TEKNOLOGISK
INSTITUT

Module 12

Delegates, Events,
and Lambda Expressions

Introducing Delegates

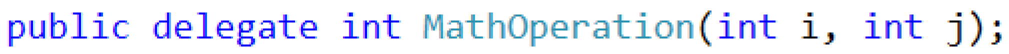
- We have covered values in C#
- We have covered references to objects in C#

- It is in fact also possible to construct type-safe references to methods
  - Or possibly a list of methods
- Thus method invocation is delegated to such an entities

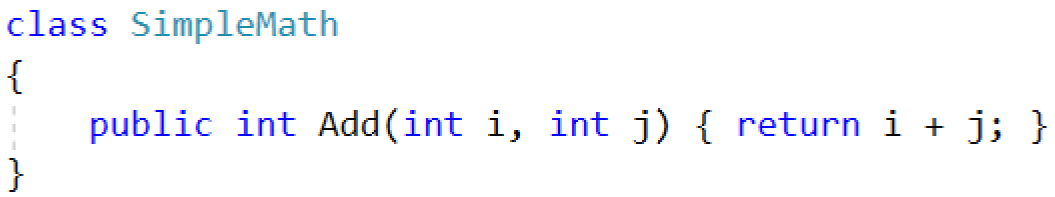
- These entities are called *Delegates* and form the basis for event-driven programming in .NET

Defining a Delegate

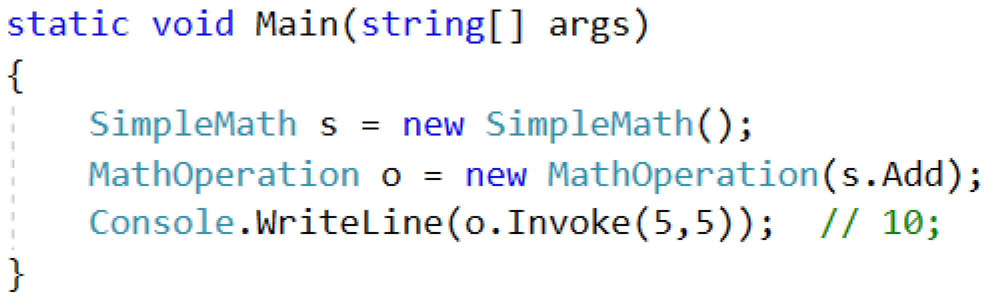
- Use the `delegate` keyword to define delegates
- Instances of this type are references to methods with this signature
- You can define delegates with any legal signature
- Delegates can reference both static and instance methods with the same syntax



```
public delegate int MathOperation(int i, int j);
```



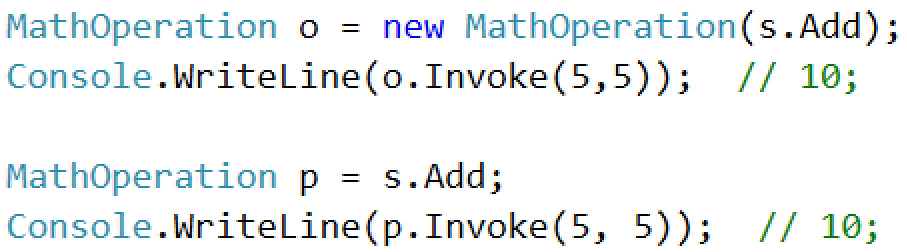
```
class SimpleMath
{
    public int Add(int i, int j) { return i + j; }
}
```



```
static void Main(string[] args)
{
    SimpleMath s = new SimpleMath();
    MathOperation o = new MathOperation(s.Add);
    Console.WriteLine(o.Invoke(5,5));  // 10;
}
```

Method Group Conversions

- This feature allows you to use delegates with the method name only



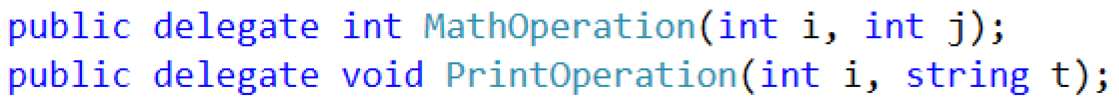
```
MathOperation o = new MathOperation(s.Add);
Console.WriteLine(o.Invoke(5,5));  // 10;

MathOperation p = s.Add;
Console.WriteLine(p.Invoke(5, 5));  // 10;
```

