VT Academy
Thinking Ahead, Beyond Boundaries

Object Oriented Programming

Delegate, Event and Lambda Expression

Session 9

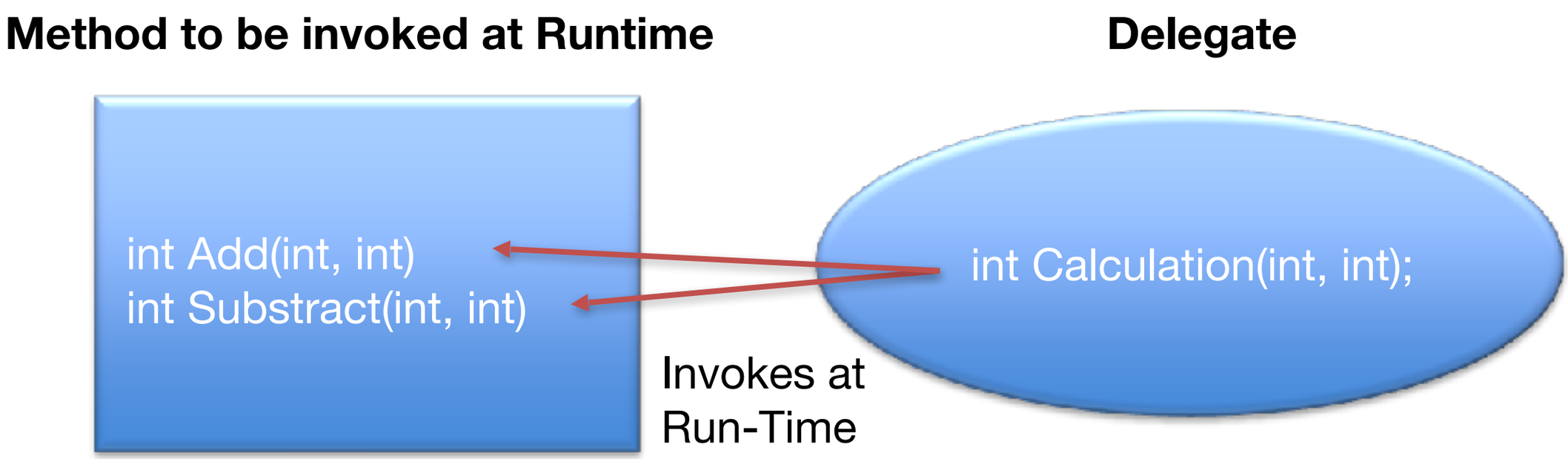
Objectives

VTC Academy
Thinking Ahead, Beyond Boundaries

- Delegates
- Events
- Lambda Expressions

Delegates

- C# delegates are similar to pointers to functions, in C or C++
- A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.
- Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.



Delegates Benefits

- Ensures type safety
- Directly execute statements
- Invokes multiple methods
- Refers both instance and static methods

Declaring Delegates

- Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which has the same signature as that of the delegate.

- Syntax:

```
delegate <return type> <delegate-name> <parameter list>
```

- Example:

```
public delegate int MyDelegate (string s);
```

Instantiating Delegates

- Once a delegate type is declared, a delegate object must be created with the **new** keyword and be associated with a particular method.

- When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method.

- For example:

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

Delegates Example

```
using System;
delegate int NumberChanger(int n);
namespace DelegateDemo {
   class Program {
      static int num = 10;
      public static int AddNum(int p) {
         num += p;
         return num;
      }
      public static int MultiplyNum(int q) {
         num *= q;
         return num;
      }
      public static int getNum() {
         return num;
      }
      static void Main(string[] args) {
         // create delegate instances
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultiplyNum);
         // calling the methods using the delegate objects
         nc1(25);
         Console.WriteLine("Value of Num: {0}", getNum());
         nc2(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

Multicasting of a Delegate

- Delegate objects can be composed using the "+" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed.
- The "-" operator can be used to remove a component delegate from a composed delegate.
- Using this property of delegates you can create an invocation list of methods that will be called when a delegate is invoked. This is called **multicasting** of a delegate.

Multicasting of a Delegate

VTC Academy
Thinking Ahead, Beyond Boundaries

