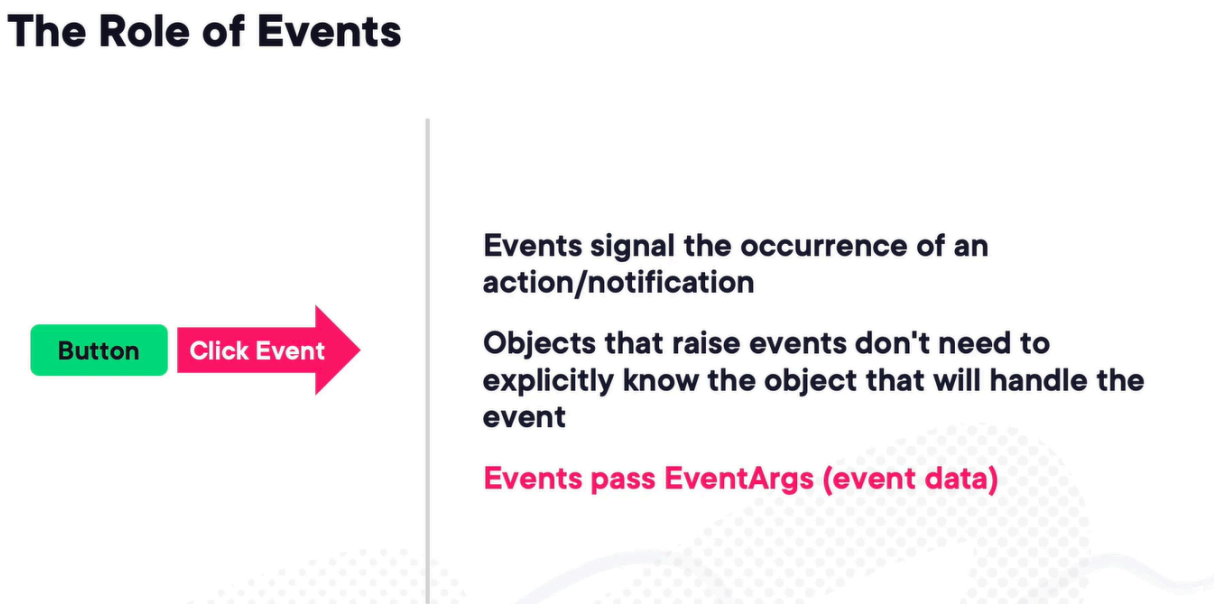
2.1 The Role of Events:

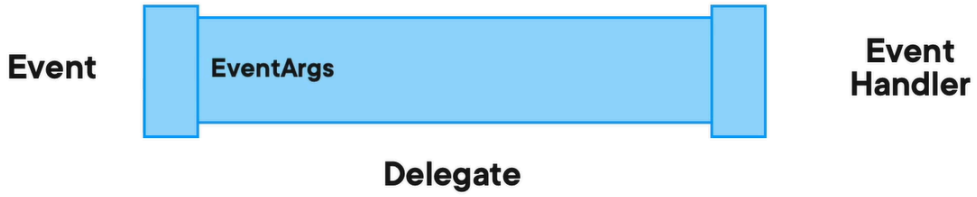
-An Event is a notification.
-Provide a way to trigger notifications from end users or objects.



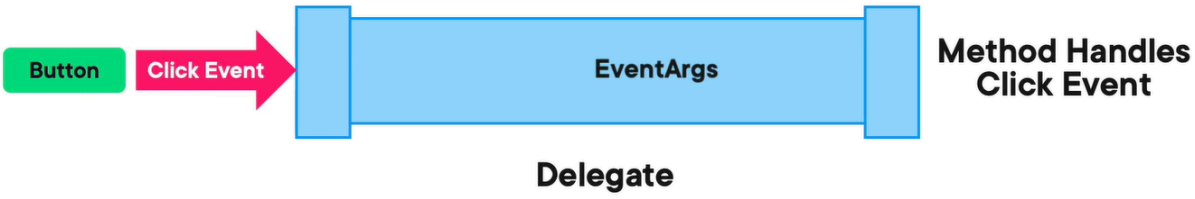
2.2 The Role of Delegates:

-Responsible for connecting Events to Event Handlers.