- This is still type-safe..!
- C# compiler just silently does the conversion for us
- Much more convenient, maintainable, and readable

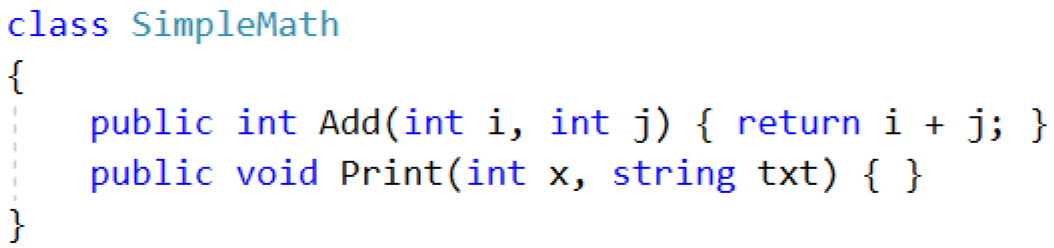
- Use this whenever you can!

Invoking a Delegate

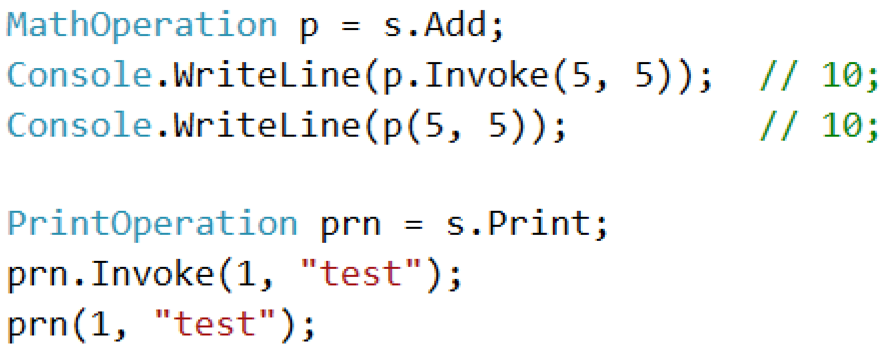
- A delegate can be invoked with the same syntax as method invocations
- And return values are used like conventional methods



```
public delegate int MathOperation(int i, int j);
public delegate void PrintOperation(int i, string t);
```



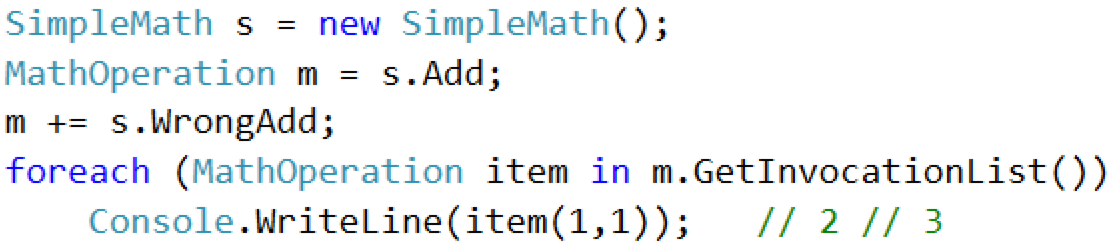
```
class SimpleMath
{
    public int Add(int i, int j) { return i + j; }
    public void Print(int x, string txt) { }
}
```



```
MathOperation p = s.Add;
Console.WriteLine(p.Invoke(5, 5));  // 10;
Console.WriteLine(p(5, 5));         // 10;

PrintOperation prn = s.Print;
prn.Invoke(1, "test");
prn(1, "test");
```

