

Delegates and Lambdas



Filip Ekberg

Principal Consultant & CEO

@fekberg | fekberg.com

Using a Delegate in C#

```
var read = Console.ReadLine;
```

```
var input = read();
```



Using a Delegate in C#

```
var read = Console.ReadLine;
```

```
var input = read();
```



Point to the method **Console.ReadLine()**



Example: Passing a Delegate to a Method

Replace or extend
functionality in runtime

Receive a callbacks when
operation completes



Delegate

“A delegate is a type that represents references to methods with a particular parameter list and return type.”



Overview



Delegate keyword

Action and Action<T>

Func and Func<T>

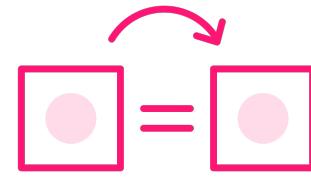
Lambdas



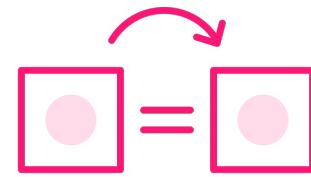
Creating a Delegate



What is a Delegate?



Defines the **required method signature**



A **reference type** used to **reference** a **method**



Defined using the **delegate keyword**



Creating a Delegate

```
delegate Order Buy    (Item item, int quantity);
```



Creating a Delegate

```
delegate Order Buy (Item item, int quantity);
```



Define the **required method signature:**
Return type and parameters



Creating a Delegate

```
delegate Order Buy    (Item item, int quantity);  
  
Order AddToCart    (Item item, int quantity) { ... }  
  
ProcessedOrder BuyNow (Item item, int quantity) { ... }
```



Creating a Delegate

```
delegate Order Buy      (Item item, int quantity);  
Order   AddToCart  
ProcessedOrder BuyNow (Item item, int quantity) { ... }
```

Return type match (ProcessedOrder inherits from Order)



Creating a Delegate

```
delegate Order Buy    (Item item, int quantity);  
  
Order AddToCart    (Item item, int quantity) { ... }  
  
ProcessedOrder BuyNow (Item item, int quantity) { ... }
```

Parameters match!



Return type cannot be less derived



Covariance and Contravariance

Parameters support
Contravariance

Return type support Covariance



Using a Delegate

```
delegate Order Buy(Item item, int quantity);
```

```
void BuyAll(IEnumerable<Item> items, Buy buy)
```

```
{
```

```
}
```



Using a Delegate

```
delegate Order Buy(Item item, int quantity);
```

```
void BuyAll(IEnumerable<Item> items, Buy buy)
```

```
}
```



Accept a method reference (delegate)



Using a Delegate

```
delegate Order Buy(Item item, int quantity);
```

```
void BuyAll(IEnumerable<Item> items, Buy buy)
{
    foreach(var item in items)
    {
        buy(item, 1);
    }
}
```



Using a Delegate

```
delegate Order Buy(Item item, int quantity);
```

```
void BuyAll(IEnumerable<Item> items, Buy buy)
{
    foreach(var item in items)
    {
        buy(item, 1); ← Could be either AddToCart or BuyNow
    }
}
```



**Delegates allow you to
reference a method and
later invoke it**

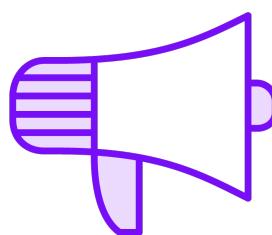


When to Use Delegates



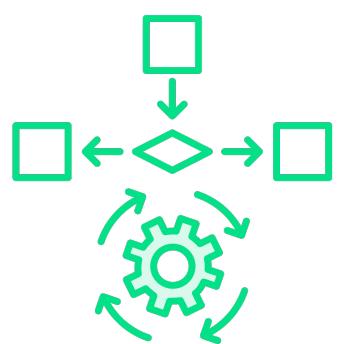
When you need a callback

Commonly used in the past when performing background work



Events

When a publisher notifies that an event has occurred, every subscriber is executed and can act on it



Extensibility

Allow a method to run additional functionality, or replace functionality during runtime



```
private void Initialize(Order order)
{
}

public void Process(Order order)
{
    // Run some code.. ←
    Initialize(order);
    // How do I produce a shipping l
}

}
```