Delegates Are a Pipeline

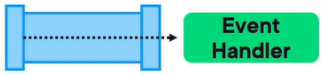


Delegates Are a Pipeline



2.3 The Role of Event Handlers:

What Is an Event Handler?



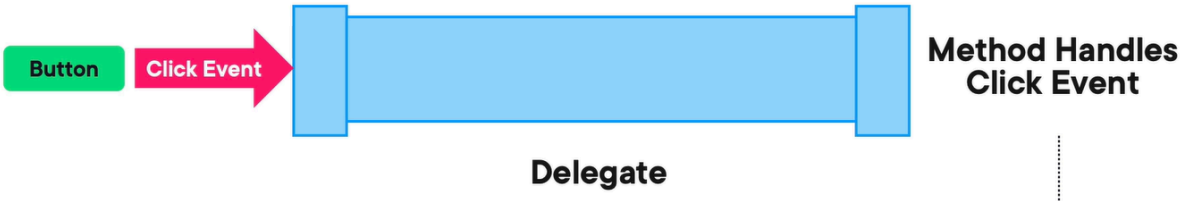
An event handler is responsible for receiving and processing data from a delegate

Event handler may receive parameters, for example:

- Sender
- EventArgs

EventArgs are responsible for encapsulating event data

Delegates Are a Pipeline

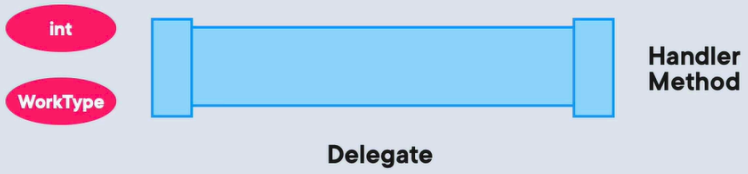



```
public void Submit_Click(object sender, EventArgs e)
{
  // Handle button click here
}
```

3.1 Creating a Delegate:

-Function pointers pretty much.

```
public delegate int WorkPerformedHandler(int hours, WorkType workType);
```



Creating a Delegate

Custom delegates are defined using the **delegate** keyword.

-In C#, a **delegate** is essentially a type-safe function pointer, which can hold references to methods with a particular signature (input parameters and return type). This allows delegates to be called like methods, but they can also store multiple methods (functions) in a chain or multicast delegate.

Why can delegates be chained like this?

When you use the `+=` operator on delegates, you're adding multiple method references to the delegate. This is allowed because:

- **Delegates are multicast**: A delegate can hold references to more than one method. When a delegate is invoked, it calls all the methods in the order they were added to the delegate (this is known as a *multicast delegate*).

- **Data structure of a delegate**: A delegate is essentially a reference type that contains a list (or array) of method references, which it can invoke. The delegate itself doesn't hold the method bodies, but rather, references to methods that share the same signature. This list of references forms the underlying "chain" when delegates are combined with `+=`.

Internally how does this work?

When you chain delegates using the `+=` operator, C# internally uses a data structure to store these references. This data structure could be considered like a **linked list** or **array** of method pointers. The key idea is that a delegate instance holds an invocation list, which stores references to the methods in the order they were added.

Example breakdown:

1.Delegate creation:

```
var delegate1 = new WorkPerformedHandler(WorkPerformed1);
var delegate2 = new WorkPerformedHandler(WorkPerformed2);
```

At this point, `delegate1` and `delegate2` each point to one method (`WorkPerformed1` and `WorkPerformed2` respectively).