Multicasting Delegates

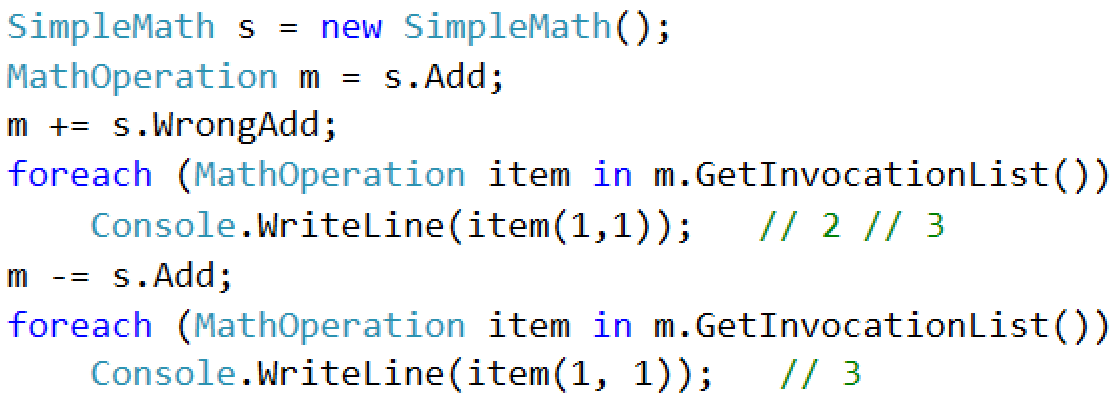
- C# delegates are in fact multicasting
- Each delegate actually references a *list of methods* to be invoked – not just a single method!
- It has an internal invocation list



```
SimpleMath s = new SimpleMath();
MathOperation m = s.Add;
m += s.WrongAdd;
foreach (MathOperation item in m.GetInvocationList())
    Console.WriteLine(item(1,1));   // 2 // 3
```

Removing Targets from Invocation List

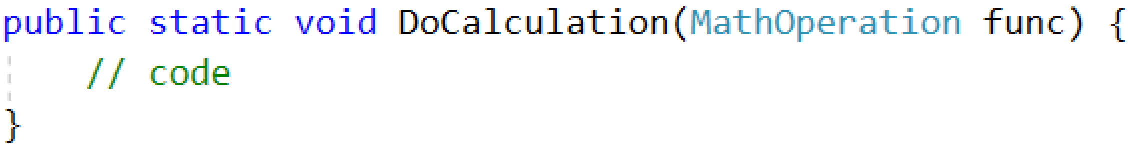
- As demonstrated earlier, the += operator adds a target to the invocation list.
- In a similar vein, the -= operator removes targets from the invocation list



```
SimpleMath s = new SimpleMath();
MathOperation m = s.Add;
m += s.WrongAdd;
foreach (MathOperation item in m.GetInvocationList())
    Console.WriteLine(item(1,1));   // 2 // 3
m -= s.Add;
foreach (MathOperation item in m.GetInvocationList())
    Console.WriteLine(item(1, 1));   // 3
```

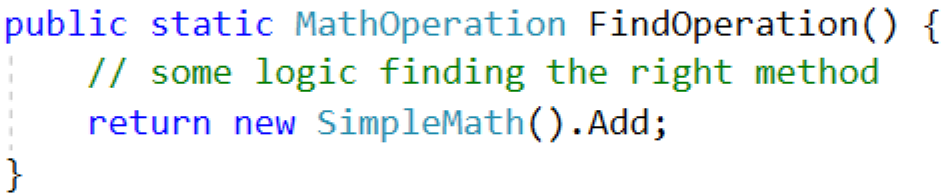
Delegates as Parameters

- Delegates can be supplied as parameters to methods



```
public static void DoCalculation(MathOperation func) {
    // code
}
```

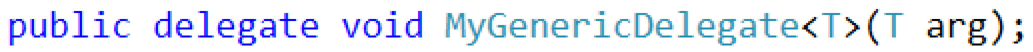
- Similarly, delegates can be returned from methods