**Whoever executes
Process can pass a
method reference to
determine which
method to run**

**Some libraries rely on delegates
to allow functionality to be
determined by the consumer**



Examples of Delegates in .NET

```
var orderNumbers = new[] { 1337, 35, 101, 30 };
```

```
orderNumbers.OrderBy(number => number);
```

```
Task.Run(HeavyOperator)  
    .ContinueWith(TheCallback);
```

```
void HeavyOperator() {}  
void TheCallback(Task task) {}
```



Examples of Delegates in .NET

```
var orderNumbers = new[ ] { 1337, 35, 101, 30 };  
orderNumbers.OrderBy(number => number);
```

```
Task.Run(HeavyOperator)  
.ContinueWith(TheCallback);
```

```
void HeavyOperator() { }  
void TheCallback(Task task) { }
```



Examples of Delegates in .NET

```
var orderNumbers = new[ ] { 1337, 35, 101, 30 };
```

```
orderNumbers.OrderBy(number => number);
```

```
Task.Run(HeavyOperator)  
.ContinueWith(TheCallback);
```

```
void HeavyOperator() { }  
void TheCallback(Task task) { }
```



Delegates are a powerful language feature!



If processing runs in the background, a delegate can be used to get updated information



Declaring a Delegate

```
// Outside a class  
public delegate void ProcessCompleted();
```

```
// Inside a class  
public class OrderProcessor  
{  
    public delegate void OrderInitialized();  
}
```

```
OrderProcessor.OrderInitialized onInitialized      = SendMessageToWarehouse;  
  
ProcessCompleted onCompleted                  = SendConfirmationEmail;
```



The method could be
decided during runtime!



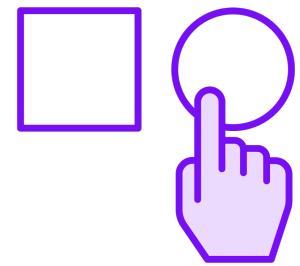
A normal **delegate** can be
invoked by anyone that has
access to it



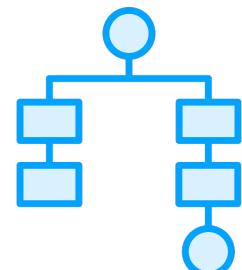
Benefits of This Approach



Delegate the implementation elsewhere



Determine which method to reference during runtime



Extensibility and flexibility can be achieved in many ways. Using delegates is one way.



Returning a Delegate

```
OrderProcessor.OrderInitialized GetOnInitialized()
```

```
{  
    return SendMessageToWarehouse;  
}
```

```
GetOnInitialized()(order);
```

```
GetOnInitialized()?.Invoke(order);
```



Returning a Delegate

```
OrderProcessor.OrderInitialized GetOnInitialized()
```

```
{  
    return SendMessageToWarehouse;  
}
```

```
GetOnInitialized()(order);
```

```
GetOnInitialized()?.Invoke(order);
```



Returning a Delegate

```
OrderProcessor.OrderInitialized GetOnInitialized()
{
    return SendMessageToWarehouse;
}
```

Invokes the method **SendMessageToWarehouse**

```
GetOnInitialized()(order);
GetOnInitialized()?.Invoke(order);
```



Multicast Delegates

“The multicast delegate contains a list of the assigned delegates. When the multicast delegate is called, it invokes the delegates in the list, in order.

Only delegates of the same type can be combined.”



Invoke a Multicast Delegate

```
OrderProcessor.ProcessCompleted chain = SendConfirmationEmail;  
chain += LogOrderProcessCompleted;  
chain += UpdateStock;  
  
// Invoke like a method  
chain(order);  
  
// Invoke through a method call  
chain.Invoke(order);
```



Invoke a Multicast Delegate

```
OrderProcessor.ProcessCompleted chain = SendConfirmationEmail;  
chain += LogOrderProcessCompleted;  
chain += UpdateStock;  
  
// Invoke like a method  
chain(order);  
  
// Invoke through a method call  
chain.Invoke(order);
```



Invoked in the order they were added



**Anyone with access to the
delegate can modify the
chain**



Delegate.Remove

“Removes the last occurrence of the invocation list of a delegate from the invocation list of another delegate”

Subtracting a delegate uses Delegate.Remove() internally.

