

Delegate, Event, Anonymous



Lesson Objectives

- Explain delegates
- Explain events
- Explain Anonymous Method, Anonymous Type

Section 1

DELEGATES

In the .NET Framework, a delegate points to one or more methods. Once you instantiate the delegate, the corresponding methods invoke.

Delegates are objects that contain references to methods that need to be invoked instead of containing the actual method names.

Using delegates, you can call any method, which is identified only at run-time.

A delegate is like having a general method name that points to various methods at different times and invokes the required method at run-time.

In C#, invoking a delegate will execute the referenced method at run-time.

To associate a delegate with a particular method, the method must have the same return type and parameter type as that of the delegate.

Declaring Delegates 1-2

- Delegates in C# are declared using the `delegate` keyword followed by the return type and the parameters of the referenced method.
- Declaring a delegate is quite similar to declaring a method except that there is no implementation. Thus, the declaration statement must end with a semi-colon.
- The following figure displays an example of declaring delegates:

Valid Delegate Declaration

```
public delegate int Calculation(int numOne, int numTwo);
```



Invalid Delegate Declaration

```
public delegate Calculation(int numOne, int numTwo)
{
}
```



Declaring Delegates 2-2

- The following syntax is used to declare a delegate:

Syntax

```
<access_modifier> delegate <return_type> DelegateName([list_of_parameters]);
```

- where,
 - ✓ `access_modifier`: Specifies the scope of access for the delegate. If declared outside the class, the scope will always be public.
 - ✓ `return_type`: Specifies the data type of the value that is returned by the method.
 - ✓ `DelegateName`: Specifies the name of the delegate.
 - ✓ `list_of_parameters`: Specifies the data types and names of parameters to be passed to the method.
- The following code declares the delegate **Calculation** with the return type and the parameter types as integer:

Snippet

```
public delegate int Calculation(int numOne, int numTwo);
```

Instantiating Delegates 1-2

- The next step after declaring the delegate is to instantiate the delegate and associate it with the required method by creating an object of the delegate.
- Like all other objects, an object of a delegate is created using the `new` keyword.
- This object takes the name of the method as a parameter and this method has a signature similar to that of the delegate.
- The created object is used to invoke the associated method at run-time.
- The following figure displays an example of instantiating delegates:

```
Calculation objCalculation = new Calculation(Addition);
```

Delegate Name Object Name Referenced Method

Instantiating Delegates 2-2

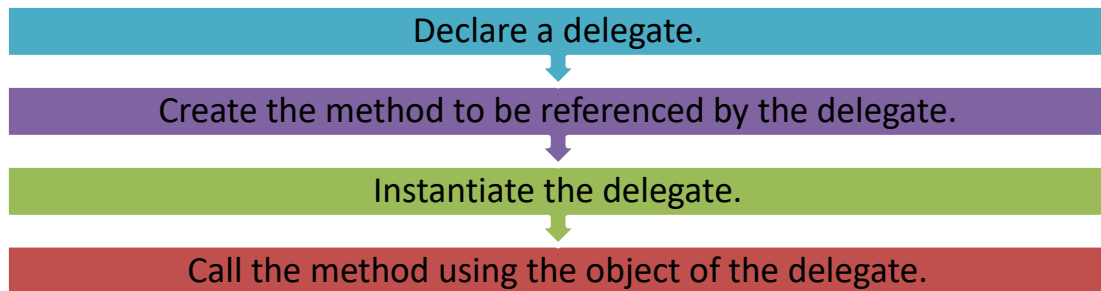
- The following syntax is used to instantiate a delegate:

Syntax

```
<DelegateName><objName> = new <DelegateName>(<MethodName>);
```

- where,
 - ✓ **DelegateName**: Specifies the name of the delegate.
 - ✓ **objName**: Specifies the name of the delegate object.
 - ✓ **MethodName**: Specifies the name of the method to be referenced by the delegate object.