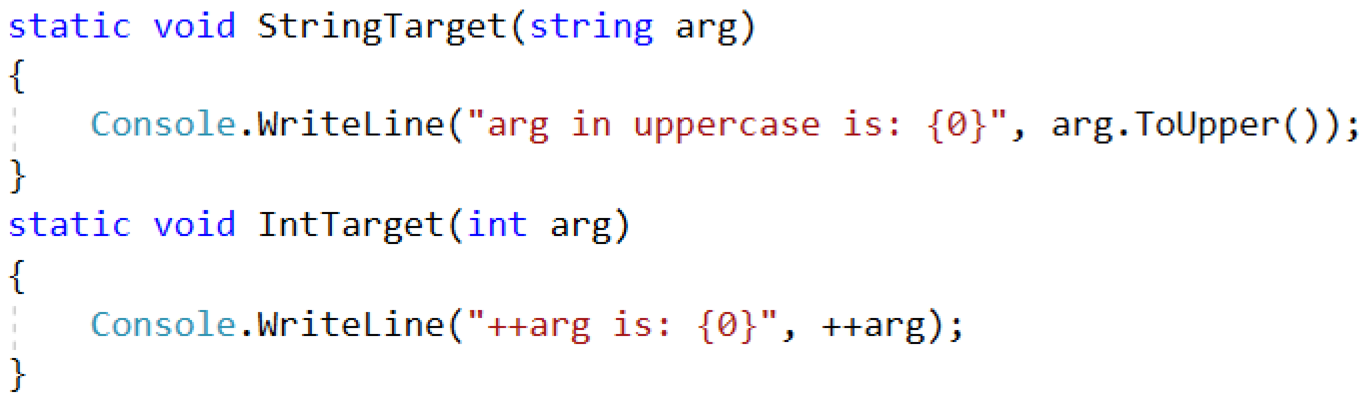
```
public static MathOperation FindOperation() {
    // some logic finding the right method
    return new SimpleMath().Add;
}
```

Generic Delegates

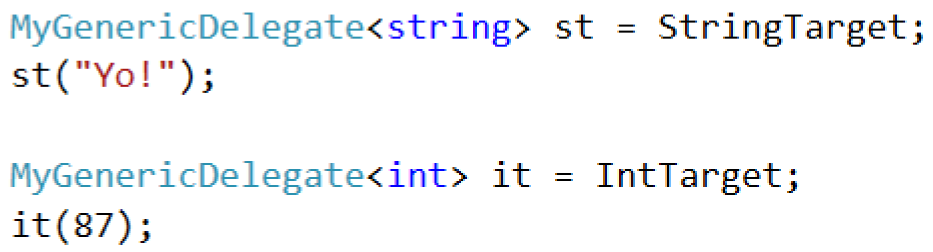
- Delegates can be generic
  - Good example is EventHandler<T>



```
public delegate void MyGenericDelegate<T>(T arg);
```



```
static void StringTarget(string arg)
{
    Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}
static void IntTarget(int arg)
{
    Console.WriteLine("++arg is: {0}", ++arg);
}
```

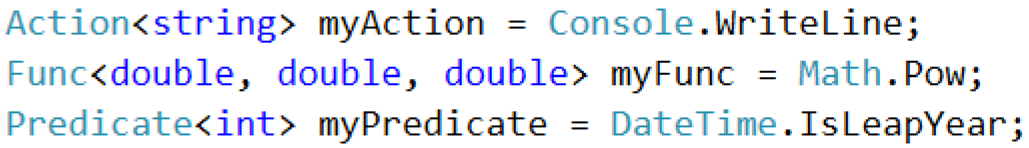


```
MyGenericDelegate<string> st = StringTarget;
st("Yo!");

MyGenericDelegate<int> it = IntTarget;
it(87);
```

Predefined delegates

- There are predefined generic delegates in the framework

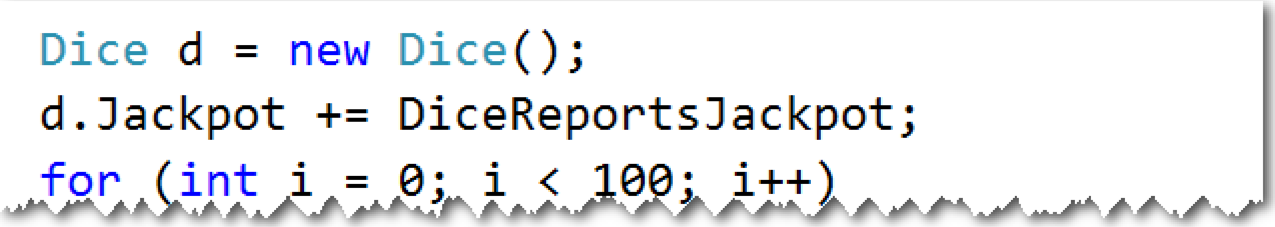
- Action<[arguments]>
  - Represents a void
- Func<[arguments],[returntype]>
  - Represents a function with a return value
- Predicate<[type to compare]>
  - Represents a function that returns a bool



