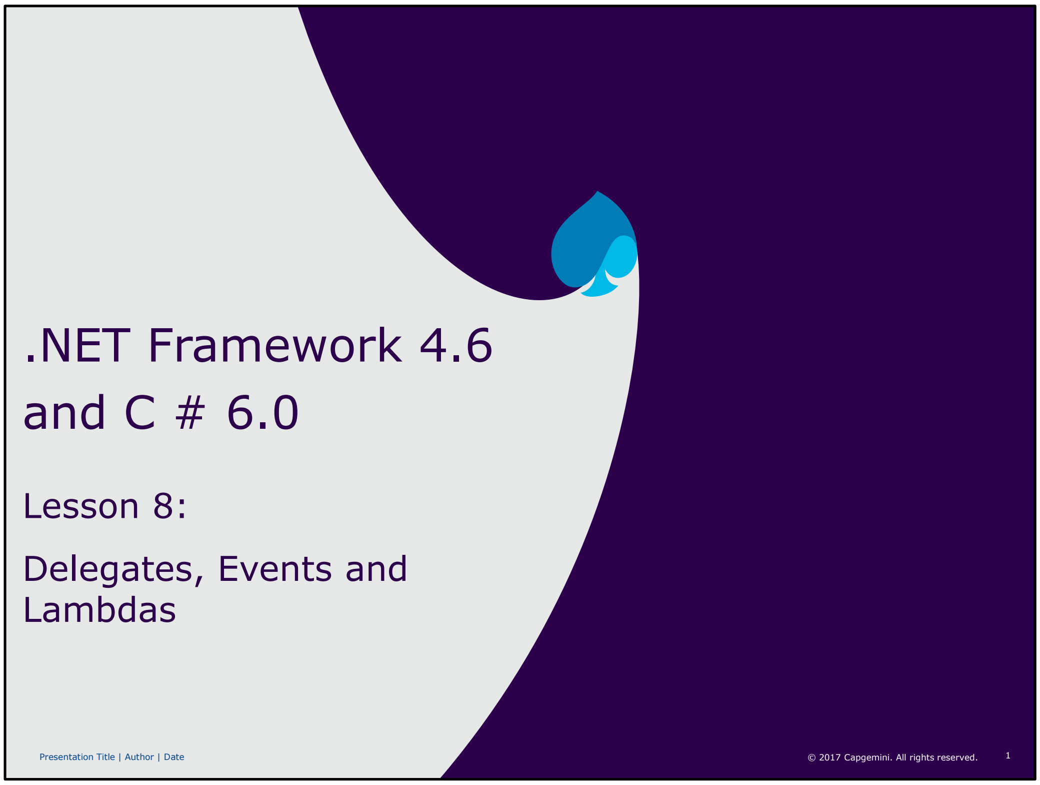


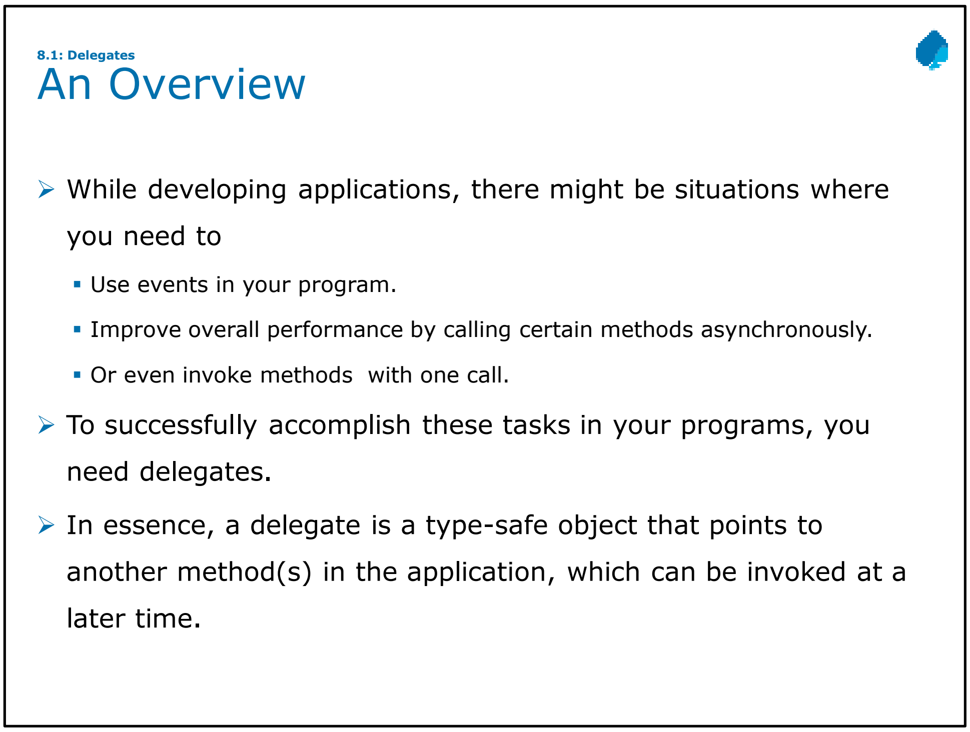
.NET Framework 4.6 and C # 6.0

Lesson 8:

Delegates, Events and Lambdas

Lesson Objectives

- In this lesson we will cover the following :
  - Understand the concept of a delegate.
  - Understand how to declare a delegate, assign it to a method and invoke it.
  - Understand Multicast Delegates and its uses.
  - Understand Generic Delegates
  - Understand Event handling in C#.
  - Understand Lambdas.


8.1: Delegates

An Overview

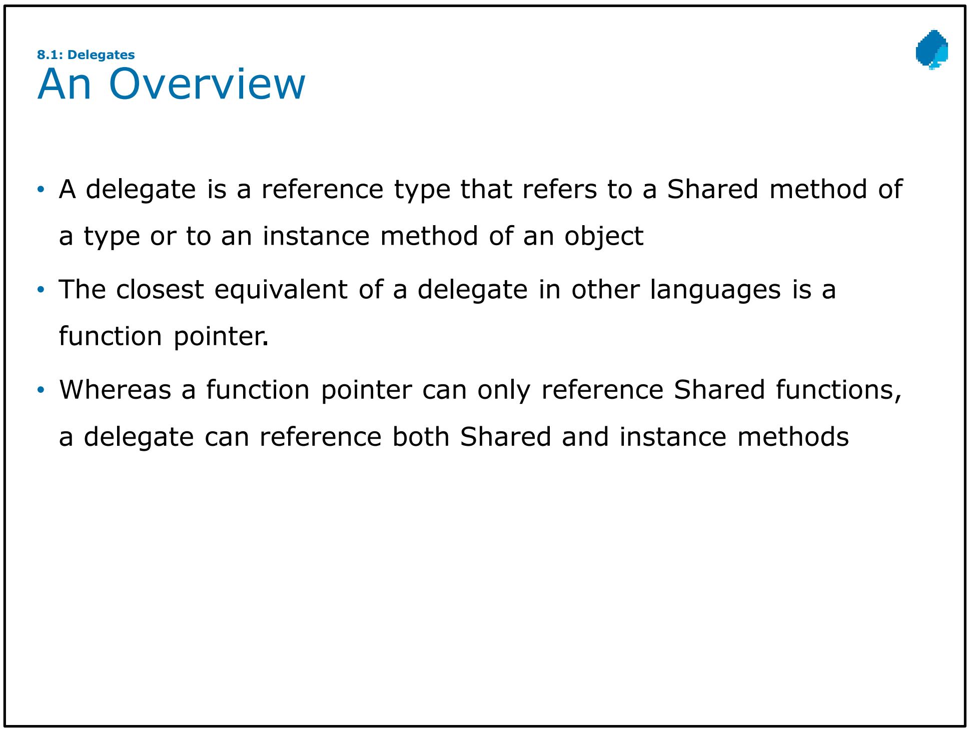
- While developing applications, there might be situations where you need to
  - Use events in your program.
  - Improve overall performance by calling certain methods asynchronously.
  - Or even invoke methods with one call.
- To successfully accomplish these tasks in your programs, you need delegates.
- In essence, a delegate is a type-safe object that points to another method(s) in the application, which can be invoked at a later time.

An Overview –

The CLR provides a runtime that explicitly supports a flexible callback mechanism. From the beginning of time, or at least from the beginning of Windows time, there has always been the need for a callback function that the system, or some other entity, calls at specific times to notify you of something interesting. This mechanism works just fine in a C-based programming environment.

Delegates are now the preferred method of implementing callbacks in the CLR. The delegate contains a couple of useful fields. The first one holds a reference to an object, and the second holds a method pointer. When you invoke the delegate, the instance method is called on the contained reference.

By implementing delegates, you can also separate the invocation of a method from its definition. By referencing a method as a delegate, you can also treat the method as an object. You can also add multiple method references to the invocation list of a delegate. Then, when the delegate is invoked, you can call all the referenced methods.



8.1: Delegates

An Overview

- A delegate is a reference type that refers to a Shared method of a type or to an instance method of an object
- The closest equivalent of a delegate in other languages is a function pointer.
- Whereas a function pointer can only reference Shared functions, a delegate can reference both Shared and instance methods

Delegates
Delegates enable scenarios that other languages - such as C++, Pascal, and Modula—have addressed with function pointers. Unlike C++ function pointers, delegates are fully object oriented; unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.