- A delegate can be declared either before creating the class (having the method to be referenced) or can be defined within the class.
- The following are the four steps to implement delegates in C#:



```
class DelegatesDemo
{
    public delegate double Temperature(double temp);

    public static double FahrenheitToCelsius(double temp)
    {
        return ((temp-32) / 9)*5;
    }

    public static void Main()
    {
        temperature tempConversion = new temperature(FahrenheitToCelsius);

        double tempF = 96;

        double tempC = tempConversion(tempF);

        Console.WriteLine("Temperature in Fahrenheit = {0:F}".tempF);

        Console.WriteLine("Temperature in Celsius = {0:F}".tempC);
    }
}
```

Multiple Delegates 1-2

- In C#, a user can invoke multiple delegates within a single program. Depending on the delegate name or the type of parameters passed to the delegate, the appropriate delegate is invoked.
- The following code demonstrates the use of multiple delegates by creating two delegates **CalculateArea** and **CalculateVolume** that have their return types and parameter types as double:

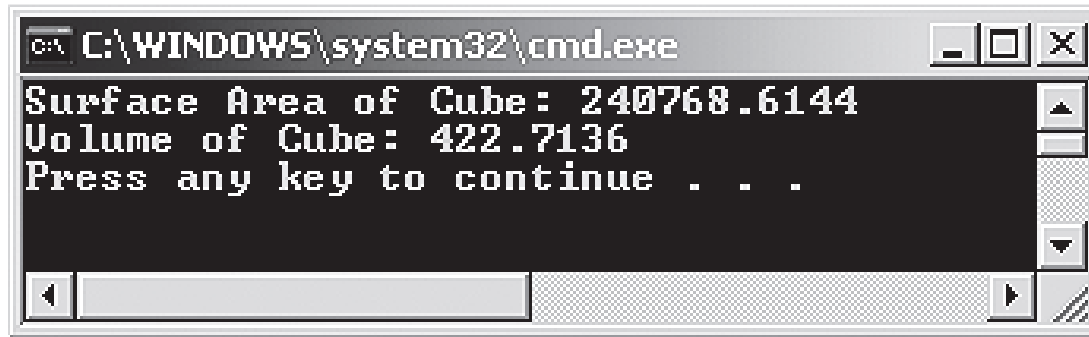
Snippet

```
using System;
public delegate double CalculateArea(double val);
public delegate double CalculateVolume(double val);

class Cube
{
    static double Area(double val)
    {
        return 6 * (val * val);
    }
    static double Volume(double val)
    {
        return (val * val);
    }
    static void Main(string[] args)
    {
        CalculateArea objCalculateArea = new CalculateArea(Area);
        CalculateVolume objCalculateVolume = new
        CalculateVolume(Volume);
        Console.WriteLine ("Surface Area of Cube: " +
        objCalculateArea(200.32));
        Console.WriteLine("Volume of Cube: " +
        objCalculateVolume(20.56));
    }
}
```

Multiple Delegates 2-2

- In the code:
 - ✓ When the delegates **CalculateArea** and **CalculateVolume** are instantiated in the Main() method, the references of the methods **Area** and **Volume** are passed as parameters to the delegates **CalculateArea** and **CalculateVolume** respectively.
 - ✓ The values are passed to the instances of appropriate delegates, which in turn invoke the respective methods.
- The following figure shows the use of multiple delegates:



A screenshot of a Windows command prompt window. The title bar reads "C:\WINDOWS\system32\cmd.exe". The window contains the following text:
Surface Area of Cube: 240768.6144
Volume of Cube: 422.7136
Press any key to continue . . .
The window has a scrollbar on the right and a status bar at the bottom.

Multicast Delegates 1-3

- A single delegate can encapsulate the references of multiple methods at a time to hold a number of method references.
- Such delegates are termed as 'Multicast Delegates' that maintain a list of methods (invocation list) that will be automatically called when the delegate is invoked.
- Multicast delegates in C# are sub-types of the `System.MulticastDelegate` class. Multicast delegates are defined in the same way as simple delegates, however, the return type of multicast delegates can only be void.
- If any other return type is specified, a run-time exception will occur because if the delegate returns a value, the return value of the last method in the invocation list of the delegate will become the return type of the delegate resulting in inappropriate results. Hence, the return type is always void.
- To add methods into the invocation list of a multicast delegate, the user can use the '+' or the '+=' assignment operator. Similarly, to remove a method from the delegate's invocation list, the user can use the '-' or the '-=' operator. When a multicast delegate is invoked, all the methods in the list are invoked sequentially in the same order in which they have been added.