Example:

```
chain -= logMethod;
```



Define This Method Using a Lambda

Accept one parameter
of type order

Return true or false depending on
if the order has any items



A lambda produces an anonymous function



Lambda



Lambda

(parameter1, parameter2)



Lambda

(parameter1, parameter2) =>



Lambda

```
(parameter1, parameter2) =>
```

Lambda operator



Lambda

```
(parameter1, parameter2) => parameter1 + parameter2;
```



Lambda

```
(parameter1, parameter2) => parameter1 + parameter2;
```



Lambda

```
(parameter1, parameter2) => parameter1 + parameter2;
```

```
(parameter1, parameter2) =>
{
    return parameter1 + parameter2;
};
```



Lambda

```
// Lambda Expression  
(parameter1, parameter2) => parameter1 + parameter2;  
  
(parameter1, parameter2) =>  
{  
    return parameter1 + parameter2;  
};
```



Lambda

```
// Lambda Expression  
(parameter1, parameter2) => parameter1 + parameter2;
```

```
// Lambda Statement  
(parameter1, parameter2) =>  
{  
    return parameter1 + parameter2;  
};
```



Inferring Types

```
OrderProcessor.OrderInitialized action = (order) =>
{
    return order.IsReadyForShipment;
};
```



Inferring Types

```
OrderProcessor.OrderInitialized action = (order) =>  
{  
    return order.IsReadyForShipment;  
};
```



The type is **inferred from** the delegate **OrderInitialized**



**Use a lambda statement for
anonymous functions that
does not return anything**



Removing an anonymous function from a multicast delegate is not simple



You should aim to keep
anonymous functions as
simple as possible to reduce
complexity



Explicit Types

```
OrderProcessor.OrderInitialized action = bool (order) =>
{
    return order.IsReadyForShipment;
};
```



Explicit Types

```
OrderProcessor.OrderInitialized action = bool (order) =>  
{  
    return order.IsReadyForShipment;  
};
```



Return type



Explicit Types

```
OrderProcessor.OrderInitialized action = bool (Order order) =>
{
    return order.IsReadyForShipment;
};
```



Explicit Types

```
OrderProcessor.OrderInitialized action = bool (Order order) =>  
{  
    return order.IsReadyForShipment;  
};
```



Parameter type



Attributes can be used with lambdas



Delegate expected?

You can use a lambda!



Lambda + LINQ

```
people.Where(person => person.Age > 20);
```



Lambda + LINQ

```
people.Where(person => person.Age > 20);
```



**Uses the delegate
to find matches**



Creating a generic delegate means that it is reusable



What Would This Delegate Look Like?

Generic Parameter

No Return Value



Creating a Generic Delegate



Creating a Generic Delegate

delegate



Creating a Generic Delegate

```
delegate void Action
```



Creating a Generic Delegate

```
delegate void Action<T>
```



Creating a Generic Delegate

```
delegate void Action<T>(T input);
```



Creating a Generic Delegate

```
delegate void Action<T>(T input);
```

```
Action<Order> action = SendConfirmationEmail;
```

```
Action<Order> action = (order) => { };
```



Creating a Generic Delegate with Return Value

```
delegate TResult Func<T, TResult>(T input);
```



Creating a Generic Delegate with Return Value

```
delegate TResult Func<T, TResult>(T input);
```



Add one more generic type



Creating a Generic Delegate with Return Value

```
delegate TResult Func<T, TResult>(T input);  
  
Func<Order, bool> func = SendMessageToWarehouse;  
  
Func<Order, bool> func = (order) => order.IsReadyForShipment;
```



**Action<T> and Func<T, TResult> are
already a part of .NET!**



Delegate

```
delegate bool OrderInitialized(Order order);
```



Delegate

```
delegate bool OrderInitialized(Order order);
```



All methods that return **bool** with a parameter
of type **order** (alternatively its base class if any)
will match this delegate



Delegate

```
delegate bool OrderInitialized(Order order);
```

```
void Process(OrderInitialized onInitialized)  
{ ... }
```



Delegate

```
delegate bool OrderInitialized(Order order);
```

```
void Process(OrderInitialized onInitialized)  
{ ... }
```