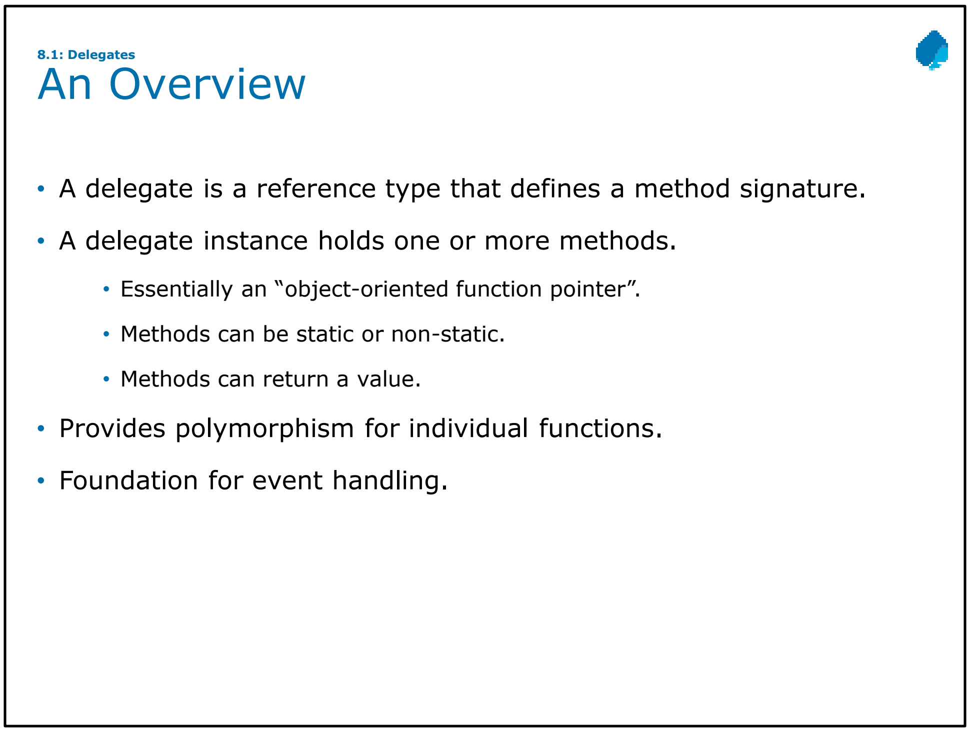
A delegate lets you pass a function as a parameter. The type safety of delegates requires the function you pass as a delegate to have the same signature as the delegate declaration.

A delegate declaration defines a class that derives from the class System.Delegate. A delegate instance encapsulates one or more methods, each of which is referred to as a callable entity. For instance methods, a callable entity consists of an instance and a method on the instance. For static methods, a callable entity consists of just a method. Given a delegate instance and an appropriate set of arguments, one can invoke all of that delegate's instance's methods with that set of arguments.

Delegates are used extensively in programming for Windows, callback methods are used when you need to pass a function pointer to another function that will then call you back (via the passed pointer). Callbacks serve many purposes, but the following are the most common:

Asynchronous processing - An example of using a callback method in asynchronous processing is when the code being called will take a good deal of time to process the request. Typically, the scenario works like this: Client code makes a call to a method, passing to it the callback method. The method being called starts a thread and returns immediately. The thread then does the majority of the work, calling the callback function as needed. This has the obvious benefit of allowing the client to continue processing without being blocked on a potentially lengthy synchronous call.

Injecting custom code into a class's code path - Another common use of callback methods is when a class allows the client to specify a method that will be called to do custom processing. Let's look at an example in Windows to illustrate this. Using the ListBox class in Windows, you can specify that the items be sorted in ascending or descending order. Besides some other basic sort options, the ListBox class can't really give you any latitude and still be a generic class. Therefore, the ListBox class also enables you to specify a callback function for sorting. That way, when ListBox sorts the items, it calls the callback function and your code can then do the custom sorting you need.



8.1: Delegates

An Overview

- A delegate is a reference type that defines a method signature.
- A delegate instance holds one or more methods.
  - Essentially an "object-oriented function pointer".
  - Methods can be static or non-static.
  - Methods can return a value.
- Provides polymorphism for individual functions.
- Foundation for event handling.

A delegate in C# is similar to a function pointer in C/C++. You can explain a little about Function pointers and how delegates are OOP way to define function pointers

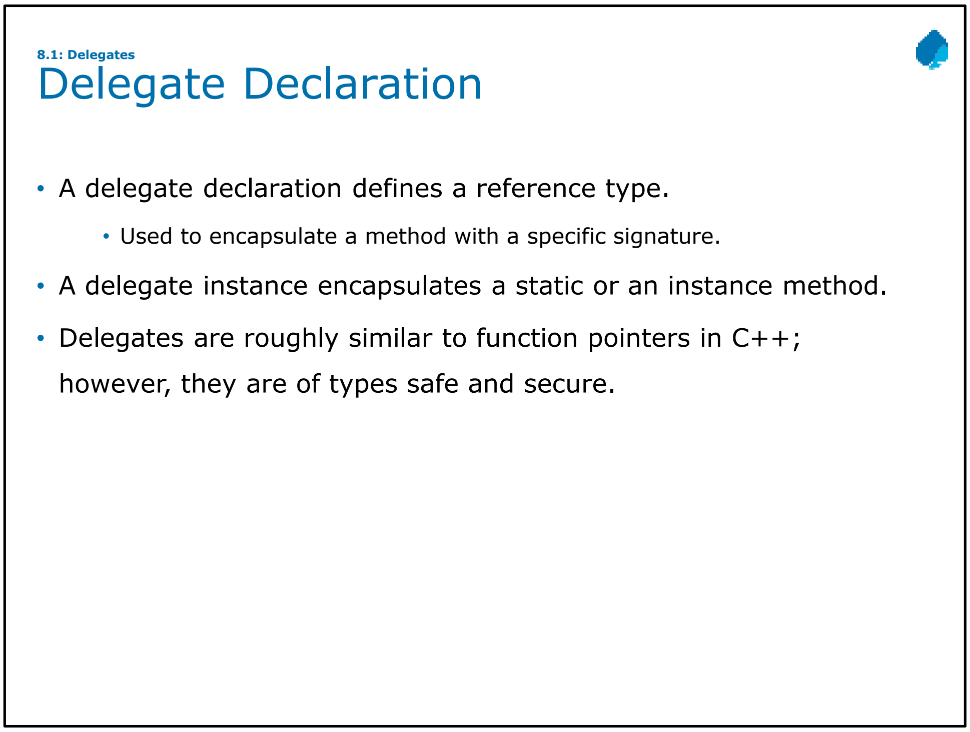
In straightforward language, a delegate is an object that can refer to a method. Thus, when you create a delegate, you are creating an object that can hold a reference to a method. Furthermore, the method can be called through this reference. Thus, a delegate can invoke the method to which it refers.

Even though a method is not an object, it too has a physical location in memory, and the address of its entry point is the address called when the method is invoked. This address can be assigned to a delegate. Once a delegate refers to a method, the method can be called through that delegate.

It is important to understand that the same delegate can be used to call different methods during the runtime of a program by simply changing the method to which the delegate refers. Thus, the method that will be invoked by a delegate is not determined at compile time, but rather at runtime. This is the principal advantage of a delegate.

Why Delegates

- In general, delegates are useful as they:
  - Support events.
  - Give your program a way to execute methods at runtime without precise knowledge of which method it is at compile time.



8.1: Delegates

Delegate Declaration

- A delegate declaration defines a reference type.
  - Used to encapsulate a method with a specific signature.
- A delegate instance encapsulates a static or an instance method.
- Delegates are roughly similar to function pointers in C++; however, they are of types safe and secure.

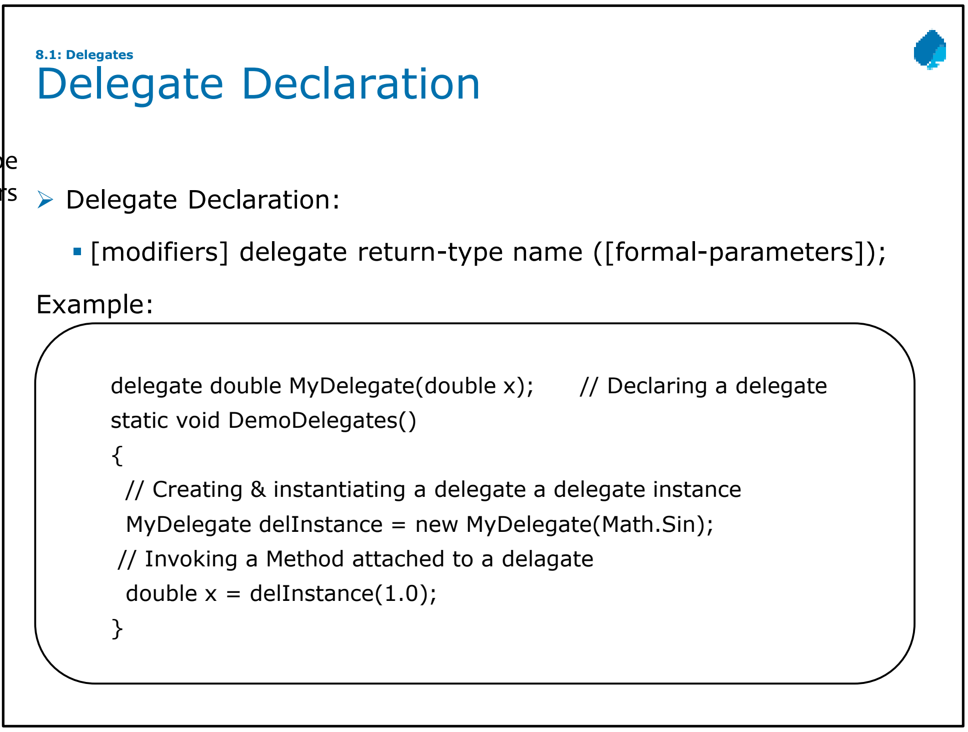
Delegate declaration

A delegate declaration defines a reference type that can be used to encapsulate a method with a specific signature. A delegate instance encapsulates a static or an instance method. Delegates are roughly similar to function pointers in C++; however, delegates are type-safe and secure.

The delegate declaration takes the form:
[attributes] [modifiers] delegate result-type identifier ([formal-parameters]);
where:
attributes (Optional): Additional declarative information. For more information on attributes and attribute classes see attributes chapter.
modifiers (Optional): The allowed modifiers are new, public, protected, internal and private.
result-type: The result type, which matches the return type of the method.
identifier: The delegate name.
formal-parameters (Optional): Parameter list. If a parameter is a pointer, the delegate must be declared with the unsafe modifier.

Delegates are the basis for events.

Explain the return-type and formal-parameters concept in detail



8.1: Delegates

Delegate Declaration

- Delegate Declaration:
  - [modifiers] delegate return-type name ([formal-parameters]);

Example:

```
delegate double MyDelegate(double x);      // Declaring a delegate
static void DemoDelegates()
{
  // Creating & instantiating a delegate a delegate instance
  MyDelegate delInstance = new MyDelegate(Math.Sin);
 // Invoking a Method attached to a delagate
  double x = delInstance(1.0);
}
```

A delegate is declared using the keyword delegate.

Here, ret-type is the type of value returned by the methods that the delegate will be calling. The name of the delegate is specified by name. The parameters required by the methods called through the delegate are specified in the parameter-list. A delegate can call only methods whose return type and parameter list match those specified by the delegate's declaration.