Multicast Delegates 2-3

- The following code creates a multicast delegate **Maths**. This delegate encapsulates the reference to the methods **Addition**, **Subtraction**, **Multiplication**, and **Division**:

Snippet

```
using System;
public delegate void Maths (int valOne, int valTwo);

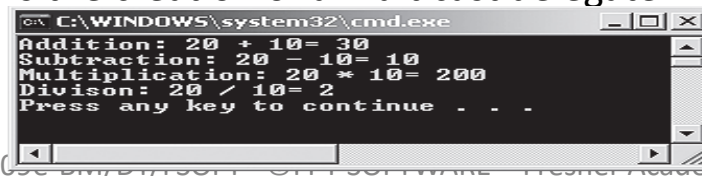
class MathsDemo
{
    static void Addition(int valOne, int valTwo)
    {
        int result = valOne + valTwo;
        Console.WriteLine("Addition: " + valOne + " + " +
            valTwo + "= " + result);
    }
    static void Subtraction(int valOne, int valTwo)
    {
        int result = valOne - valTwo;
        Console.WriteLine("Subtraction: " + valOne + " - " +
            valTwo + "= " + result);
    }
    static void Multiplication(int valOne, int valTwo)
    {
        int result = valOne * valTwo;
        Console.WriteLine("Multiplication: " + valOne + " * "
            + valTwo + "= " + result);
    }
}
```

Multicast Delegates 3-3

```
static void Division(int valOne, int valTwo)
{
    int result = valOne / valTwo;
    Console.WriteLine("Division: " + valOne + " / " +
        valTwo + "=" + result);
}

static void Main(string[] args)
{
    Maths objMaths = new Maths(Addition);
    objMaths += new Maths(Subtraction);
    objMaths += new Maths(Multiplication);
    objMaths += new Maths(Division);
    if (objMaths != null)
    {
        objMaths(20, 10);
    }
}
```

- In the code:
 - ✓ The delegate **Maths** is instantiated in the **Main()** method. Once the object is created, methods are added to it using the '+' assignment operator, which makes the delegate a multicast delegate.
- The following figure shows the creation of a multicast delegate:



System.Delegate Class 1-2

- The Delegate class of the System namespace is a built-in class defined to create delegates in C#.
- All delegates in C# implicitly inherit from the Delegate class. This is because the delegate keyword indicates to the compiler that the defined delegate in a program is to be derived from the Delegate class. The Delegate class provides various constructors, methods, and properties to create, manipulate, and retrieve delegates defined in a program.
- The following table lists the constructors defined in the Delegate class:

Constructor	Description
<code>Delegate(object, string)</code>	Calls a method referenced by the object of the class given as the parameter
<code>Delegate(type, string)</code>	Calls a static method of the class given as the parameter

- The following table lists the properties defined in the Delegate class:

Property	Description
Method	Retrieves the referenced method
Target	Retrieves the object of the class in which the delegate invokes the referenced method

System.Delegate Class 2-2

- The following table lists some of the methods defined in the `Delegate` class:

Method	Description
<code>Clone</code>	Makes a copy of the current delegate
<code>Combine</code>	Merges the invocation lists of the multicast delegates
<code>CreateDelegate</code>	Declares and initializes a delegate
<code>DynamicInvoke</code>	Calls the referenced method at run-time
<code>GetInvocationList</code>	Retrieves the invocation list of the current delegate

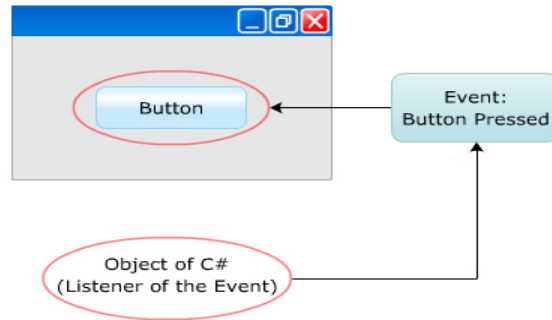
Delegates - Summary

- A delegate in C# is used to refer to a method in a safe manner.
- A single delegate can encapsulate the references of multiple methods at a time to hold a number of method references.
-

