Delegates

The word delegate in a personal context could mean entrusting another to complete a set of instructions or tasks for a given situation. I delegate my colleague to organize a meeting. The meeting could include several actions such as invitations, setting the agenda, booking equipment, and ordering food!

The **delegate** reserved-word in code relates to its meaning in the real world. Execute a series of actions or methods. Business logic requires a series of actions or methods that need to execute in a particular order, at a certain time, or when a condition arises. Delegates help to engineer flexible responsive software in code.

Technical Note:

A **delegate** is a reference that specifies a method or methods to call. Delegates implement callbacks and execute events. In an Object Oriented Programming language such as C Sharp, everything you declare is an object. A delegate object encapsulates a reference to a method. The delegate object contains code that can call the referenced method, without having to know at compile time which method or methods will be invoked.

Any method from any accessible class or struct that matches the delegate type (signature) can be assigned to the delegate. The method can be either static or an instance method. This flexibility means you can programmatically change method calls, or plug new code into existing classes.

You can use a delegate as a parameter in a method making delegates ideal for defining callback methods. The first method calls the second method passing the delegate reference as an argument and the second method calls back another method.

A typical application places the business logic in a Library that declares the delegate and methods to execute. The client application references the library. The application can declare a reference to the delegate and use it to execute the methods attached to it. Using this approach the business logic of a process is separate from the client application making it easier to update, change, and maintain over time.