A delegate can invoke either an instance method associated with an object or a static method associated with a class. All that matters is that the return type and signature of the method agree with that of the delegate.

Explain the return-type and formal-parameters concept in detail

You can instantiate the delegate object to represent a specific method on an object or class.

When you have defined a delegate type, you can create an instance of the delegate type, and bind the delegate instance to a specific method, and then use the delegate to invoke the method indirectly when you are ready. To instantiate a delegate object, use the new operator to create an instance of a delegate type.

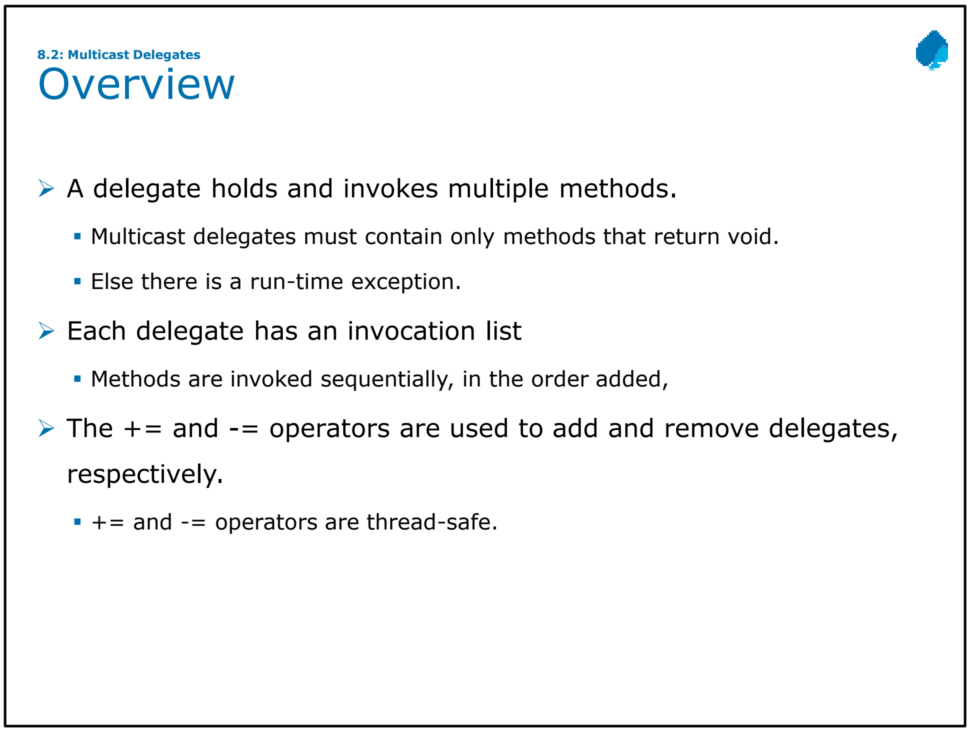
Pass a single parameter into the constructor, to specify the method that you want the delegate to represent. Use the syntax objectname. method-name or class-name.method-name to represent the method.

A delegate can invoke either an instance method associated with an object or a static method associated with a class. All that matters is that the return type and signature of the method agree with that of the delegate.

You can invoke a delegate synchronously or asynchronously. When you invoke a delegate synchronously the program waits until the method invoked by the delegate has been completed before it continues execution. When you invoke a delegate asynchronously, you specify a method to call after the delegate is completed, the code after the delegate invocation continues immediately after the delegate is invoked and does not wait for the method invoked by the delegate to complete.

You can define a Calculator class with Add, Modify methods with 2 int parameters. Create a delegate and call the methods using delegate.

Demo Using Delegates



8.2: Multicast Delegates

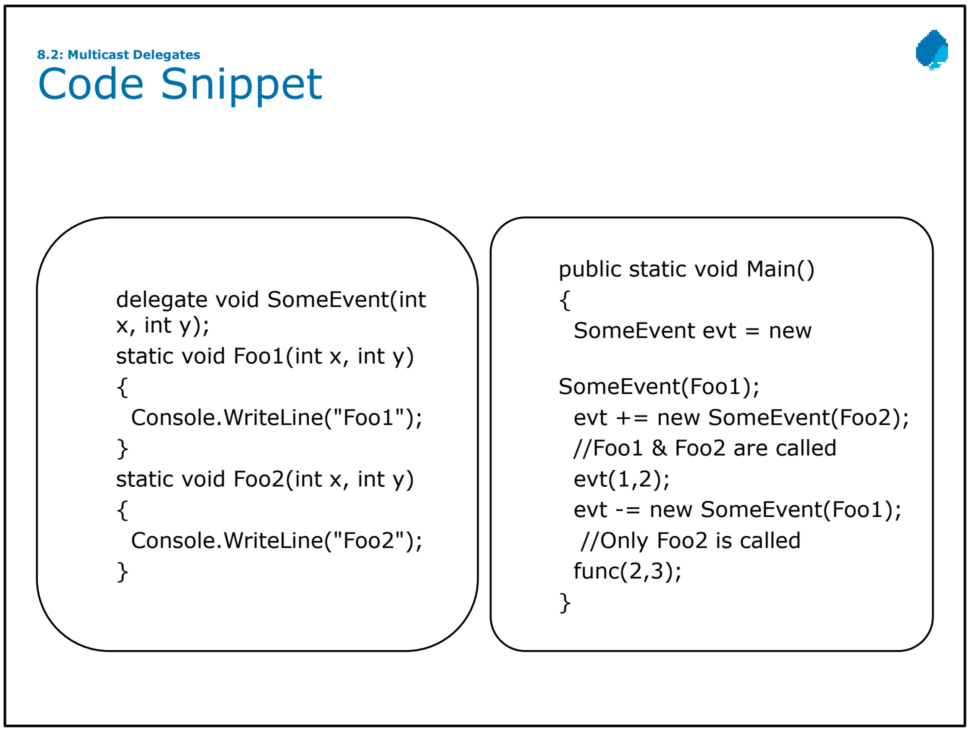
Overview

- A delegate holds and invokes multiple methods.
  - Multicast delegates must contain only methods that return void.
  - Else there is a run-time exception.
- Each delegate has an invocation list
  - Methods are invoked sequentially, in the order added,
- The += and -= operators are used to add and remove delegates, respectively.
  - += and -= operators are thread-safe.

If we want to call more than one method, we need to make an explicit call through a delegate more than once. However, it is possible for a delegate to wrap more than one method. Such a delegate is known as a **multicast delegate**.

One of the most exciting features of a delegate is its support for multicasting. In simple terms, multicasting is the ability to create an invocation list, or chain, of methods that will be automatically called when a delegate is invoked. Such a chain is very easy to create. Simply instantiate a delegate, and then use the + or += operator to add methods to the chain. To remove a method, use − or − =. If the delegate returns a value, then the value returned by the last method in the list becomes the return value of the entire delegate invocation. Thus, a delegate that will make use of multicasting will usually have a void return type.

A multicast delegate is a class derived from System.MulticastDelegate, which in turn is derived from System.Delegate. System.MulticastDelegate has additional members to allow chaining of method calls together into a list.



8.2: Multicast Delegates

Code Snippet

```
delegate void SomeEvent(int
x, int y);
static void Foo1(int x, int y)
{
  Console.WriteLine("Foo1");
}
static void Foo2(int x, int y)
{
  Console.WriteLine("Foo2");
}
```

```
public static void Main()
{
  SomeEvent evt = new

SomeEvent(Foo1);
  evt += new SomeEvent(Foo2);
  //Foo1 & Foo2 are called
  evt(1,2);
  evt -= new SomeEvent(Foo1);
   //Only Foo2 is called
  func(2,3);
}
```

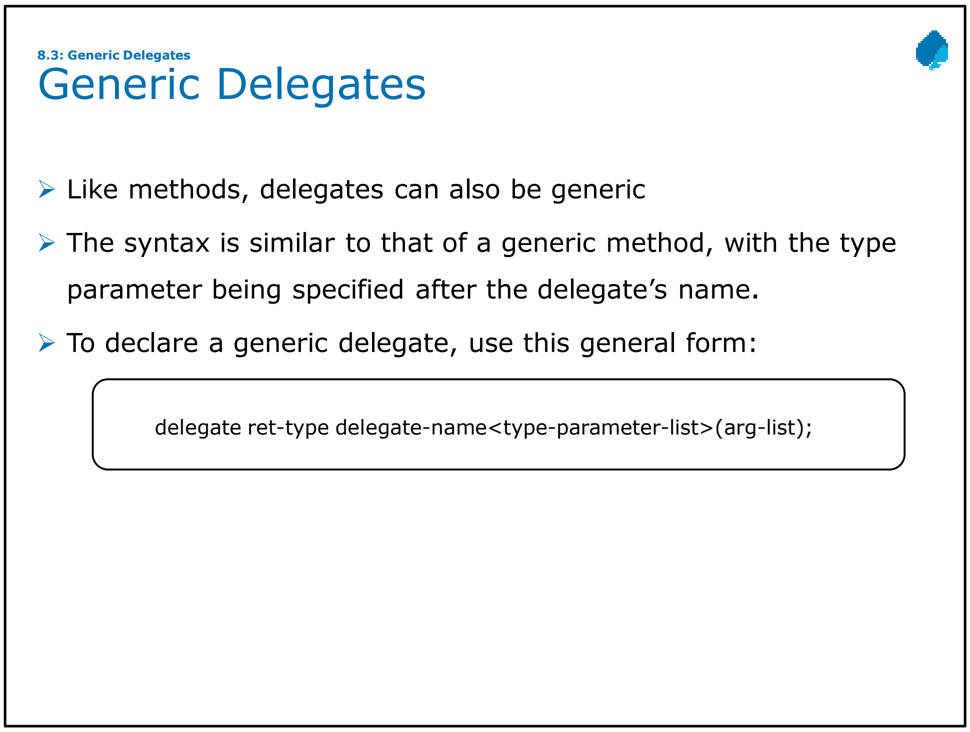
Here, we simply add both operations into the same multicast delegate. Multicast delegates recognize the operators + and +=. If we'd preferred, we could have expanded out the last two lines of the above code to this, which has the same effect:

```
SomeEvent evt1 = new SomeEvent(Foo1);
SomeEvent evt2 = new SomeEvent(Foo2);
SomeEvent evt = evt1 + evt2;
```

Multicast delegates also recognize the operators - and -= to remove method calls from the delegate.