Section 2

EVENTS

- Consider a group of people at a party playing Bingo. When a number is called, the participants check if the number is on their cards whereas the non-participants go about their business, enjoying other activities.
- If this situation is analyzed from a programmer's perspective, the calling of the number corresponds to the occurrence of an event.
- The notification about the event is given by the announcer.
- Here, the people playing the game are paying attention (subscribing) to what the announcer (the source of the event) has to say (notify).
- Similarly, in C#, events allow an object (source of the event) to notify other objects (subscribers) about the event (a change having occurred).
- The following figure depicts the concept of events:



- An event is a user-generated or system-generated action that enables the required objects to notify other objects or classes to handle the event. Events in C# have the following features:
 - They can be declared in classes and interfaces.
 - They can be declared as abstract or sealed.
 - They can be declared as virtual.
 - They are implemented using delegates.
- Events can be used to perform customized actions that are not already supported by C#.
- Events are widely used in creating GUI based applications, where events such as, selecting an item from a list and closing a window are tracked.

Creating and Using Events

- Following are the four steps for implementing events in C#:

Define a public delegate for the event.

Create the event using the delegate.

Subscribe to listen and handle the event.

Raise the event.

- Events use delegates to call methods in objects that have subscribed to the event.
- When an event containing a number of subscribers is raised, many delegates will be invoked.

Declaring Events 1-2

- An event declaration consists of two steps, creating a delegate and creating the event. A delegate is declared using the delegate keyword.
- The delegate passes the parameters of the appropriate method to be invoked when an event is generated.
- This method is known as the event handler.
- The event is then declared using the event keyword followed by the name of the delegate and the name of the event.
- This declaration associates the event with the delegate.
- The following figure displays the syntax for declaring delegates and events:

Declaring a Delegate:

```
<access_modifier> delegate <return type> <Identifier> (parameters);
```

Declaring an Event:

```
<access_modifier> event <DelegateName> <EventName>;
```

Declaring Events 2-2

- An object can subscribe to an event only if the event exists. To subscribe to the event, the object adds a delegate that calls a method when the event is raised.
- This is done by associating the event handler to the created event, using the **+= addition assignment** operator which is known as subscribing to an event.
- To unsubscribe from an event, use the **-= subtraction assignment** operator.
- The following syntax is used to create a method in the receiver class:

Syntax

```
<access_modifier> <return_type> <MethodName> (parameters);
```

- where,
 - ✓ **objectName**: Is the object of the class in which the event handler is defined.

Raising Events 1-2

- An event is raised to notify all the objects that have subscribed to the event. Events are either raised by the user or the system.
- Once an event is generated, all the associated event handlers are executed. The delegate calls all the handlers that have been added to the event.
- However, before raising an event, it is important for you to create handlers and thus, make sure that the event is associated to the appropriate event handlers.
- If the event is not associated to any event handler, the declared event is considered to be null.
- The following figure displays the raising events:

```
public delegate void Display();  
  
class Events  
{  
    event Display Print;  
  
    void Show()  
    {  
        Console.WriteLine("This is an event driven program");  
    }  
  
    static void Main(string[] args)  
    {  
        Events objEvents = new Events();  
        objEvents.Print += new Display(objEvents.Show);  
        objEvents.Print();  
    }  
}  
  
Output:  
This is an event driven program
```

Invoking the Event Handler through the Created Event

Raising Events 2-2

- The following code can be used to check a particular condition before raising the event:

Snippet

```
if (condition)
{
    eventMe();
}
```

- In the code:
- If the checked condition is satisfied, the event eventMe is raised.
- The syntax for raising an event is similar to the syntax for calling a method. When eventMe is raised, it will invoke all the delegates of the objects that have subscribed to it. If no objects have subscribed to the event and the event has been raised, an exception is thrown.

Events - Summary

- An event is a data member that enables an object to provide notifications to other objects about a particular action.