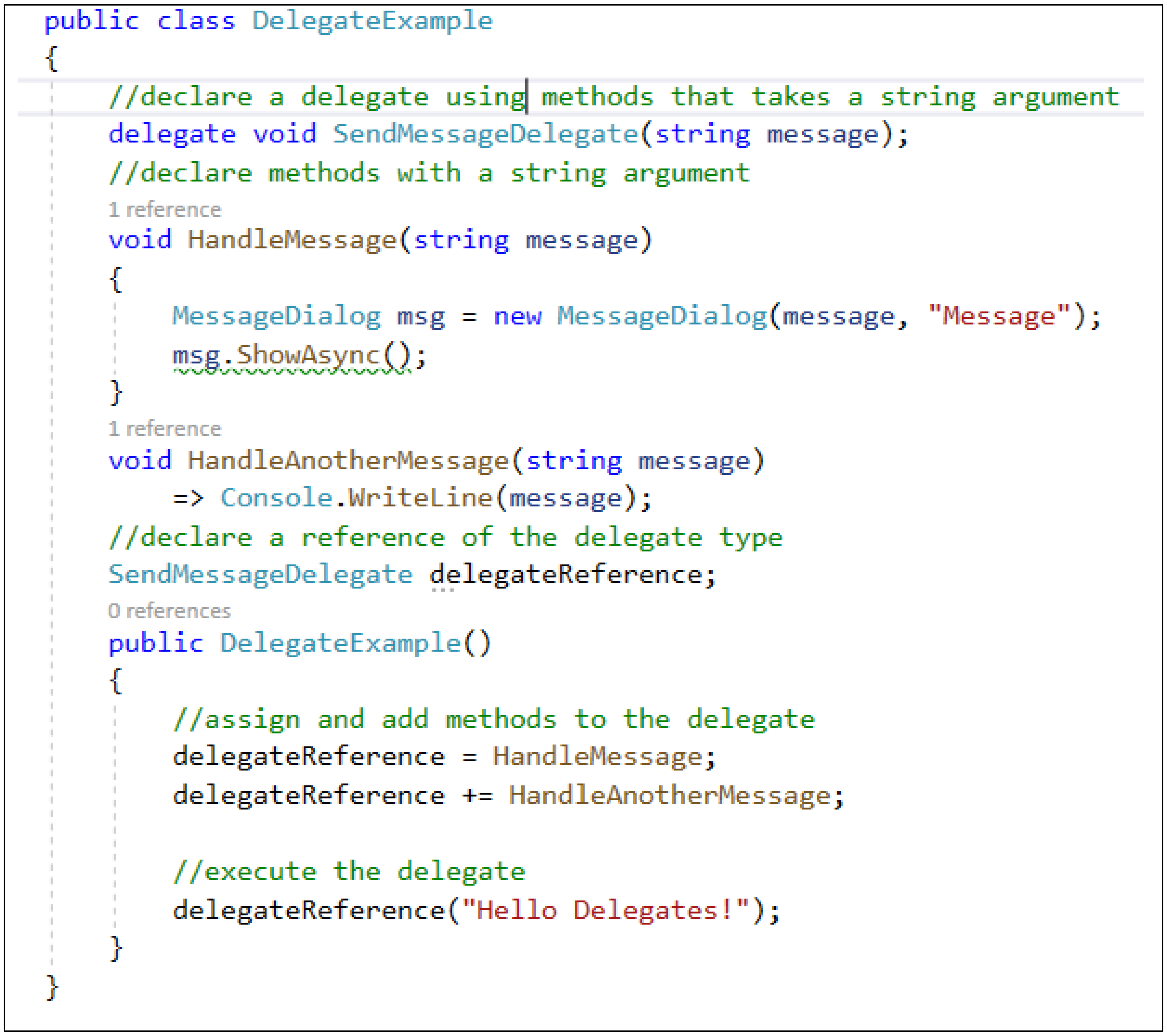
Key Properties of Delegates

- Delegates in C# are fully object-oriented, and encapsulate both an object instance and a method.
- Delegates allow methods to be passed as parameters.
- Delegates are used to define callback methods.
- Delegates are chained together; for example, multiple methods can be called on a single event.
- Methods don't have to match the delegate type exactly.
- Lambda expressions are a more concise way of writing inline code blocks. With a single syntax, you can express a method definition, declaration, and the invocation of delegate to execute it.

Key Steps in using Delegates:

- Declare the delegate
- Implement a method(s) that runs when the delegate is called
- Declare an instance of the delegate and use that reference to call the methods associated with it

Example:



```
public class DelegateExample
{
    //declare a delegate using methods that takes a string argument
    delegate void SendMessageDelegate(string message);
    //declare methods with a string argument
    1 reference
    void HandleMessage(string message)
    {
        MessageDialog msg = new MessageDialog(message, "Message");
        msg.ShowAsync();
    }
    1 reference
    void HandleAnotherMessage(string message)
        => Console.WriteLine(message);
    //declare a reference of the delegate type
    SendMessageDelegate delegateReference;
    0 references
    public DelegateExample()
    {
        //assign and add methods to the delegate
        delegateReference = HandleMessage;
        delegateReference += HandleAnotherMessage;

        //execute the delegate
        delegateReference("Hello Delegates!");
    }
}
```

Events

An event is a notification sent by an object to signal the occurrence of an action. The class that raises events is the **Publisher** and the class that receives the event is the **Subscriber**. There can be multiple subscribers to an event. For example: many buttons subscribing to the same click event.

An event uses a delegate. The delegate defines the signature for the event handler method used by the subscribing class.

Example: You register a button click event and write code in the method handler provided. In the properties window for UWP you select the lightning bolt and type in a method name for the event you want to use. The line of code generated:

btnSubmit.Click += Submit_Click;

In the code behind the page, you are provided a handler method for the event that matches its underlying delegate:

private void Submit_Click(object sender, object e) { … }

Events involve declaring the event; raising the event; and consuming the event.

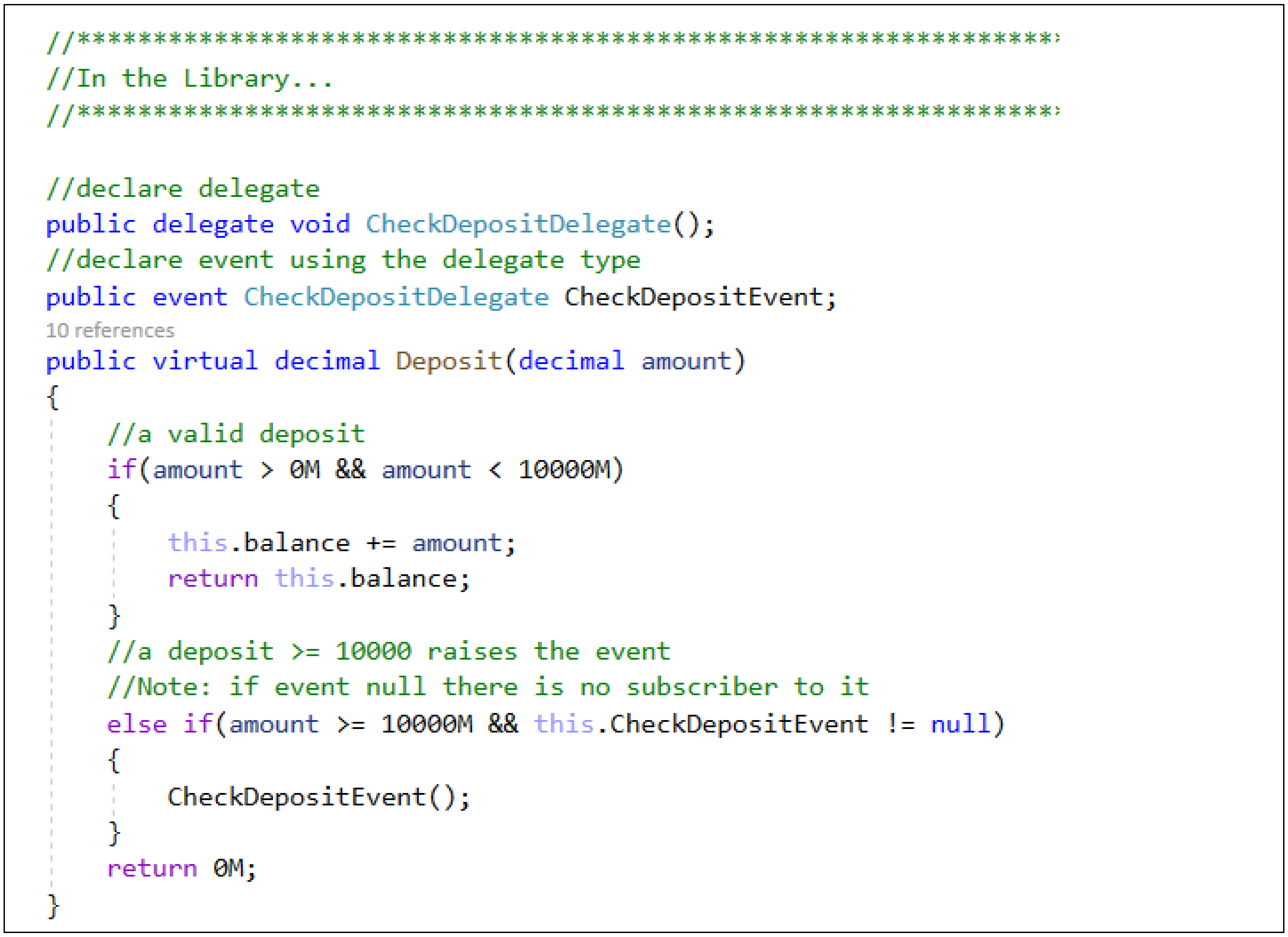
Key Steps in using Events:

1. Declare a **delegate**
2. Declare a variable of the delegate with **event** keyword
3. Write code (For Example: business logic) to raise the event for a given situation and check if the event is registered or subscribed to: check for null
4. Subscribe to the event or register the event in another class (client application)
5. Write a method with code to execute when the event is raised (step 3 above contains logic to raise the event)

For components such as a Button, the first three steps above are within the API. In your UWP application, you complete steps 4 and 5 above.

Example:

Using a Bank Account class if 10,000 or more is deposited an event is raised. The publishing code written in the Bank Account class is separate from the client application that subscribes to the event:



```
//*****************************************************************
//In the Library...
//*****************************************************************

//declare delegate
public delegate void CheckDepositDelegate();
//declare event using the delegate type
public event CheckDepositDelegate CheckDepositEvent;
10 references
public virtual decimal Deposit(decimal amount)
{
    //a valid deposit
    if(amount > 0M && amount < 10000M)
    {
        this.balance += amount;
        return this.balance;
    }
    //a deposit >= 10000 raises the event
    //Note: if event null there is no subscriber to it
    else if(amount >= 10000M && this.CheckDepositEvent != null)
    {
        CheckDepositEvent();
    }
    return 0M;
}
```

A client application can subscribe to the Bank example above as follows:

```
BankAccount account = new SavingAccount();            //declare a bank account object
account.CheckDepositEvent += EventHandlerMethod;  //subscribe (register) the event
decimal amount = account.Deposit(10000M);          //causes the event
private void EventHandlerMethod()
{
  //this code executes when the event is raised
}
```

Note: There is a requirement by Canadian banks to report deposits $10,000 plus. In this example, the bank applications can write their own code in the event handler method to report the deposit. It may log to a file, execute another business-to-business process with the government, etc. Just like, you write the code for the click event of your buttons. You register the event you want. The method handler is provided, and you write the code to respond to that event.

Built-in Event Handler Delegate in .NET Framework API

The .NET Framework includes built-in delegate types for common events so that they have the same signature as other events you are used to using with components. Typically, these events include two parameters: the source called the sender and event data that is specific to each type of event. You can avoid declaring delegates if you choose to use these events. You declare the following using EventHandler from the .NET Framework:

public event EventHandler SomeProcess;

You still need to write logic-code that checks if the event is subscribed to and raise the event such as in the Deposit method shown above. You then subscribe to it in your application (in another class) and code the Handler method when the event is raised.

Final Notes

- An event is a wrapper around a delegate. It depends on the delegate.
- Use "event" keyword with delegate type variable to declare an event.
- Use built-in delegate EventHandler or EventHandler<TEventArgs> for common events.
- The publisher class raises an event, and the subscriber class registers for an event and provides the event-handler method.
- The signature of the handler method must match the delegate signature.
- Register with an event using the += operator. Unsubscribe it using the -= operator. Cannot use the = operator.
- Pass event data using EventHandler<TEventArgs>.
- Derive EventArgs base class to create custom event data class.
- Events can be declared static, virtual, sealed, and abstract.
- An Interface can include the event as a member.
- Event handlers are invoked synchronously if there are multiple subscribers.

A free online chapter covering Delegates, Events, and Lambda Expressions from O'Rielly:

https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2008/ff518994(v=orm.10)