Published Apr 07, 2017 | Alfredo Ball

Understanding C# Delegates The Easy Way : Part 1 (Basics)

I wanted to write this post because looking back on when I was learning this concept, I realize that some people just like to sound smart.

They'd start out by saying, "This is a difficult concept for many people", which caused my mind to become demoralized and the difficulty became a self fullfilling prophecy.

Metaphors to the rescue! What use is an unfamiliar term in an explanation? So I will build a bridge to the concept by using already familiar terms and concepts within a metaphor.

Remember, it's very simple.

Suppose Bob knows how to paint portraits, but he's sick today, so he calls up Steve to do it since they are both qualified. They work at the same place and never work on the same day.

Bob gives Steve the customer photo to paint as a portrait and effectively DELEGATES the responsibility to Steve. Steve possesses the method of painting portraits. Bare in mind that Bob and Steve are from different Art schools so they have their own style.

In C# a method is defined by a Name, Input Parameters, Result, and Code Logic.

Forget the Code Logic part for now. Input Parameters, and Result are called the METHOD SIGNATURE, but not the Name. Each Input Parameter has a type, like a basic integer, string, or a non-primitive type such as a custom class (Product, Person, Vehicle, etc...)

That's a basic method. But what if we had a method called PaintPortrait. Well we might have an input argument type of Photograph and result type of Portrait.

Well the method's INTENT is well established, a photograph will be turned into a portrait so the result of the method could well be a class called Portrait which will include the painter's information, artistic style, and the portrait image itself.

Here is where delegates come into play. Bob and Steve both know how to paint, so each one could very well paint the portrait in their own way.

Considering this, we can declare the PaintPortrait method as a delegate type (no longer a method), which means it itself does not containt any logic for painting portraits, but instead Bob or Steve's methods can be associated to the PaintPortrait delegate so that either Bob's method or Steve's method will be invoked depending on which one is working that day.

The C# delegate declaration could look like this:

```
delegate Portrait PaintPortrait(Photograph photograph)
```

Photograph is the input argument, and Portrait is the result. But there's no logic for actually painting a portrait.

We can't assign a method to a delegate declaration directly because a delegate is just a definition of a method signature. We'll have to create a property of type PaintPortrait, which we can then set to a method reference (see below).

The catch is that their method signatures must match the signature of the PaintPortrait delegate type.

Bob's method could look like this:

```
public Portrait BobsPaintPortraitMethod(Photograph photograph)
{
    var portrait = new Portrait();



    return portrait;
}
```

Steve's method could like this:

```
public static Portrait StevesPaintPortraitMethod(Photograph photograph)
{
    var portrait = new Portrait();



    return portrait;
}
```

We can now create a property of type PaintPortrait that will hold a reference to either Bob or Steve's portrait painting method:

```
PaintPortrait PortraitPaintingProperty { get; set; }
```

And we can assign it a method reference like this:

```
PortraitPaintingProperty = BobsPaintPortraitMethod
```

or

```
PortraitPaintingProperty = StevesPaintPortraitMethod
```

Notice how we are not calling the BobsPaintPortraitMethod / StevesPaintPortraitMethod method, we are just assigning it as a reference to the PortraitPaintingProperty.

Now we can use the PortraintPaintingProperty property as if we were calling either Bob's or Steve's portraint painting method:

```
PortraitPaintingProperty(new Photograph())
```

This call will be routed to either Bob's or Steve's method.

That is the essence of delegation. There is more to know, like using a delegate type as a method's input argument type, but this is the core concept.