Section 3

ANONYMOUS METHOD

- An anonymous method is an inline nameless block of code that can be passed as a delegate parameter.
- Delegates can invoke one or more named methods that are included while declaring the delegates.
- Prior to anonymous methods, if you wanted to pass a small block of code to a delegate, you always had to create a method and then pass it to the delegate.
- With the introduction of anonymous methods, you can pass an inline block of code to a delegate without actually creating a method.
- The following code displays an example of anonymous method:

```
void Action()  
{  
    System.Threading.Thread objThread = new  
        System.Threading.Thread  
        (delegate()  
        {  
            Console.Write("Testing... ");  
            Console.WriteLine("Threads.");  
        });  
    objThread.Start();  
}
```

} Anonymous Method

- An anonymous method is used in place of a named method if that method is to be invoked only through a delegate.
- An anonymous method has the following features:
 - ✓ It appears as an inline code in the delegate declaration.
 - ✓ It is best suited for small blocks.
 - ✓ It can accept parameters of any type.
 - ✓ Parameters using the `ref` and `out` keywords can be passed to it.
 - ✓ It can include parameters of a generic type.
 - ✓ It cannot include jump statements such as `goto` and `break` that transfer control out of the scope of the method.

Creating Anonymous Methods 1-3

- An anonymous method is created when you instantiate or reference a delegate with a block of unnamed code.
- Following points need to be noted while creating anonymous methods:
 - ✓ When a `delegate` keyword is used inside a method body, it must be followed by an anonymous method body.
 - ✓ The method is defined as a set of statements within curly braces while creating an object of a delegate.
 - ✓ Anonymous methods are not given any return type.
 - ✓ Anonymous methods are not prefixed with access modifiers.

Creating Anonymous Methods 2-3

- The following figure and snippet display the syntax and code for anonymous methods respectively:

```
// Create a delegate instance

<access modifier> delegate <return type>
<DelegateName> (parameters);

// Instantiate the delegate using an anonymous method

<DelegateName> <objDelegate> = new <DelegateName>
(parameters)
{ /* ... */ };
```

Snippet

```
using System;
class AnonymousMethods
{
    //This line remains same even if named methods are used
    delegate void Display();
    static void Main(string[] args)
    {
        //Here is where a difference occurs when using
        // anonymous methods
        Display objDisp = delegate()
        {
            Console.WriteLine("This illustrates an anonymous method");
        };
        objDisp();
    }
}
```

Creating Anonymous Methods 3-3

- In the code:
 - ✓ A delegate named **Display** is created.
 - ✓ The delegate **Display** is instantiated with an anonymous method.
 - ✓ When the delegate is called, it is the anonymous block of code that will execute.

This illustrates an anonymous method

Output

Referencing Multiple Anonymous Methods 1-2


- C# allows you to create and instantiate a delegate that can reference multiple anonymous methods.
- This is done using the += operator.
- The += operator is used to add additional references to either named or anonymous methods after instantiating the delegate.
- The following code shows how one delegate instance can reference several anonymous methods:

Snippet

```
using System;
class MultipleAnonymousMethods
{
    delegate void Display();
    static void Main(string[] args)
    {
        //delegate instantiated with one anonymous
        // method reference
        Display objDisp = delegate()
        {
            Console.WriteLine("This illustrates one anonymous
                               method");
        };
    }
}
```

Referencing Multiple Anonymous Methods 2-2

```
//delegate instantiated with another anonymous method
// reference
objDisp += delegate()
{
    Console.WriteLine("This illustrates another anonymous
        method with the same delegate instance");
};
objDisp();
```

- In the code,  An anonymous method is created during the delegate instantiation and another anonymous method is created and referenced by the delegate using the += operator.

This illustrates one anonymous method

This illustrates another anonymous method with the same delegate instance

Output

Anonymous Method - Summary

- Anonymous methods allow you to pass a block of unnamed code as a parameter to a delegate.

Section 4

SYSTEM-DEFINED GENERIC DELEGATES

System-Defined Generic Delegates

- A delegate is a reference to a method. Consider an example to understand this.
- You create a delegate **CalculateValue** to point to a method that takes a `string` parameter and returns an `int`.
- You need not specify the method name at the time of creating the delegate.
- At some later stage in your code, you can instantiate the delegate by assigning it a method name.
- The .NET Framework and C# have a set of predefined generic delegates that take a number of parameters of specific types and return values of another type.
- The advantage of these predefined generic delegates is that they are ready for reuse with minimal coding.

Func Delegate 1 - 2

- Following are the commonly used predefined generic delegates:

`Func<TResult>() Delegate`

It represents a method having zero parameters and returns a value of type **TResult**.

`Func<T, TResult>(T arg)
Delegate`

It represents a method having one parameter of type **T** and returns a value of type **TResult**.

