Delegates

Delegates

- A delegate is an object that knows how to call a method
- Special reference type: reference to a method
- Behaves like a C++ function pointer
- C# delegates enable to add function “variables”

Why Delegates?

- Necessary indirection

  Delegates provide a way for .NET components to call your code without the .NET components having to know anything about your code beyond the method signature.

- Synchronous and Asynchronous Method Invocation

- Event Foundation

Delegate Type

- A delegate type defines the kind of method that delegate instance can call. Assign methods to delegate with matching signatures
    - Same parameter types
    - Same order of parameters
    - Same return type

Example

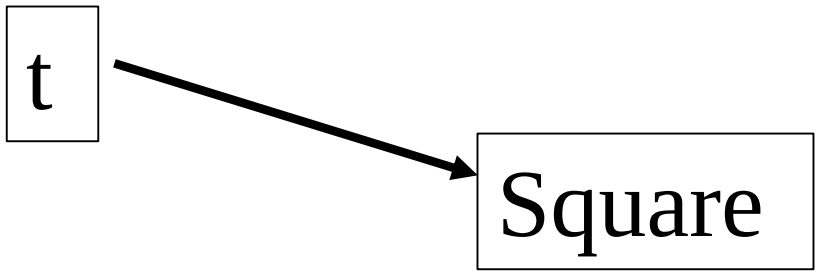
```
delegate int Transformer(int x); //defines a delegate type called Transformer
class Test
{
   static void Main()
   {
      Transformer t = Square; // Create delegate instance
      int result = t(3); // Invoke delegate
      Console.WriteLine (result); // 9
   }
      static int Square(int x) { return x * x; }
}
```

- The statement:

```
Transformer t = Square;
```

is shorthand for:

```
Transformer t = new Transformer (Square);
```



t references Square after its constructor has been invoked

- The expression: t(3) is shorthand for: t.Invoke(3)

Writing Plug-in Methods with Delegates

- A delegate variable is assigned a method at runtime. This is useful for writing plugin methods.

- Lamda2 example

Multicast Delegates

- All delegate instances have *multicast* capability: a delegate instance

can reference not just a single target method, but also a list of target methods

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

- Invoking d will now call both SomeMethod1 and SomeMethod2. Delegates are invoked in the order they are added.

- Use GetInvocationList to Obtain an Invocation List as an Array of Delegate References

- http://msdn.microsoft.com/en-us/library/ms173175.aspx
- http://msdn.microsoft.com/en-us/library/ms173171.aspx

Lambda Expressions

- A lambda expression has the form of :

(*parameters*) => *expression-or-statement-block*

- A lambda expression is an unnamed method written in place of a delegate instance. The compiler immediately converts the lambda expression to a delegate instance

```
delegate int Transformer (int i);
Transformer sqr = x => x * x;
Console.WriteLine (sqr(3)); // 9
```

http://msdn.microsoft.com/en-us/library/bb397687.aspx

- `x => x * x` // A simple expression that returns the square of its parameter // The type of parameter x is inferred from the context.

- `x => { return x * x ; }` // Semantically the same as the preceding expression, but using a C# statement block as a body rather than a simple expression

- `(int x) => x / 2` // A simple expression that returns the value of the parameter divided by 2. The type of parameter x is stated explicitly.

- `() => folder.StopFolding(0)` // Calling a method. The expression takes no parameters. The expression might or might not return a value.

- `(x, y) => { x++; return x / y; }` // Multiple parameters; the compiler infers the parameter types. The parameter x is passed by value, so the effect of the ++ operation is local to the expression.

Lambda Expression

- If a lambda expression takes parameters, you specify them in the parentheses to the left of the => operator. You can omit the types of parameters, and the C# compiler will infer their types from the context of the lambda expression.
- Lambda expressions can return values, but the return type must match that of the delegate they are being added to.
- The body of a lambda expression can be a simple expression or a block of C# code made up of multiple statements, method calls, variable definitions, and other code items.
- Variables defined in a lambda expression method go out of scope when the method finishes.
- A lambda expression can access and modify all variables outside the lambda expression that are in scope when the lambda expression is defined. Be very careful with this feature!