2.Chaining delegates:

```
delegate1 += delegate2;
```

When you use `+=`, C# combines the methods referred to by `delegate1` and `delegate2` into a single invocation list (i.e., it adds the reference to `WorkPerformed2` into `delegate1`'s internal invocation list).

Delegate invocation:

```
int finalHours = delegate1(10, WorkTypeEnum.Golf);
```

Calling `delegate1` now invokes both `WorkPerformed1` and `WorkPerformed2` in order. The return value from each function is used in the chain (though in your example, the final return value isn't collected or used because the delegates return `int` but don't pass it along).

Data Structure:

Internally, a delegate is typically represented as a class with the following key components:

1. **Target object**: The object to which the method belongs (if it is an instance method).
2. **Method pointer**: A reference to the method itself (if it's a static method or an instance method on a particular object).

3. **Invocation list**: An array (or list) of method references. When you chain delegates, this list is modified to hold multiple methods.

Multicast Delegate:

When multiple methods are chained, the delegate becomes a multicast delegate. The delegate will invoke each method in the order they were added. If one of the methods in the chain throws an exception, the remaining methods in the chain will not be executed.

Summary:

- A delegate is a **type-safe function pointer** that can hold references to one or more methods.
- You can chain delegates together using the `+=` operator, which adds method references to the delegate's internal invocation list.
- The delegate itself is likely implemented as a class that holds an **invocation list** of method references, allowing multiple methods to be executed when the delegate is invoked.
- The **multicast delegate** feature allows calling multiple methods in sequence when a single delegate is invoked.

By chaining delegates, you can decouple the caller from the specific methods it needs to invoke, allowing for flexible, dynamic method invocation.

3.2 Defining an Event:

```
public event WorkPerformedHandler? WorkPerformed;
```

Defining an Event

Events can be defined in a class using the **event** keyword.

-WorkPerformedHandler is a Delegate.

3.3 Raising an Event:

-Old syntax:

```
public event WorkPerformedHandler? WorkPerformed;

...

if (WorkPerformed != null) {
    WorkPerformed(8, WorkType.GenerateReports);
}
```

Raising Events

Events are raised by calling the event like a method.

-New syntax:

```
public event WorkPerformedHandler? WorkPerformed;

...

WorkPerformed?.Invoke(8, WorkType.GenerateReports);
```

Using Invoke() to Raise an Event

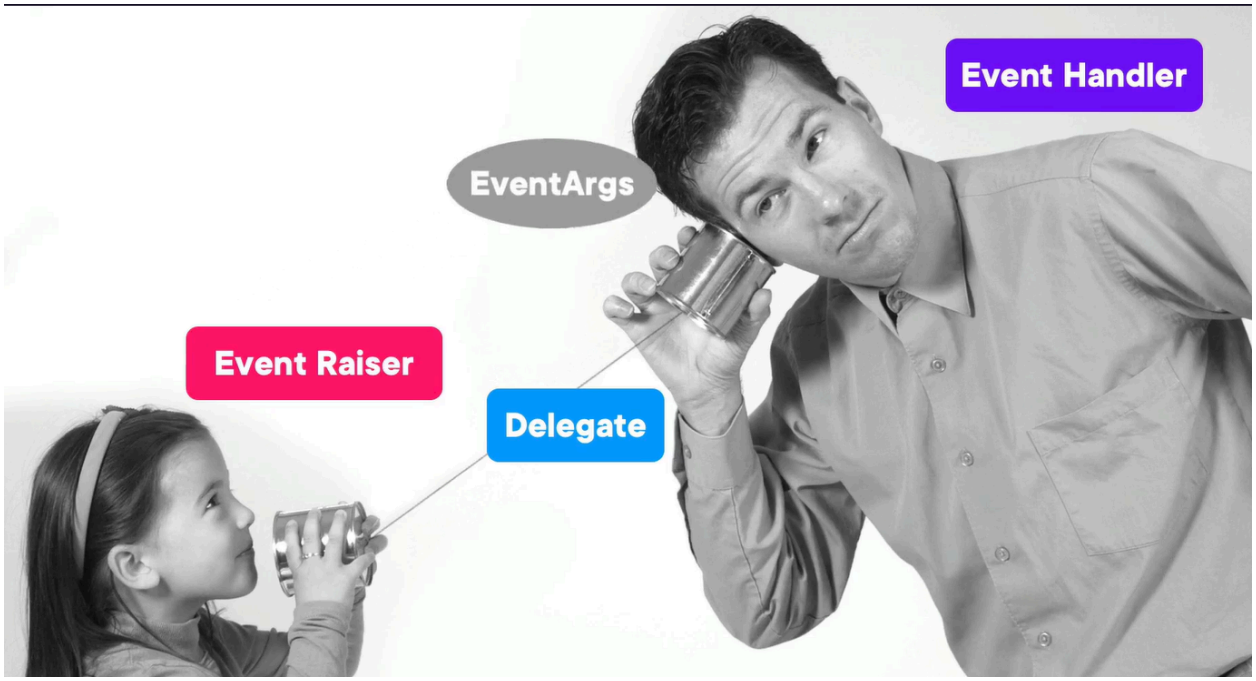
The Invoke() method can be used to simplify the code required to raise an event.