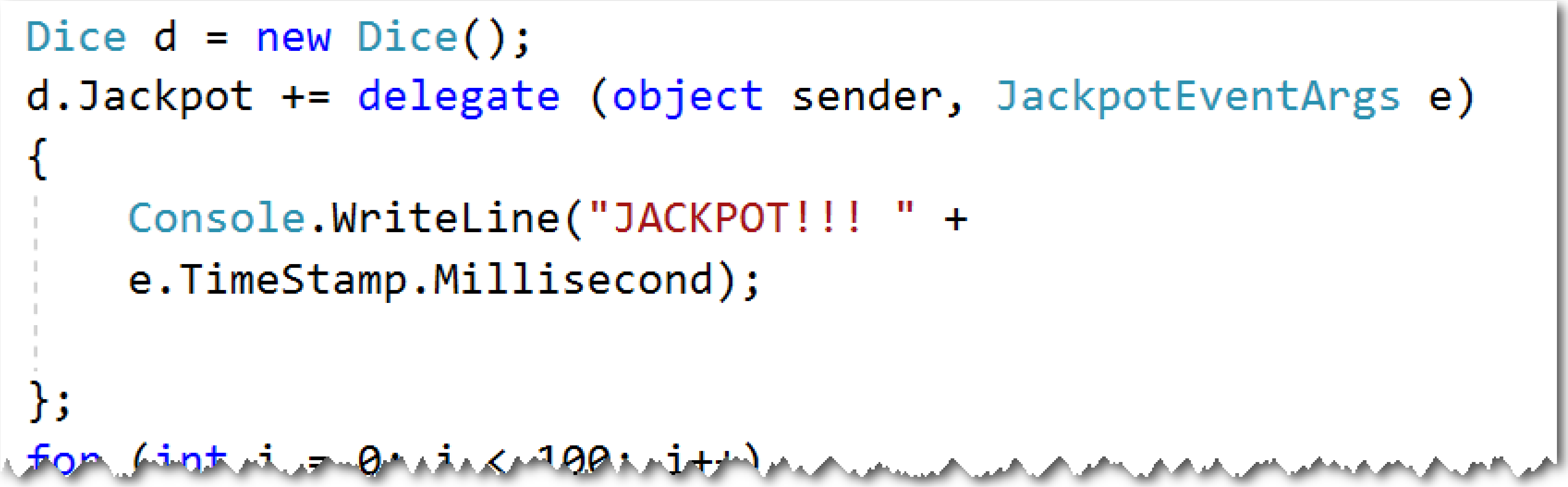
```
Action<string> myAction = Console.WriteLine;
Func<double, double, double> myFunc = Math.Pow;
Predicate<int> myPredicate = DateTime.IsLeapYear;
```

Defining Anonymous Methods

- When method code is only used once, the method code can be inlined as a delegate in an *anonymous method*



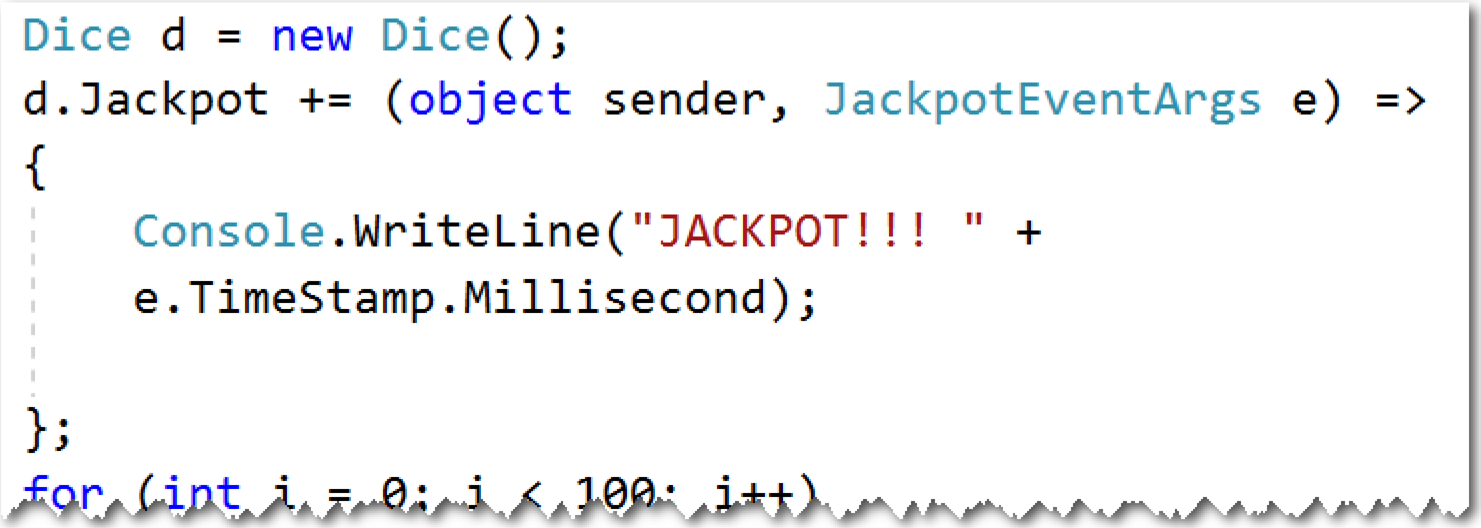
```
Dice d = new Dice();
d.Jackpot += DiceReportsJackpot;
for (int i = 0; i < 100; i++)
```



```
Dice d = new Dice();
d.Jackpot += delegate (object sender, JackpotEventArgs e)
{
    Console.WriteLine("JACKPOT!!! " +
    e.TimeStamp.Millisecond);

};
for (int i = 0; i < 100; i++)
```

