Inside this class a public event is declared.

```
class Publisher{
 public event DelegateEvent eventName;
  …
 … someMethod(…){
    //code that raises an event
    EventArgsSubClass args=new EventArgsSubClass(<some data>);
    eventName(this, args);
    //or eventName(this, null);
 }
}
```

Events

3. Define the class(es) that handle the appearance of an event (`Observer`). The name of the event handler conventionally starts with “On”.

```
class Observer{
  //the methods that matches the delegate signature
  public void OnEventName(Object sender, EventArgs args){
  //event handling code
  } …}
```

- Configuration

```
class StartApp{
  … Main(){
  //create the publisher(subject)
  Publisher pub=new Publisher(…);
  //create the observers
  Observer obs1=new Observer(…);
  Observer obs2=new Observer(…);
  //subscribe the observers to the event
  pub.eventName+=new DelegateEvent(obs1.OnEventName);
  pub.eventName+=new DelegateEvent(obs2.OnEventName);
  pub.someMethod(…);//explicit call of the method that raises the event
  }
}
```

Events Example

```
public delegate void TimerEvent(object sender, EventArgs args);
class ClockTimer{
   public event TimerEvent timer;
   public void start(){
    for(int i=0;i<3;i++){
        timer(this, null);
        Thread.Sleep(1000);
    }
   }
}
class Test{
   static void Main(){
    ClockTimer clockTimer=new ClockTimer();
    clockTimer.timer+=new TimerEvent(OnClockTick);
    clockTimer.start();
   }
   public static void OnClockTick(object sender, EventArgs args){
    Console.WriteLine("Received a clock tick event!");
   }
}
```

Events

Remarks

For reusability, the `EventArgs` is subclassed and it is named according to the information it contains. It typically exposes data as properties or as read-only fields.

The rules for choosing or defining a delegate for the event are:

- It must have a void return type.
- It must accept two arguments: the first of type `object`, and the second a subclass of `EventArgs`. The first argument indicates the event publisher, and the second argument contains the extra information to be passed.

Starting with .NET Framework 2.0 a generic delegate, called `System.EventHandler<T>,` is defined, that satisfies these rules.

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e) where TEventArgs : EventArgs;
//publisher class
public event EventHandler<TEventArgs> concreteEvent;
protected virtual void OnEvent (TEventArg e) {
    if (concreteEvent != null) concreteEvent (this, e);
  }
```

Events vs Properties

An event can be implemented as a property:

```
public delegate void DelegateEvent(object sender, EventArgs args);
public class Publisher{
 private DelegateEvent concreteEvent;
 …
 public DelegateEvent Event{
   get {return concreteEvent;}
   set{ concreteEvent=value;}
 } … }
```

Disadvantages:

Replace other subscribers by reassigning `Event` (instead of using the `+=` operator).

```
publisher.Event=new DelegateEvent(someMethod);
```

Clear all subscribers (by setting `Event` to `null`).

```
publisher.Event=null;
```

Broadcast to other subscribers by explicitly invoking the delegate:

```
publisher.Event(null, arguments);
```

Event accessors

An event's accessors are the implementations of its `+=` and `-=` functions.

By default, accessors are implemented implicitly by the compiler.

```
public event TimerEvent timer;
```

The compiler converts this to the following:

A private delegate field;

A public pair of event accessor functions, whose implementations forward the `+=` and `-=` operations to the private delegate field.

It is possible to explicitly define event accessors:

```
private TimerEvent _timer;          // declare a private delegate
public event TimerEvent timer{
 add    { _timer += value;  }
  remove  { _timer -= value;  }
}
```

The `add` and `remove` parts of an event are compiled to `add_XYZ` and `remove_XYZ` methods.

The `+=` and `-=` operations on an event are compiled to calls to the `add_XYZ` and `remove_XYZ` methods.