Demo Multicast Delegates

Notice the placement of the type parameter list. It immediately follows the delegate's name.



8.3: Generic Delegates

Generic Delegates

- Like methods, delegates can also be generic
- The syntax is similar to that of a generic method, with the type parameter being specified after the delegate's name.
- To declare a generic delegate, use this general form:

```
delegate ret-type delegate-name<type-parameter-list>(arg-list);
```

Generic Delegates:
The advantage of generic delegates is that they let you define, in a type-safe manner, a generalized form that can then be matched to any specific type of method.

Generic Delegates (Cont..)

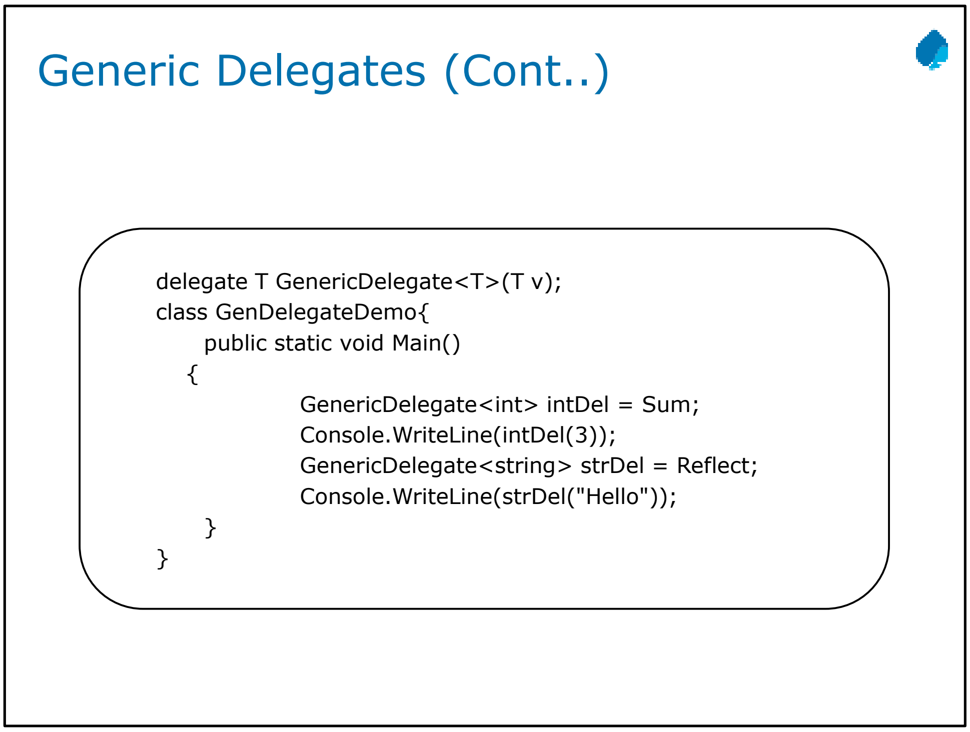
```
class GenDelegateDemo
{    static int Sum(int Value)
     {
         int result = 0;
         for(int i=Value; i>0; i--)
         result += i;
        return result;
     }
   static string Reflect(string str)
     {
          string result = "";
          foreach(char ch in str)
          result = ch + result;
           return result;
     }
}
```

Generic Delegates (Cont..)
A delegate can define its own type parameters. Code that references the generic delegate can specify the type argument to create a closed constructed type, just like when instantiating a generic class or calling a generic method, as shown in the following example:

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }
 Del<int> m1 = new Del<int>(Notify);
```

C# version 2.0 has a new feature called method group conversion, which applies to concrete as well as generic delegate types, and enables you to write the previous line with this simplified syntax:

```
Del<int> m2 = Notify;
```



Generic Delegates (Cont..)

```
delegate T GenericDelegate<T>(T v);
class GenDelegateDemo{
     public static void Main()
   {
                GenericDelegate<int> intDel = Sum;
                Console.WriteLine(intDel(3));
                GenericDelegate<string> strDel = Reflect;
                Console.WriteLine(strDel("Hello"));
     }
}
```

Explanation of the Example in slide:

First, notice how the GenericDelegate delegate is declared:

delegate T GenericDelegate <T>(T v);

Notice that T can be used as the return type even though the type parameter T is specified after the name GenericDelegate .

Inside GenDelegateDemo, the methods Sum( ) and Reflect( ) are declared, as shown here:

static int Sum(int v) {…} static string Reflect(string str) {…}

The Sum( ) method returns the summation of the integer value passed as an argument. The Reflect( ) method returns a string that is the reverse of the string passed as an argument.

Inside Main( ), a delegate called intDel is instantiated and assigned a reference to Sum( ):

GenericDelegate <int> intDel = sum;

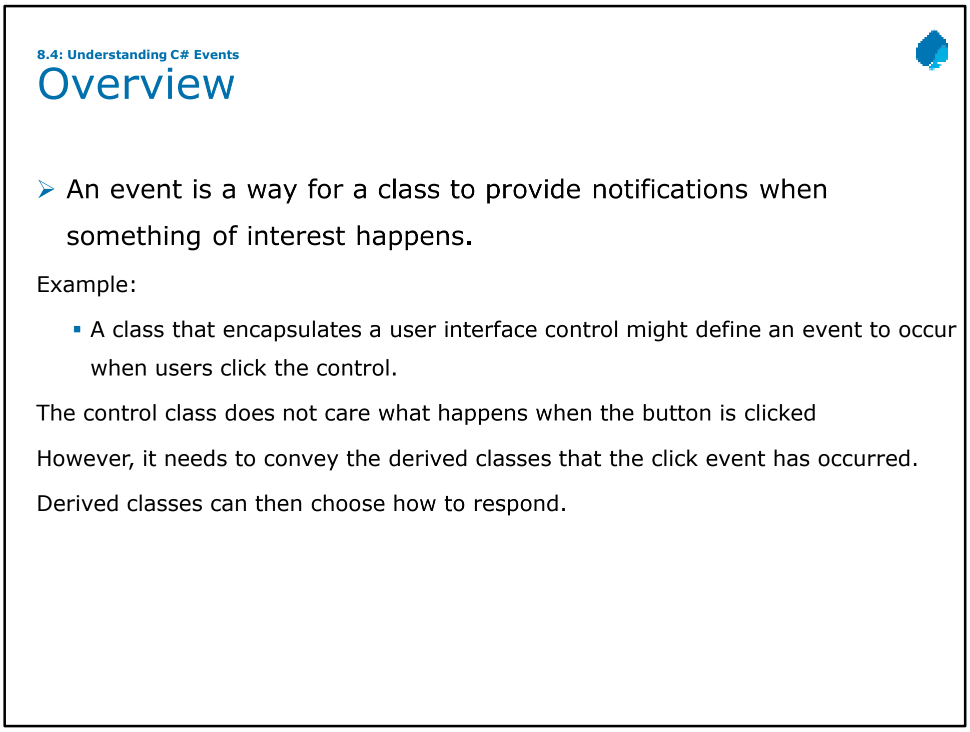
Because Sum( ) takes an int argument and returns an int value, Sum( ) is compatible with an int instance of GenericDelegate .

In similar fashion, the delegate strDel is created and assigned a reference to Reflect( ):

SomeOp<string> strDel = Reflect;

Because Reflect() takes a string argument and returns a string result, it is compatible with the string version of GenericDelegate .

Demo Generic Delegates


8.4: Understanding C# Events

Overview

- An event is a way for a class to provide notifications when something of interest happens.

Example:

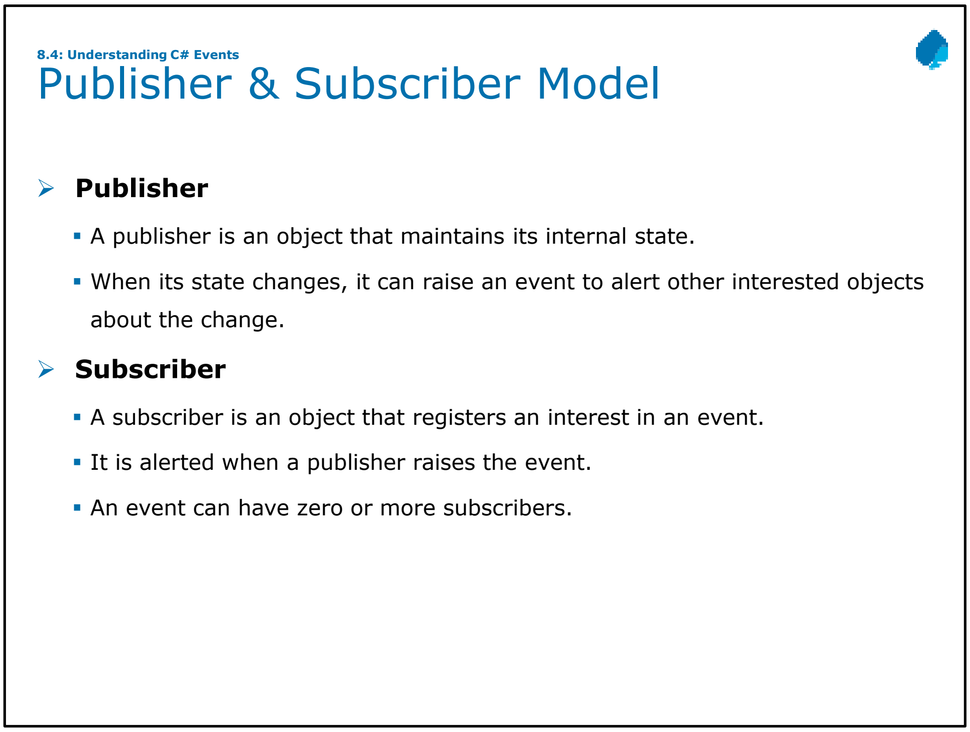
- A class that encapsulates a user interface control might define an event to occur when users click the control.

The control class does not care what happens when the button is clicked

However, it needs to convey the derived classes that the click event has occurred.

Derived classes can then choose how to respond.

Events allow an object to notify other objects that a change has occurred. The other objects can register an interest in an event, and they will be notified when the event occurs. In other words, events allow objects to register that they need to be notified about changes to other objects.



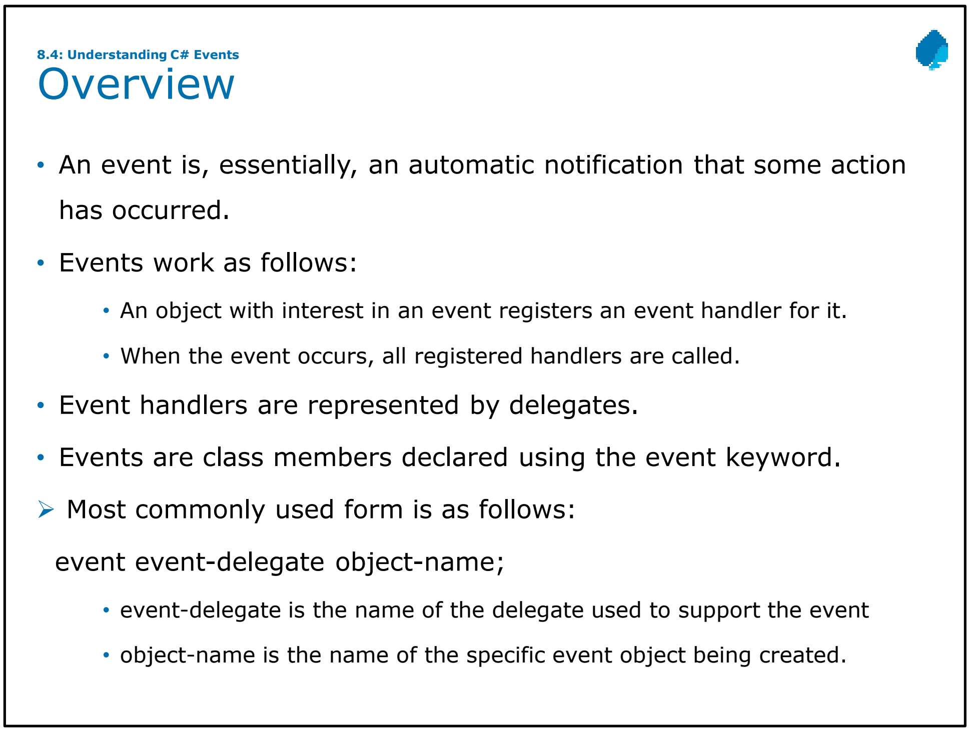
8.4: Understanding C# Events

Publisher & Subscriber Model

- **Publisher**
  - A publisher is an object that maintains its internal state.
  - When its state changes, it can raise an event to alert other interested objects about the change.
- **Subscriber**
  - A subscriber is an object that registers an interest in an event.
  - It is alerted when a publisher raises the event.
  - An event can have zero or more subscribers.

When something of importance takes place in a class, the class raises an event. The class raising the event is like a publisher. One or more other classes can subscribe to the event with event handlers. The event handlers execute some predefined logic when the event is raised. If no-one is subscribed to the event, the event is ignored and nothing happens.

For example, in the windows user interface you can add a button, the button class raises a click event when someone clicks on the button. If you do not create an event handler for the button click event, nothing happens when the user clicks on the button. If you create an event handler for the button click event in your form class you can execute the desired code such as saving a record, or displaying another form when the button is clicked. You can have multiple subscribers to an event as well. When you have multiple subscribers then the event handler defined in each subscriber will execute.



8.4: Understanding C# Events

Overview

- An event is, essentially, an automatic notification that some action has occurred.
- Events work as follows:
  - An object with interest in an event registers an event handler for it.
  - When the event occurs, all registered handlers are called.
- Event handlers are represented by delegates.
- Events are class members declared using the event keyword.

➢ Most commonly used form is as follows:

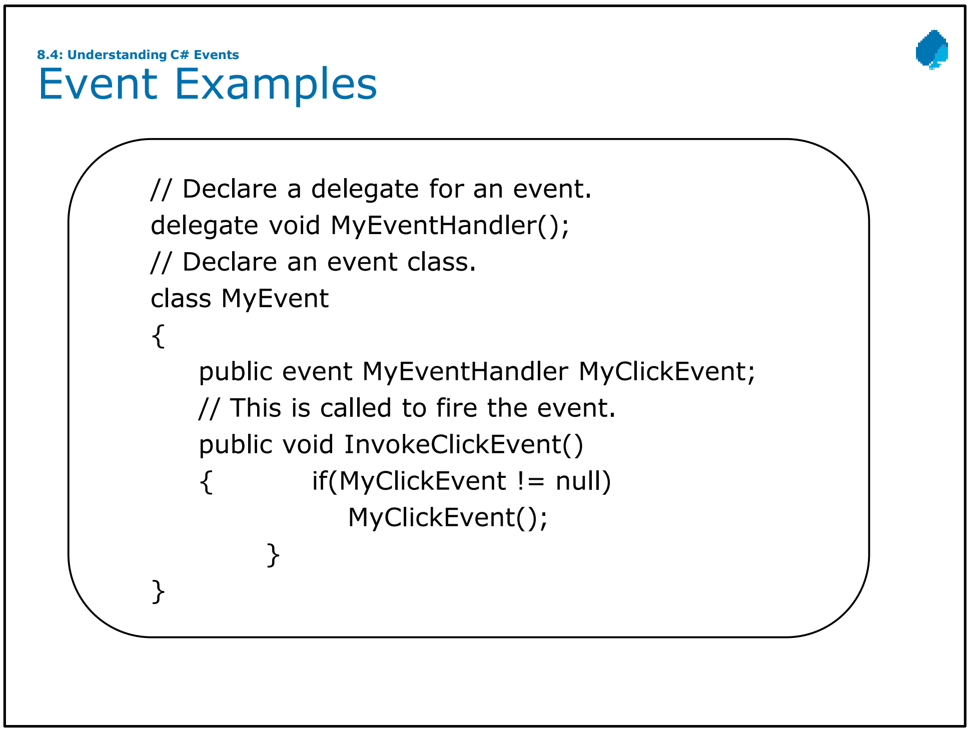
event event-delegate object-name;

  - event-delegate is the name of the delegate used to support the event
  - object-name is the name of the specific event object being created.

Events are the usual means by which an application running on the Windows platform can receive notifications when something interesting happens.

In C# terms, events are actually a special form of delegates.

Here, event-delegate is the name of the delegate used to support the event, and object-name is the name of the specific event object being created.



8.4: Understanding C# Events

Event Examples