Use like any other reference type



A delegate is simply a method reference and has a very small memory footprint



Using a Delegate

```
void Process(OrderInitialized onInitialized)  
{ ... }
```



Using a Delegate

```
void Process(OrderInitialized onInitialized)  
{ ... }
```



**Accepts a method reference to any method
matching the given delegate**



Using a Delegate

```
void Process(OrderInitialized onInitialized)  
{ ... }  
  
Process(SendMessageToWarehouse);
```



Using a Delegate

```
void Process(OrderInitialized onInitialized)  
{ ... }  
  
Process(SendMessageToWarehouse);  
  
Process(order => order.IsReadyForShipment);
```



Using a Delegate

```
void Process(OrderInitialized onInitialized)
```

```
{ ... }
```

```
Process(SendMessageToWarehouse);
```

```
Process(order => order.IsReadyForShipment);
```



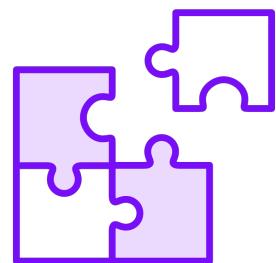
**Anonymous method defined with a lambda.
This results in a delegate that matches OrderInitialized**



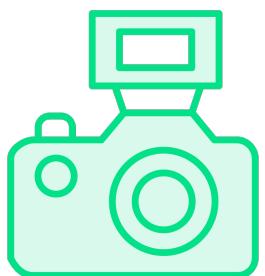
Benefits of Using a Lambda



Makes the code easier to read



The logic is defined in-place and the intent is clearly communicated



Can capture local variables



Consuming a Delegate

```
void Process(OrderInitialized onInitialized)  
{  
    var result = onInitialized();  
  
}  
}
```



Consuming a Delegate

```
void Process(OrderInitialized onInitialized)  
{  
    var result = onInitialized();  
  
}  
Could throw null a reference exception!
```



Consuming a Delegate

```
void Process(OrderInitialized onInitialized)  
{  
    var result = onInitialized?.Invoke();  
}
```