-Exposing and raising events:

Exposing and Raising Events

```
public delegate void WorkPerformedHandler(int hours, WorkType workType);

public class Worker
{
    public event WorkPerformedHandler? WorkPerformed;
    public void DoWork(int hours, WorkType workType)
    {
        // Do work here and notify consumer that work has been performed
        OnWorkPerformed(hours, workType);
    }

    protected virtual void OnWorkPerformed(int hours, WorkType workType)
    {
         WorkPerformed?.Invoke(hours, WorkType.GenerateReports);
    }
}
```

3.4 Creating an EventArgs Class:



```
public class WorkPerformedEventArgs
{
    public int Hours { get; }
    public WorkType WorkType { get; }

    public WorkPerformedEventArgs(int hours, WorkType workType) =>
      (Hours, WorkType) = (hours, workType);
}
```

Creating a Custom EventArgs Class

An EventArgs class defines properties that hold data that is sent when the event is raised.

```
public delegate void WorkPerformedHandler(object sender,
  WorkPerformedEventArgs e);
```

Using a Custom EventArgs Class

To use a custom EventArgs class, the delegate must reference the class in its signature.

```
public deleg                                    bject sender,
  WorkPerfor                                  e);

public event EventHandler<WorkPerformedEventArgs>? WorkPerformed;
```

Using EventHandler<T>

.NET includes a generic **EventHandler<T>** class that can be used instead of a custom delegate.

4.1 Instantiating Delegates and Handling Events:

```
public delegate int WorkPerformedHandler(object sender, WorkPerformedEventArgs e);



public void Manager_WorkPerformed(object sender, WorkPerformedEventArgs e) {
...
}
```

Delegate and Event Handler Method Parameters

The delegate and event handler parameter signatures must match.

```
var worker = new Worker();
worker.WorkPerformed +=
  new EventHandler<WorkPerformedEventArgs>(Worker_WorkPerformed);
```

Defining and Attaching Event Handlers

The **+=** operator is used to attach an event to an event handler.

-With the “+=” assignment operator we add another function pointer to the event pipeline(Invocation list) and whenever that event is INVOKED, all the functions inside that pipeline will be executed.
-There is also a ”-=”(minus equals) removes the function pointer from the pipeline(Invocation list).

```
var worker = new Worker();

worker.WorkPerformed +=
  new EventHandler<WorkPerformedEventArgs>(Worker_WorkPerformed);


void Worker_WorkPerformed(object sender, WorkPerformedEventArgs e)
{
    Console.WriteLine(e.Hours.ToString());
}
```

-So in hindsight, to work with events we need:

1)A FUNCTION TYPE: which is a delegate that specifies the signature of the function. When initializing the delegate we can add multiple function pointers(something like a linked list) to it with the same signature and when the delegate executes it executes the whole list of functions. We can also use the built-in generic delegate called EventHandler.
So a delegate is a type for functions and when initiated can hold multiple function pointers with the same signatures and when executed calls all the functions in sequence of adding.
A delegate is a type that represents references to methods with a particular parameter list and return type.