```
// Declare a delegate for an event.
delegate void MyEventHandler();
// Declare an event class.
class MyEvent
{
    public event MyEventHandler MyClickEvent;
    // This is called to fire the event.
    public void InvokeClickEvent()
    {         if(MyClickEvent != null)
                 MyClickEvent();
          }
}
```

The program begins by declaring a delegate for the event handler, as shown here:

delegate void MyEventHandler();

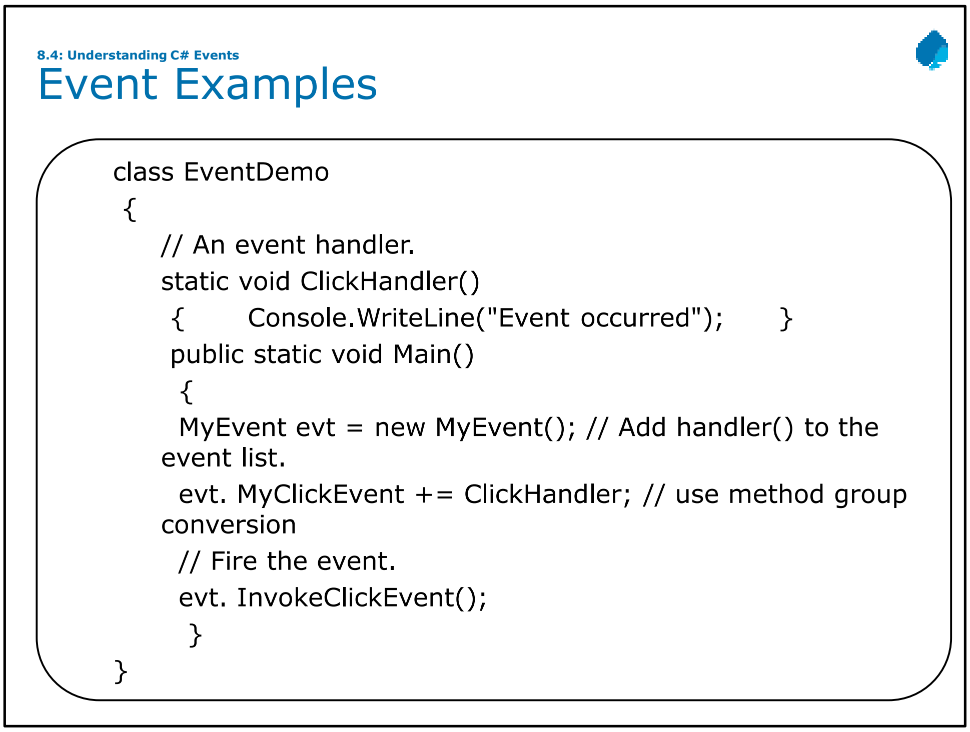
All events are activated through a delegate. Thus, the event delegate defines the return type and signature for the event. In this case, there are no parameters, but event parameters are allowed. Because events are commonly multicast, an event will normally return void.

Next, an event class, called MyEvent, is created. Inside the class, an event object called MyClickEvent is declared, using this line:
public event MyEventHandler MyClickEvent ;

The keyword event tells the compiler that an event is being declared.

Also declared inside MyEvent is the method InvokeClickEvent( ), which is the method that a program will call to signal (or “fire”) an event. (That is, this is the method called when the event occurs.) It calls an event handler through the

SomeEvent delegate, as shown here:
if(MyClickEvent != null) MyClickEvent();



Event Examples

