Delegates

Delegate

- Delegate – reference to method(s)
- Reference type containing
  - Method
  - Address
  - Link to an instance
  - Reference to an array of delegate chain

In the code

```
internal delegate void Feedback(Int32 value);
```

generates a class after compilation

```
internal class Feedback : System.MulticastDelegate
{
    // Constructor
    public Feedback(Object object, IntPtr method);
    // Method whose prototype is given in the source text
    public virtual void Invoke(Int32 value);
    // Methods providing an asynchronous callback
    public virtual IAsyncResult BeginInvoke(Int32 value, AsyncCallback callback,
                                            Object object);
    public virtual void EndInvoke(IAsyncResult result);
}
```

Method as delegate value

```
// Delegate as a class (declaration rules are the same)
delegate int MyDelegate(int x, int y);
static int SumMethod(int x, int y)
{
    return x + y;
}
...
MyDelegate d1 = SumMethod;
d1();
```

Adding Methods to a Delegate

- delegate can point to multiple methods that have the same signature and return type.
- all methods in the delegate fall into a special list - the invocation list or invocation list.
- when a delegate is called, all methods from this list are sequentially called.
- += operator is used to add methods to a delegate:

```
delegate void HelloWorld();
static void Hello()
{
    Console.Write("Hello ");
}
static void World()
{
    Console.WriteLine
    ("world!");
}
```

```
HelloWorld hello = Hello;
hello += World;
hello();

hello -= World;
hello?.Invoke();
```

```
Hello world!
Hello
```

Delegates as Method Parameters

```
static void DoOperation(int a, int b, Operation op)
{
    Console.WriteLine(op(a,b));
}
static int Add(int x, int y) => x + y;
static int Subtract(int x, int y) => x - y;
static int Multiply(int x, int y) => x * y;


DoOperation(5, 4, Add);         // 9
DoOperation(5, 4, Subtract);    // 1
DoOperation(5, 4, Multiply);    // 20

delegate int Operation(int x, int y);
```

Anonymous Methods

Allow to describe the body of a method without specifying a name

```
delegate int MyDelegate(int x, int y);

...

MyDelegate d2 = delegate(int x, int y) { return x + y; };
```

Lambda Expressions

- When declaring an anonymous method, the delegate keyword can be omitted
- And use ‘=>’ instead of return
- Like delegates, lambda expressions can be passed as method parameters

```
MyDelegate anonymous = delegate(int x, int y) { return x + y; };
MyDelegate lambda = (int x, int y) => x + y;
```

```
int Sum(int a, int b, MyDelegate func)
{
        return func(a, b);
}
Sum(1, 2, (int x, int y) => x + y);
```

Anonymous generic delegates

- Action - a family of delegates that do not return a value
- Func - a family of delegates that return a value
- Predicate - delegate returning bool

- Save time by eliminating the need to declare delegates with the same definition
- Actively used with extension methods and LINQ

```
Func<int, int, int> funcAndLambda = (x, y) => x + y;
```

Action

- Family of 17 non-returning delegates

```
public delegate void Action<T>(T obj);
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate void Action<T1, ..., T16>(T1 arg1, ..., T16 arg16);

Action<string, string> Hello = (x, y) => Console.WriteLine($"{x} {y}");
```

Func

- Family of 17 value-returning delegates

```
public delegate TResult Func<TResult>();

public delegate TResult Func<T, TResult>(T arg);

public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);

public delegate TResult Func<T1, T2, T3, TResult> (T1 arg1, T2 arg2, T3 arg3);

...

public delegate TResult Func<T1,..., T16, TResult> (T1 arg1, ..., T16 arg16);
```

```
 Func<int, int, int> funcAndLambda = (x, y) => x + y;
```

Closures

- A closure is a data structure for storing a function along with its environment
- A closure attached to a parent method has access to the members defined in the body of the parent method

```
public Person FindById(int id)
{
    return this.Find(delegate (Person p)
    {
        return (p.Id == id);
    });
}
```

https://medium.com/swlh/the-magic-of-c-closures-9c6e3fff6ff9

Capture in closure

- A variable captured in a closure extends the lifetime while the closure is alive
- It doesn't matter if it's a reference type or a value type.

```
var actions = new List<Action>();
foreach (var i in Enumerable.Range(1, 3))
{
    actions.Add(() => Console.WriteLine(i));
}

foreach (var action in actions)
{
    action();
}
// Before    C# 5:  3 3 3
// After     C# 5:  1 2 3
```

Events

- Events signal to the system that a specific action has taken place

```
class Account
{
        public delegate void AccountHandler(string message);
        public event AccountHandler? Notify;
        public int Sum { get; private set; }

        public Account(int sum) => Sum = sum;
        public void Put(int sum)
        {
                Sum += sum;
                Notify?.Invoke($"Account received: {sum}");   // Event call
        }
}

Notify += DisplayMessage;
Notify -= DisplayMessage;
```

EventArgs

- Often when an event occurs, the event handler needs to pass some information about the event

```
class AccountEventArgs
{
    public string Message{ get; }
    public int Sum { get; }
    public AccountEventArgs(string message, int sum)
    {
        Message = message;
        Sum = sum;
    }
}

Notify?.Invoke(this, new AccountEventArgs($"Account received {sum}", sum));
void DisplayMessage(Account sender, AccountEventArgs e) {…}
```