`Func<T1, T2, TResult>(T1
arg1, T2 arg2) Delegate`

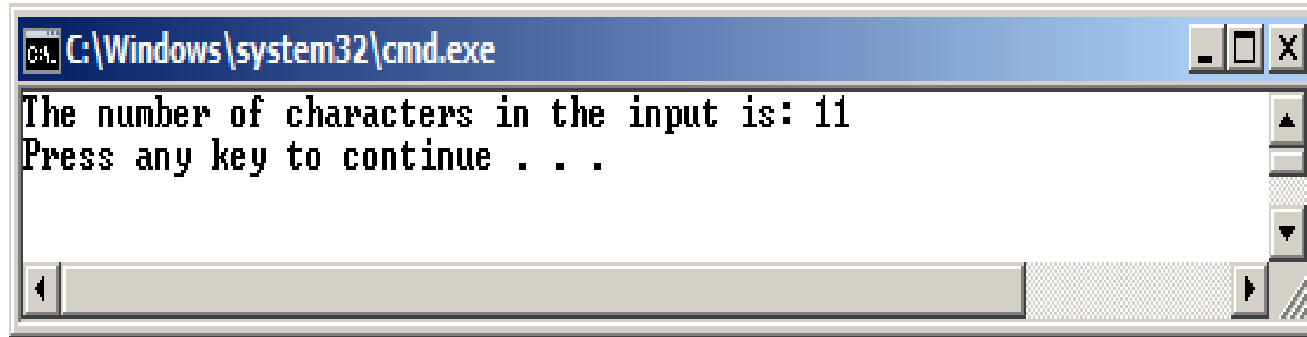
It represents a method having two parameters of type **T1** and **T2** respectively and returns a value of type **TResult**.

- The following code determines the length of a given word or phrase:

```
/// <summary>  
/// Class WordLength determines the length of a given word or phrase  
/// </summary>  
public class WordLength{  
    public static void Main() {  
        // Instantiate delegate to reference Count method  
        Func<string, int> count = Count;  
        string location = "Netherlands";  
        // Use delegate instance to call Count method  
        Console.WriteLine("The number of characters in the input is:  
{0}",count(location).ToString());  
    }  
}
```

Func Delegate 2 - 2

- The code:
 - ✓ Makes use of the `Func<T, TResult> (T arg)` predefined generic delegate that takes one parameter and returns a result.
- The following figure shows the use of a predefined generic delegate:



Action Delegate 1 - 3

- Action is also a delegate type defined in the System namespace. An Action type delegate is the same as Func delegate except that the Action delegate doesn't return a value. In other words, an Action delegate can be used with a method that has a void return type.

```
static void ConsolePrint(int i)
{
    Console.WriteLine(i);
}

static void Main(string[] args)
{
    Action<int> printActionDel = ConsolePrint;
    printActionDel(10);
}
```


Action Delegate 2 - 3

- Anonymous method with Action delegate

```
static void Main(string[] args)
{
    Action<int> printActionDel = delegate(int i)
    {
        Console.WriteLine(i);
    };

    printActionDel(10);
}
```

- Lambda expression with Action delegate

```
static void Main(string[] args)
{
    Action<int> printActionDel = i => Console.WriteLine(i);

    printActionDel(10);
}
```

Section 5

ANONYMOUS TYPES

Anonymous Types 1-8

- Anonymous type:
 - ✓ Is basically a class with no name and is not explicitly defined in code.
 - ✓ Uses object initializers to initialize properties and fields. Since it has no name, you need to declare an implicitly typed variable to refer to it.

Syntax

```
new { identifierA = valueA, identifierB = valueB, ..... }
```

where,

- ✓ `identifierA, identifierB, ...`: Identifiers that will be translated into read-only properties that are initialized with values

Anonymous Types 2-8

The following code demonstrates the use of anonymous types:

Snippet

```
using System;
/// <summary>
/// Class AnonymousTypeExample to demonstrate anonymous type
/// </summary>
class AnonymousTypeExample
{
    public static void Main(string[] args)
    {
        // Anonymous Type with three properties.
        var stock = new { Name = "Michigan Enterprises", Code = 1301,
            Price = 35056.75 };
        Console.WriteLine("Stock Name: " + stock.Name);
        Console.WriteLine("Stock Code: " + stock.Code);
        Console.WriteLine("Stock Price: " + stock.Price);
    }
}
```