2)AN EVENT: Which is a container for the functions of the same type that will be executed in sequence when the event is invoked.

3)THE FUNCTION/S OF THE TYPE: The function that will be added to the event container for executing when the event is invoked.

4.2 Delegate Inference

```
var worker = new Worker();

worker.WorkPerformed += Worker_WorkPerformed;


void Worker_WorkPerformed(object sender, WorkPerformedHandler e)
{
    Console.WriteLine(e.);
}
```

Delegate Inference

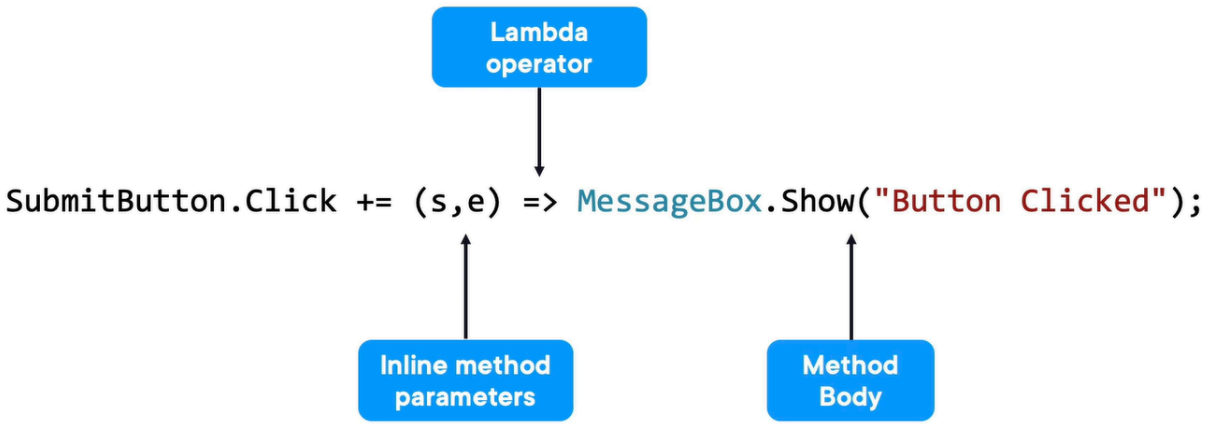
Delegate inference can be used to simplify attaching the event to the event handler.

5.Lambdas, Action<T> and Func<T, TResult>

5.1 Understanding Lambdas and Delegates:

-Lambdas are like shortcut methods(Anonymous functions pretty much)
-Example:

Understanding Lambda Expressions



-The parameters types are handled by the compiler.
-The lambda operator is a "separator" of between the input parameters and function body.
-Method body is just the body of the function.

-Another example:

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

static void Main(string[] args)
{
    AddDelegate ad = (a,b) => a + b;
    int result = ad(1,1); //result = 2
}
```

Assigning a Lambda to a Delegate

Lambda expressions can be assigned to any delegate.

-The a+b expression result is returned here, because there are no curly bracers, that is implied. It would look like this without the shorter syntax (a,b) => { return a + b;}

-Another example:

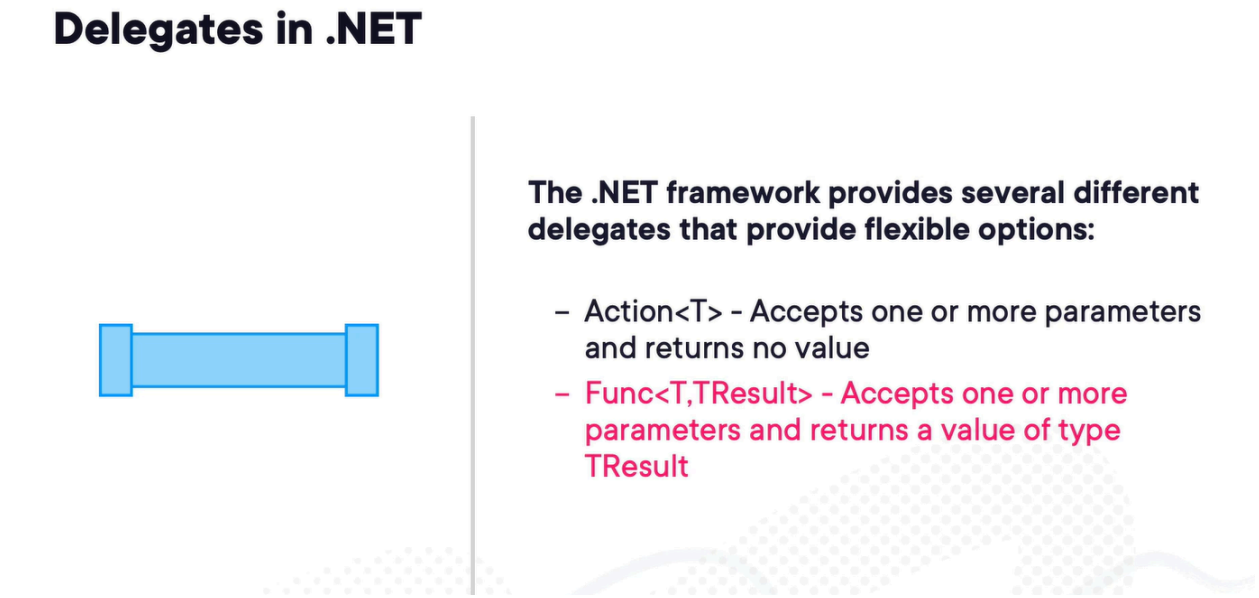
```
delegate bool LogDelegate();

static void Main(string[] args)
{
    LogDelegate ld = () =>
    {
        WriteToEventLog();
        return true;
    };
    bool status = ld();
}
```

Handling Empty Parameters

Delegates that don't accept any parameters can be handled using lambdas.

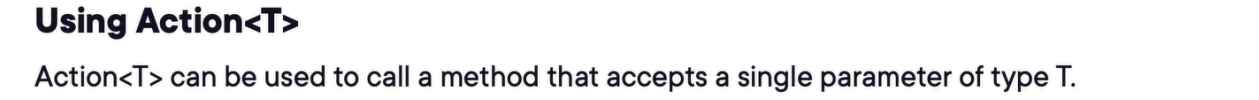
5.2 Using Action<T>:

-Built in Delegate (Function pointer).



-Example:

```
public static void Main(string[] args)
{
    Action<string> messageTarget;
    if (args.Length > 1) messageTarget = ShowWindowsMessage;
    else messageTarget = Console.WriteLine;
    messageTarget("Invoking Action!");
}

private static void ShowWindowsMessage(string message)
{
    MessageBox.Show(message);
}
```



-Action<string> messageTarget creates a delegate with one input parameter with type of string.

5.3 Using Func<T, TResult>:

-Pretty much the same thing as Action delegate, but this one has a return type.
-Example:

```
public static void Main(string[] args)
{
    Func<string, bool> logFunc;
    if (args[0] == "EventLog") logFunc = LogToEventLog;
    else logFunc = LogToFile;
    bool status = logFunc("Log Message");
}

private static bool LogToEventLog(string message) { /* log */ }

private static bool LogToFile(string message) { /*log */ }
```

Using Func<T,TResult>

Func<T,TResult> supports one or more parameters (T) and returns a value (TResult).

-The last type in the generics list is the return type.