Avoid using BeginInvoke and EndInvoke





**Curious about asynchronous
programming?**

Asynchronous Programming in C#

Filip Ekberg

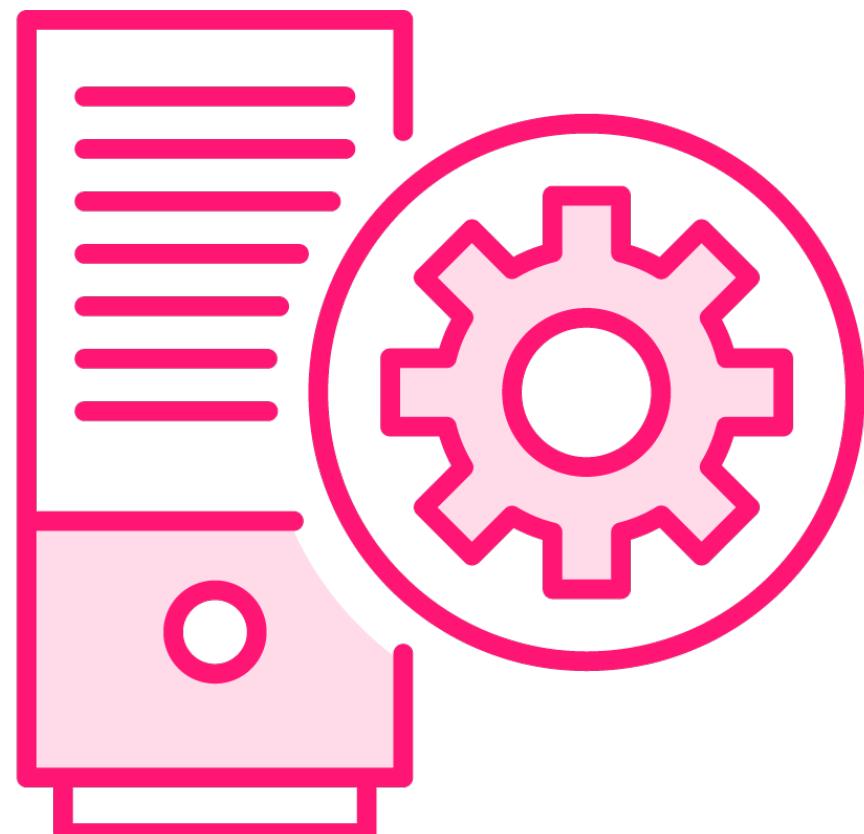


Using Func and Action

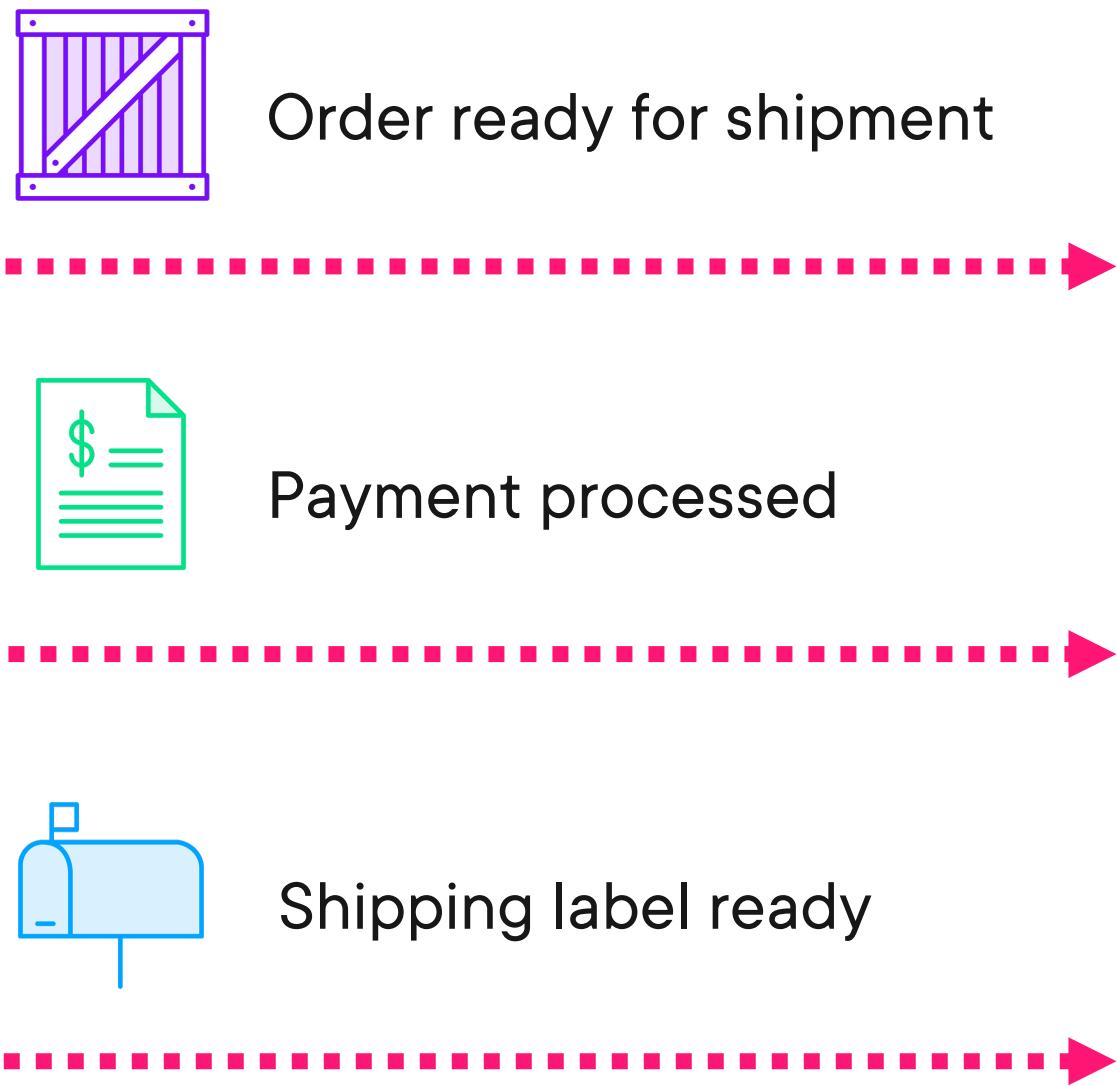
```
void Process(Func<Order, bool> onInitialized,  
            Action callback)  
{  
    var result = onInitialized?.Invoke();  
  
    callback?.Invoke();  
}
```



Broadcasting Events



Order Processor



Subscribers of the events



Up Next:

Events