Consider the following line of code:

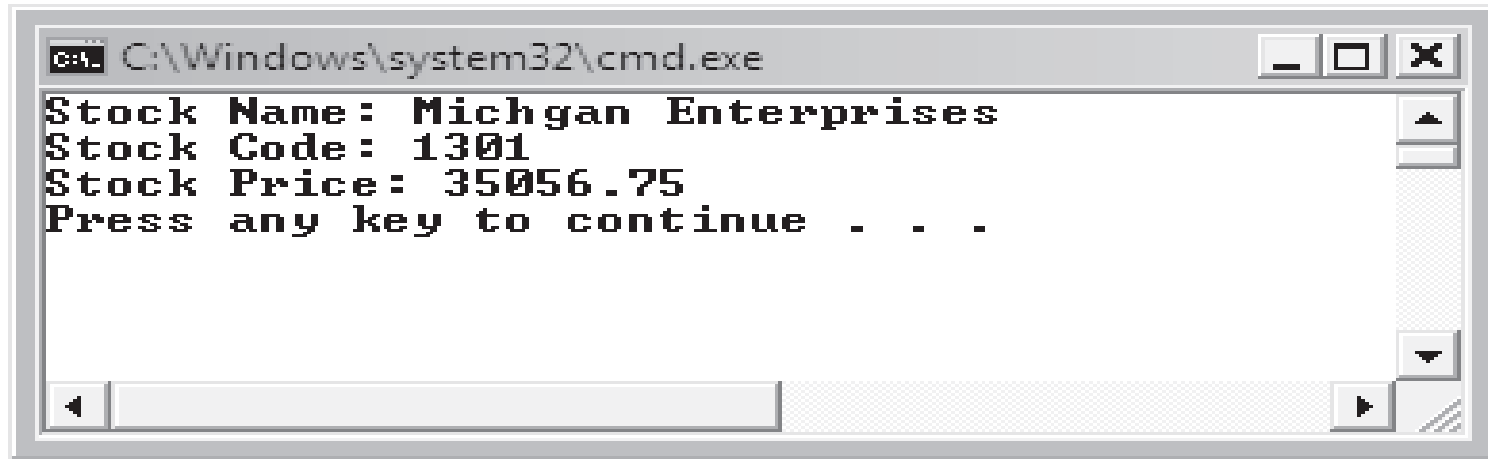
```
var stock = new { Name = "Michigan Enterprises", Code = 1301, Price = 35056.75 };
```

The compiler creates an anonymous type with all the properties that is inferred from object initializer.

In this case, the type will have properties **Name**, **Code**, and **Price**.

Anonymous Types 3-8

- The compiler automatically generates the `get` and `set` methods, as well as the corresponding private variables to hold these properties.
- At runtime, the C# compiler creates an instance of this type and the properties are given the values Michigan Enterprises, 1301, and 35056.75 respectively.
- The following figure displays output:



```
C:\Windows\system32\cmd.exe
Stock Name: Michigan Enterprises
Stock Code: 1301
Stock Price: 35056.75
Press any key to continue . . .
```

- When an anonymous type is created, the C# compiler carries out the following tasks:
 - ✓ Interprets the type
 - ✓ Generates a new class
 - ✓ Use the new class to instantiate a new object
 - ✓ Assigns the object with the required parameters
- The compiler internally creates a class with the respective properties when code is compiled.
- In this program, the class might look like the one that is shown in code.

Anonymous Types 5-8

- In this program, the class might look like the one that is shown in the following code:

Snippet

```
class __NO_NAME__  
{  
    private string _Name;  
    private int _Code;  
    private double _Price;  
    public string Name  
    {  
        get { return _Name; }  
        set { _Name = value; }  
    }  
    public int Code  
    {  
        get { return _Code; }  
        set { _Code = value; }  
    }  
    public double Price  
    {  
        get { return _Price; }  
        set { _Price = value; }  
    }  
}
```

Anonymous Types 6-8

- The following code demonstrates passing an instance of the anonymous type to a method and displaying the details:

Snippet

```
using System;
using System.Reflection;
/// <summary>
/// Class Employee to demonstrate anonymous type.
/// </summary>
public class Employee
{
    public void DisplayDetails(object emp)
    {
        String fName = "";
        String lName = "";
        int age = 0;
        PropertyInfo[] attrs = emp.GetType().GetProperties();
        foreach (PropertyInfo attr in attrs)
        {
            switch (attr.Name)
            {
                case "FirstName":
                    fName = attr.GetValue(emp, null).ToString();
                    break;
                case "LastName":
                    lName = attr.GetValue(emp, null).ToString();
                    break;
                case "Age":
                    age = (int)attr.GetValue(emp, null);
                    break;
            }
        }
    }
}
```


Anonymous Types 7-8

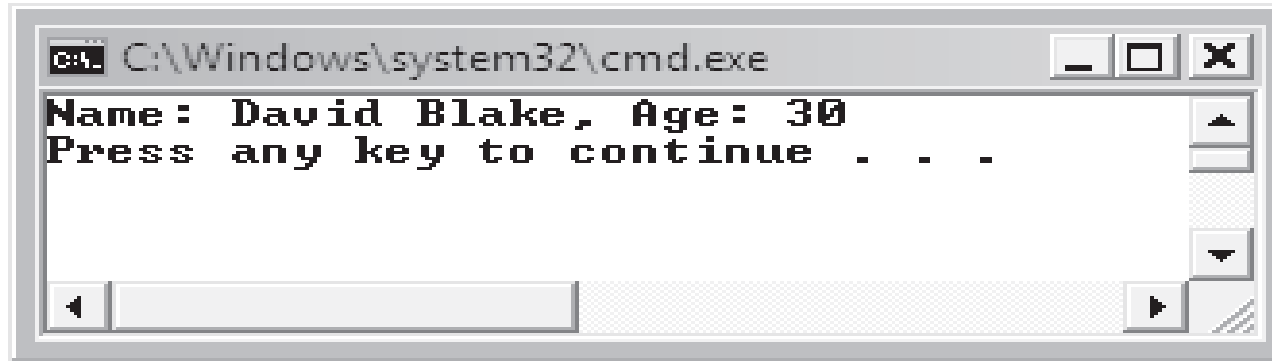
```
        Console.WriteLine("Name: {0} {1}, Age: {2}", fName, lName,
            age);
    }
}
class AnonymousExample
{
    public static void Main(string[] args)
    {
        Employee david = new Employee();
        // Creating the anonymous type instance and passing it
        // to a method.
        david.DisplayDetails(new { FirstName = "David", LastName =
            "Blake", Age = 30 });
    }
}
```

In the code:

- ✓ It creates an instance of the anonymous type with three properties, **FirstName**, **LastName**, and **Age** with values David, Blake, and 30 respectively.
- ✓ This instance is then passed to the method, **DisplayDetails()**.
- ✓ In **DisplayDetails()** method, the instance that was passed as parameter is stored in the object, emp.

Anonymous Types 8-8

- ✓ Then, the code uses reflection to query the object's properties.
 - ✓ The **GetType()** method retrieves the type of the current instance, **emp** and **GetProperties()** method retrieves the properties of the object, **emp**.
 - ✓ The details are then stored in the **PropertyInfo** collection, **attr**. Finally, the details are extracted through the **GetValue()** method of the **PropertyInfo** class.
 - ✓ If this program did not make use of an anonymous type, a lot more code would have been required to produce the same output.
- The following figure displays the output:



A screenshot of a Windows command prompt window. The title bar shows 'cmd' and the path 'C:\Windows\system32\cmd.exe'. The window contains the following text:

```
Name: David Blake, Age: 30  
Press any key to continue . . .
```

The text is displayed in a monospaced font. Below the text, there is a horizontal scrollbar and a vertical scrollbar on the right side of the window.

Anonymous Types - Summary

- You can create an instance of a class without having to write code for the class beforehand by using a new feature called anonymous types.

Lesson Summary

- A delegate in C# is used to refer to a method in a safe manner.
- An event is a data member that enables an object to provide notifications to other objects about a particular action.
- Anonymous methods allow you to pass a block of unnamed code as a parameter to a delegate.
- You can create an instance of a class without having to write code for the class beforehand by using a new feature called anonymous types.

Thank you