```
using System;
delegate int NumberChanger(int n);
namespace DelegateDemo {
   class Program {
      static int num = 10;
      public static int AddNum(int p) {
         num += p;
         return num;
      }
      public static int MultplyNum(int q) {
         num *= q;
         return num;
      }
      public static int getNum() {
         return num;
      }
      static void Main(string[] args){
         // create delegate instances
         NumberChanger nc;
         NumberChanger nc1 = new NumberChanger(AddNum);
         NumberChanger nc2 = new NumberChanger(MultiplyNum);
         nc = nc1;
         nc += nc2;
         // calling multicast
         nc(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

Events

- **Events** are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications.
- Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

Using Delegates with Events

- The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class.

- The class containing the event is used to publish the event. This is called the **publisher** class. Some other class that accepts this event is called the **subscriber** class. Events use the **publisher-subscriber** model.

- A **publisher** is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.

- A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

Creating and Using Events

- In C#, Events are created using delegated. Delegates are widely used for event handling.
- There are five steps for implementing events

1. Define a public delegate for the event
2. Create the event using the delegate
3. Subscribe to listen and handle the event
4. Raise the event
5. Events use delegates to call methods in objects that have subscribed to the event

Declaring Events

- TTo declare an event inside a class, first a delegate type for the event must be declared.
- For example:

```
public delegate string MyDel(string str);
```

- Next, the event itself is declared, using the **event** keyword

```
event MyDel MyEvent;
```

Events Example

```
using System;
namespace EventsDemo {
    public delegate string MyDelegate(string str);

    class Program {
        // 1. Declaring event
        event MyDelegate MyEvent;
        public EventProgram() {
            // 2. Subcribing to event
            this.MyEvent += new MyDelegate(this.WelcomeUser);
        }

        public string WelcomeUser(string username) {
            return "Welcome " + username;
        }
        static void Main(string[] args) {
            EventProgram eventpro = new EventProgram();
            // Raising event
            string result = eventpro.MyEvent("VTC Academy");
            Console.WriteLine(result);
        }
    }
}
```

Lambda Expressions

- A lambda expression is an anonymous function that you can use to create delegates or expression tree types.
- By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.
- To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator =>, and you put the expression or statement block on the other side.
- For example:
  - Lambda expression **x => x * x** specifies a parameter that's named **x** and returns the value of **x** squared.

Lambda Expressions Example

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

- The **=>** operator has the same precedence as assignment (=) and is right associative
- In the previous example, notice that the delegate signature has one implicitly-typed input parameter of type int, and returns an int. The lambda expression can be converted to a delegate of that type because it also has one input parameter (x) and a return value that the compiler can implicitly convert to type int.

Expression Lambdas

- A lambda expression with an expression on the right side of the => operator is called an expression lambda. Expression lambdas are used extensively in the construction of Expression Trees.

- An expression lambda returns the result of the expression and takes the following basic form:

```
(input-parameters) => expression
```

- The parentheses are optional only if the lambda has one input parameter; otherwise they are required:

```
(x, y) => x == y
```

- Sometimes it is difficult or impossible for the compiler to infer the input types:

```
(int x, string s) => s.Length > x
```

Statement Lambdas

- AA statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces:

```
(input-parameters) => { statement; }
```

- Statement lambdas, like anonymous methods, cannot be used to create expression trees.

- The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three:

```
delegate void TestDelegate(string s);

TestDelegate del = n => { string s = n + " World";
                          Console.WriteLine(s); };
```

Summary

- C# delegates are similar to pointers to functions, in C or C++
- A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.
- **Events** are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications
- A lambda expression is an anonymous function that you can use to create delegates or expression tree types.
- By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.