```
class EventDemo
 {
     // An event handler.
     static void ClickHandler()
      {       Console.WriteLine("Event occurred");      }
      public static void Main()
       {
       MyEvent evt = new MyEvent(); // Add handler() to the
     event list.
       evt. MyClickEvent += ClickHandler; // use method group
     conversion
       // Fire the event.
       evt. InvokeClickEvent();
        }
}
```

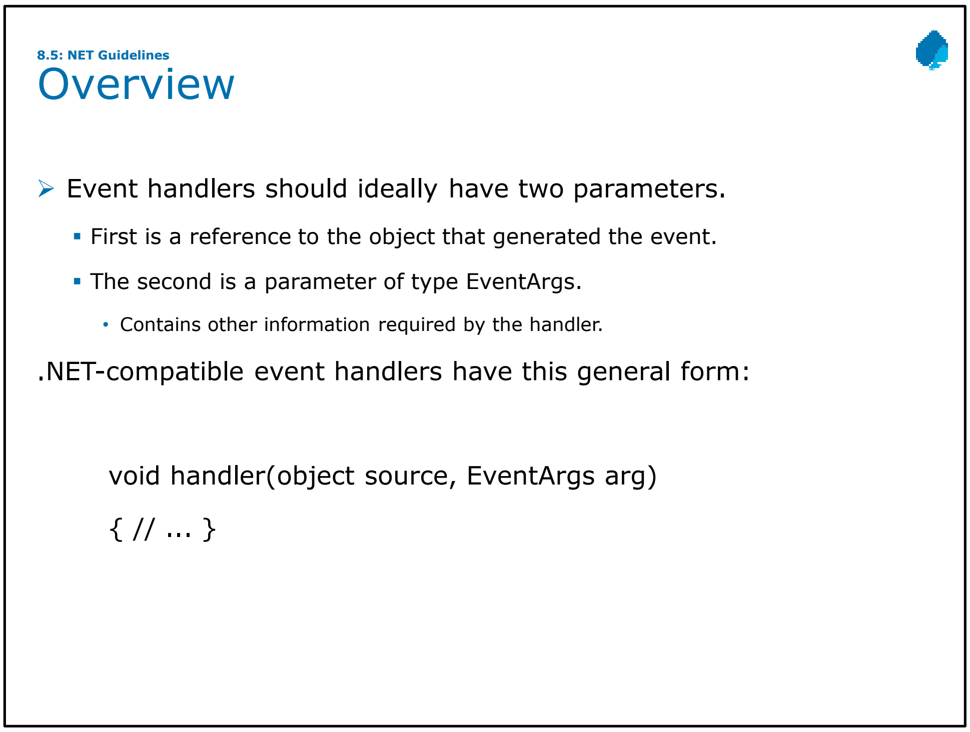
Inside EventDemo, an event handler called ClickHandler() is created.
In this simple example, the event handler just displays a message, but other handlers could perform more meaningful actions.
In Main( ), a MyEvent object is created, and ClickHandler() is registered as a handler for this event by adding it as shown here:
MyEvent evt = new MyEvent(); // Add handler() to the event list.
evt.MyClickEvent += ClickHandler; // use method group conversion

Notice that the handler is added using the += operator. Events support only += and – =. In this case, ClickHandler() is a static method, but event handlers can also be instance methods. Notice that the new method group conversion syntax can be used with delegates that support events. In the past, the handler would have been added by this line:

evt.MyClickEvent += new MyEventHandler(ClickHandler); // old style

Finally, the event is fired as shown here:
// Fire the event.
evt.OnClickEvent();

Calling InvokeClickEvent( ) causes all registered event handlers to be called. In this case, there is only one registered handler, but there could be more.


8.5: NET Guidelines

Overview

- Event handlers should ideally have two parameters.
  - First is a reference to the object that generated the event.
  - The second is a parameter of type EventArgs.
    - Contains other information required by the handler.

.NET-compatible event handlers have this general form:

```
void handler(object source, EventArgs arg)
{ // ... }
```

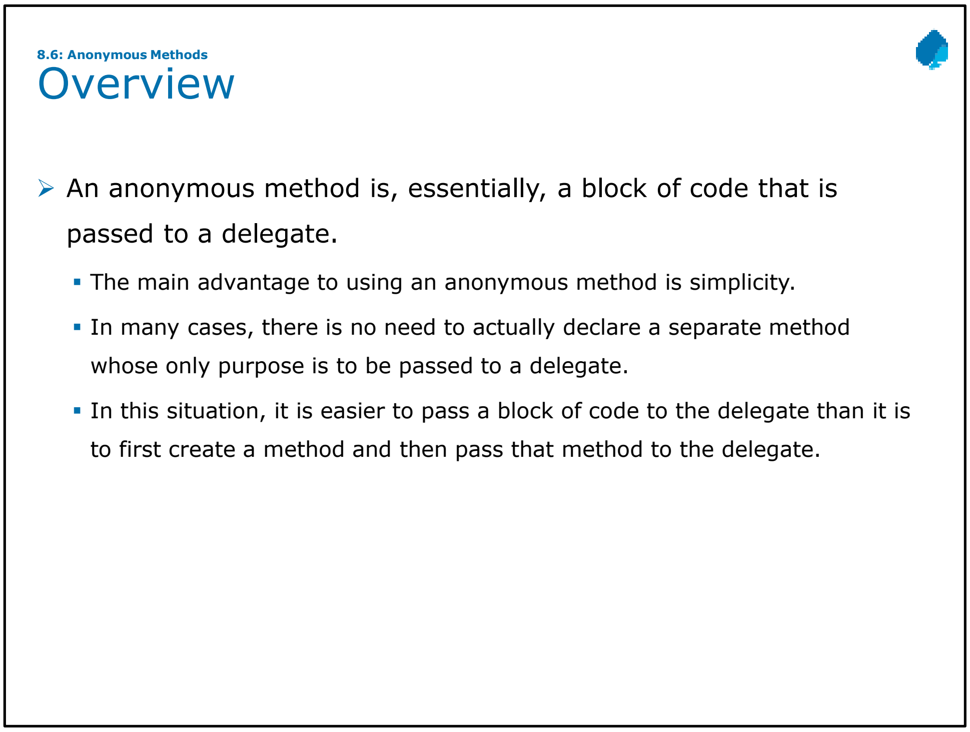
C# allows you to write any type of event that you desire. However, for component compatibility with the .NET Framework, you will need to follow the guidelines that Microsoft has established for this purpose. At the core of these guidelines is the requirement that event handlers have two parameters. The first is a reference to the object that generated the event. The second is a parameter of type EventArgs that contains any other information required by the handler. Thus, .NET-compatible event handlers will have this general form:

```
void handler(object source, EventArgs arg) {
 // ...
}
```

Typically, the source parameter is passed this by the calling code. The EventArgs parameter contains additional information and can be ignored if it is not needed.

The EventArgs class itself does not contain fields in which you pass additional data to a handler. Instead, EventArgs is used as a base class from which you will derive a class that contains the necessary fields. EventArgs does include one static field called Empty, which obtains an EventArgs object that contains no data.

Demo Events in .NET



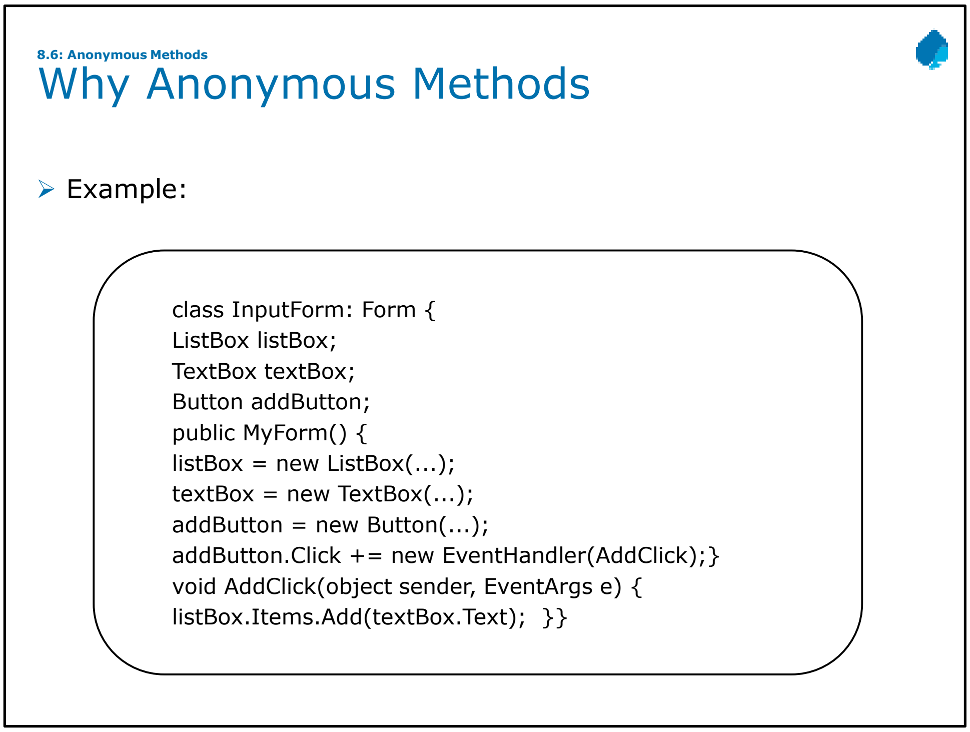
8.6: Anonymous Methods

Overview

- An anonymous method is, essentially, a block of code that is passed to a delegate.
    - The main advantage to using an anonymous method is simplicity.
    - In many cases, there is no need to actually declare a separate method whose only purpose is to be passed to a delegate.
    - In this situation, it is easier to pass a block of code to the delegate than it is to first create a method and then pass that method to the delegate.

Anonymous methods allow you to create an instance of a delegate by specifying a block of inline code to be invoked by the delegate rather than specifying an existing method to be invoked by the delegate. This allows you to conserve code because you do not need to create a new method every time a delegate is passed as a parameter to a method call. Anonymous methods are not supported in Visual Basic.

Anonymous methods are especially useful when working with events, because an anonymous method can serve as an event handler. This eliminates the need to declare a separate method, which can significantly streamline event-handling code. Anonymous methods are a new feature added by C# 2.0.



8.6: Anonymous Methods

Why Anonymous Methods

➢ Example:

```
class InputForm: Form {
ListBox listBox;
TextBox textBox;
Button addButton;
public MyForm() {
listBox = new ListBox(...);
textBox = new TextBox(...);
addButton = new Button(...);
addButton.Click += new EventHandler(AddClick);}
void AddClick(object sender, EventArgs e) {
listBox.Items.Add(textBox.Text);  }}
```

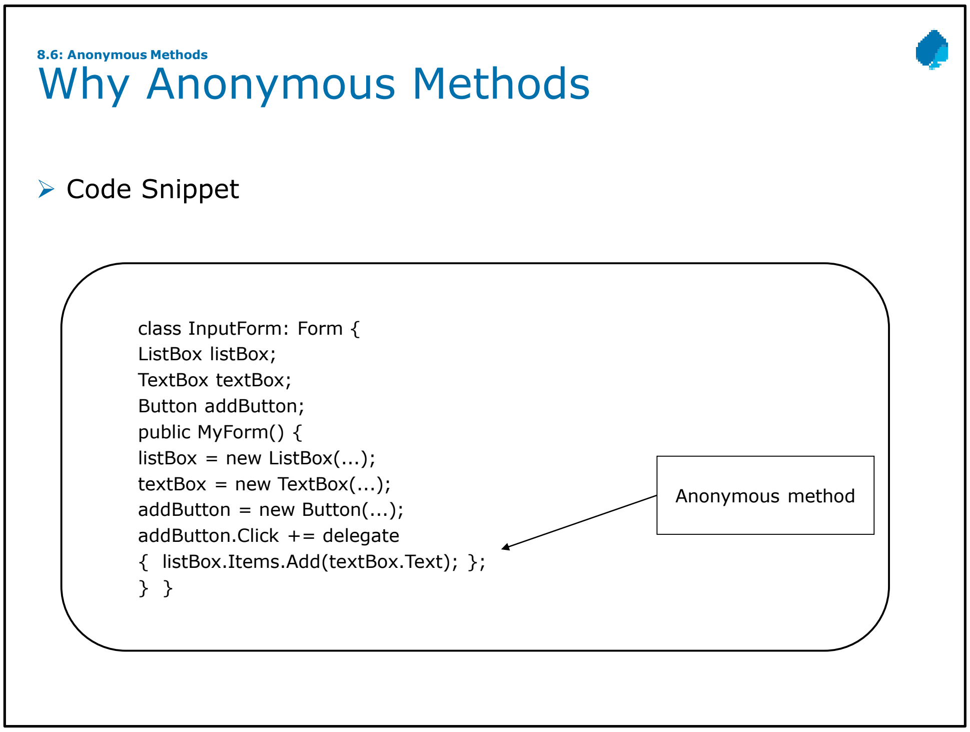
Event handlers and other callbacks are often invoked exclusively through delegates and never directly. Even so, it has thus far been necessary to place the code of event handlers and callbacks in distinct methods to which delegates are explicitly created. In contrast, anonymous methods allow the code associated with a delegate to be written “in-line” where the delegate is used, conveniently tying the code directly to the delegate instance. Besides this convenience, anonymous methods have shared access to the local state of the containing function member. To achieve the same state sharing using named methods requires “lifting” local variables into fields in instances of manually authored helper classes.

The above example shows a simple input form that contains a list box, a text box, and a button. When the button is clicked, an item containing the text in the text box is added to the list box.

Why Anonymous Methods

- Even though only a single statement is executed in response to the button’s Click event, that statement must be extracted into a separate method with a full parameter list, and an EventHandler delegate referencing that method must be manually created.

- Using an anonymous method, the event handling code becomes significantly more succinct.



8.6: Anonymous Methods

Why Anonymous Methods

- Code Snippet

```
class InputForm: Form {
ListBox listBox;
TextBox textBox;
Button addButton;
public MyForm() {
listBox = new ListBox(...);
textBox = new TextBox(...);
addButton = new Button(...);
addButton.Click += delegate
{  listBox.Items.Add(textBox.Text); };
} }
```

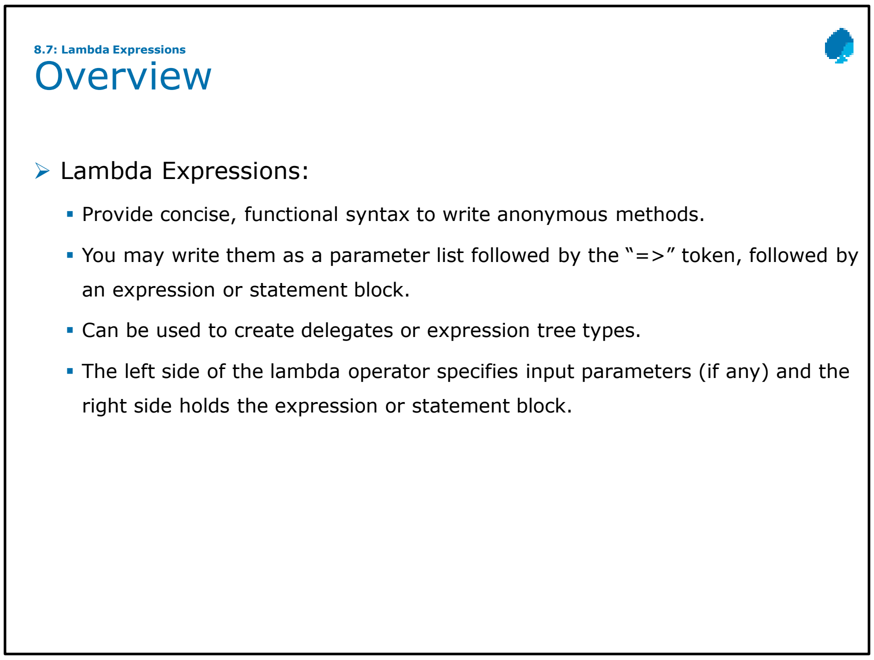
Anonymous method

An anonymous method consists of the keyword delegate, an optional parameter list, and a statement list enclosed in { and } delimiters. The anonymous method in the previous example does not use the parameters supplied by the delegate, and it can therefore omit the parameter list. To gain access to the parameters, the anonymous method can include a parameter list.

Following example uses anonymous methods to write functions "in-line." The anonymous methods are passed as parameters of a Function delegate type.

```
using System;
delegate double Function(double x);
class Test
{
        static double[] Apply(double[] a, Function f)
        {
                double[] result = new double[a.Length];
                for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
                return result;
        }
        static double[] MultiplyAllBy(double[] a, double factor)
        {
                 return Apply(a, delegate(double x) { return x *
factor; });
        }
        static void Main()
        {
                double[] a = {0.0, 0.5, 1.0};
                double[] squares = Apply(a, delegate(double x) {
                return x * x; });
                double[] doubles = MultiplyAllBy(a, 2.0);
        }
}
```

Demo Anonymous Method



8.7: Lambda Expressions

Overview

- Lambda Expressions:
  - Provide concise, functional syntax to write anonymous methods.
  - You may write them as a parameter list followed by the "=>" token, followed by an expression or statement block.
  - Can be used to create delegates or expression tree types.
  - The left side of the lambda operator specifies input parameters (if any) and the right side holds the expression or statement block.

As you can see, you don't have to explicitly declare a new method to link it with an event. You can use anonymous methods to achieve the same thing in C# 2.0.

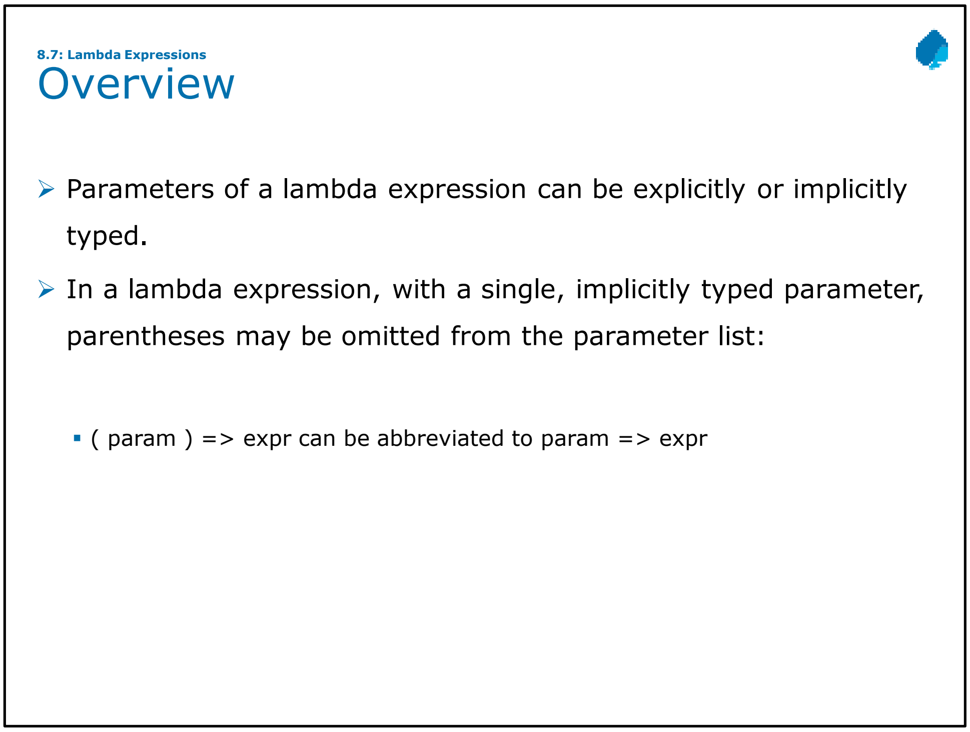
C# 3.0 introduces an even simpler syntax, lambda expressions, which you write as a parameter list followed by the "=>" token, followed by an expression or a statement block. Lambda expressions provide a more concise, functional syntax for writing anonymous methods.

A lambda expression is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.

All lambda expressions use the lambda operator =>, which is read as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression or statement block. The lambda expression x => x * x is read "x goes to x times x." This expression can be assigned to a delegate type as follows:

```
delegate int del(int i);
del myDelegate = x => x * x;
```

```
int j = myDelegate(5); //j = 25
```



8.7: Lambda Expressions

Overview

- Parameters of a lambda expression can be explicitly or implicitly typed.
- In a lambda expression, with a single, implicitly typed parameter, parentheses may be omitted from the parameter list:
  - (param) => expr can be abbreviated to param => expr

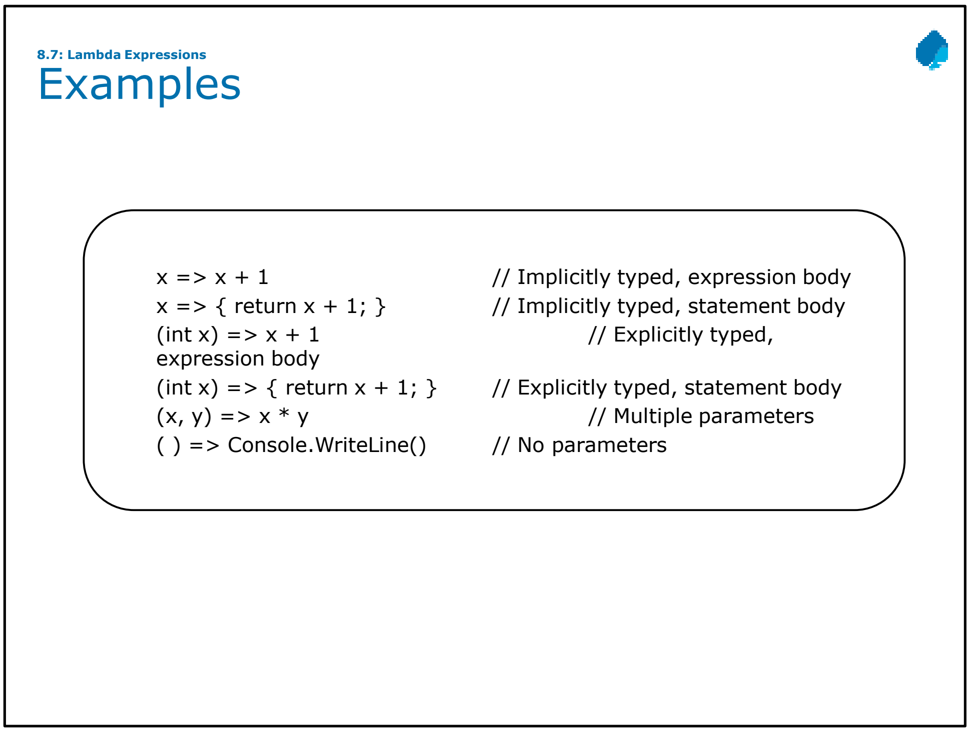
Parameters to Lambda Expressions

The parameters of a lambda expression can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each expression is explicitly specified. In an implicitly typed parameter list, the types are inferred from the context in which the lambda expression occurs:
In a lambda expression with a single, implicitly typed parameter, the parentheses may be omitted from the parameter list. In other words, a lambda expression of the form:

```
( param ) => expr
```

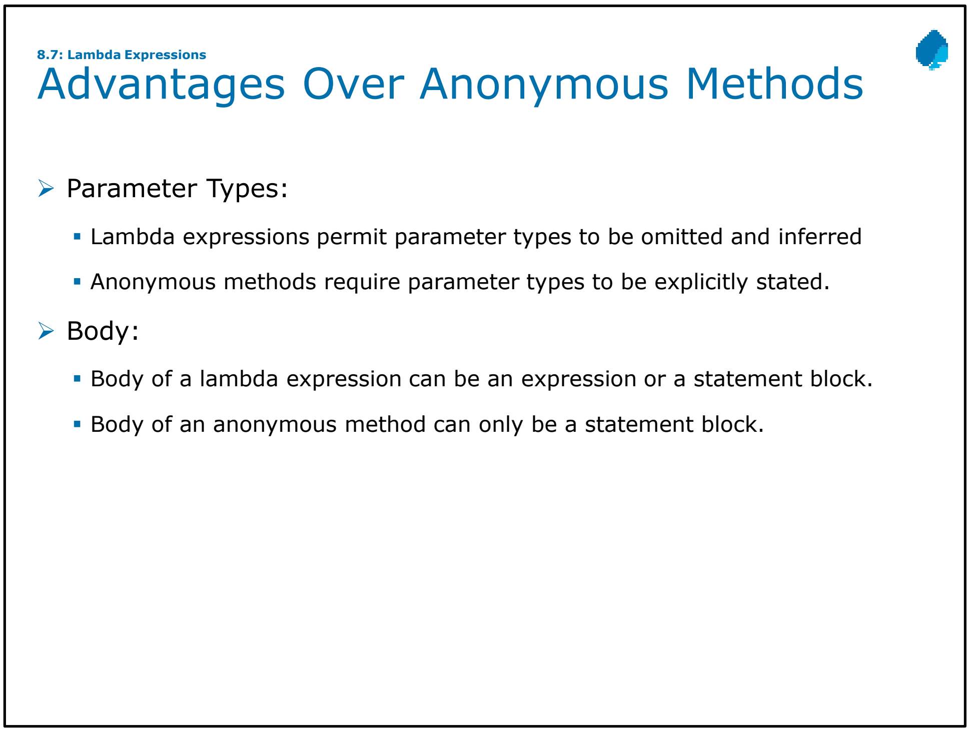
The above can be abbreviated to:

```
param => expr
```



Examples

```
x => x + 1                          // Implicitly typed, expression body
x => { return x + 1; }              // Implicitly typed, statement body
(int x) => x + 1                             // Explicitly typed,
expression body
(int x) => { return x + 1; }        // Explicitly typed, statement body
(x, y) => x * y                              // Multiple parameters
( ) => Console.WriteLine()          // No parameters
```



8.7: Lambda Expressions

Advantages Over Anonymous Methods

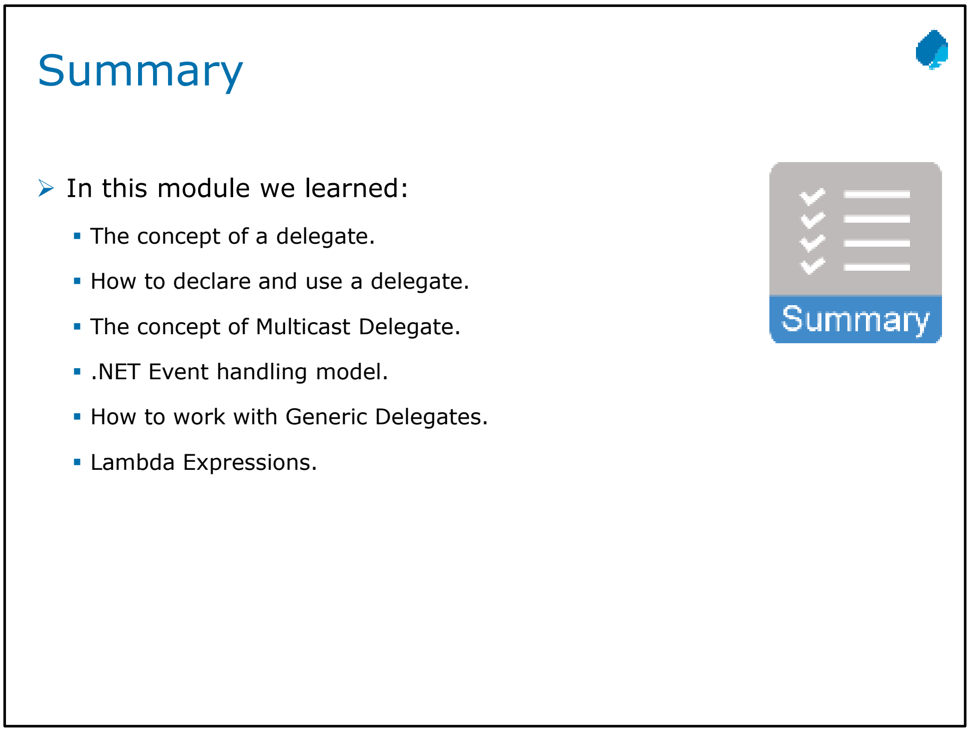
- Parameter Types:
  - Lambda expressions permit parameter types to be omitted and inferred
  - Anonymous methods require parameter types to be explicitly stated.
- Body:
  - Body of a lambda expression can be an expression or a statement block.
  - Body of an anonymous method can only be a statement block.

Lambda expressions are a functional superset of anonymous methods.
These provide the following additional functionality:
Lambda expressions permit parameter types to be omitted and inferred whereas anonymous methods require parameter types to be explicitly stated.
The body of a lambda expression can be an expression or a statement block whereas the body of an anonymous method can only be a statement block.
Lambda expressions passed as arguments participate in type argument inference and in method overload resolution.
Lambda expressions with an expression body can be converted to expression trees.

Arguments & Expression Trees

- Lambda expressions passed as arguments participate in type argument inference and in method overload resolution.

- Lambda expressions with an expression body can be converted to expression trees.

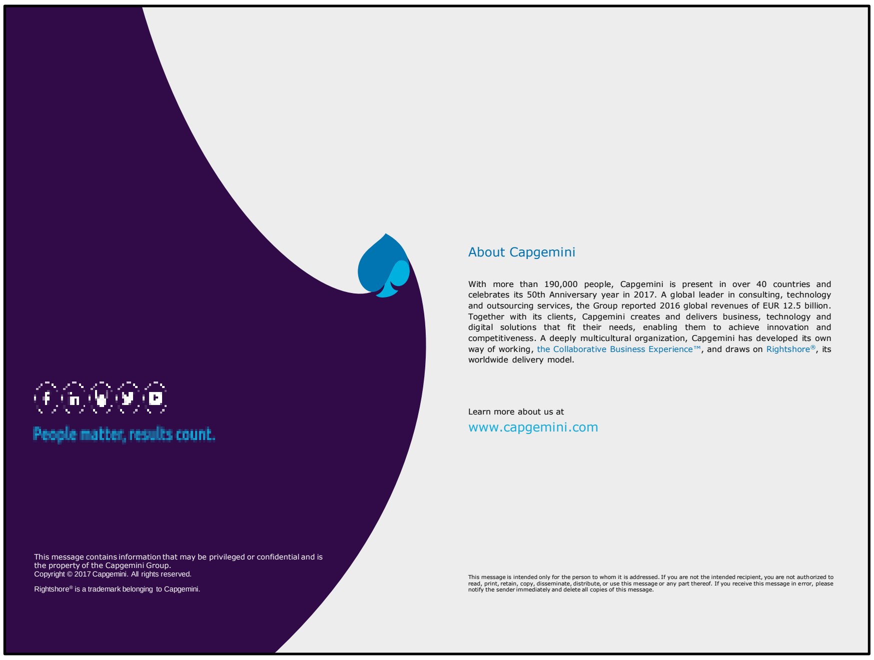


Summary

- In this module we learned:
  - The concept of a delegate.
  - How to declare and use a delegate.
  - The concept of Multicast Delegate.
  - .NET Event handling model.
  - How to work with Generic Delegates.
  - Lambda Expressions.

Review Question

- 1. What is a delegate?
- 2. What is Multicasting of Delegate?
- 3. In what scenario you will be using Multicast Delegate?
- 4. What are the advantages of Lambda expression over Anonymous methods?



People matter, results count.

About Capgemini

With more than 190,000 people, Capgemini is present in over 40 countries and celebrates its 50th Anniversary year in 2017. A global leader in consulting, technology and outsourcing services, the Group reported 2016 global revenues of EUR 12.5 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

Learn more about us at

www.capgemini.com