Defining Lambda Expressions

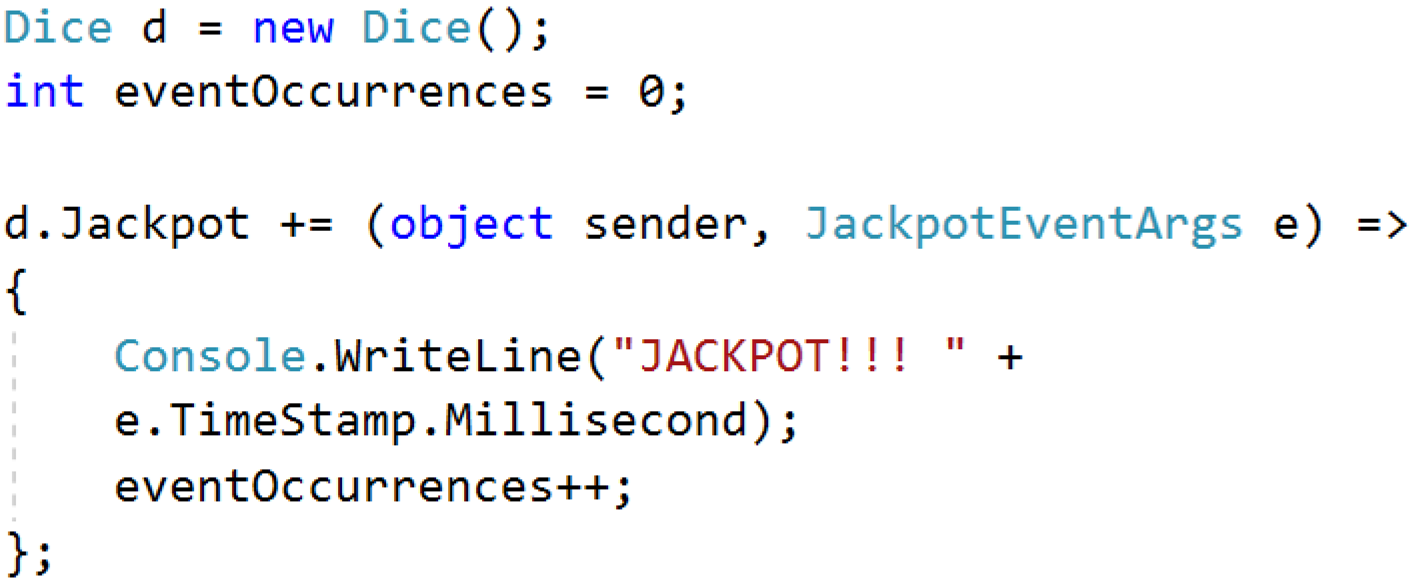
- Lambda expressions are a compact notation of the form
  - They are just short-hands for anonymous methods



```
Dice d = new Dice();
d.Jackpot += (object sender, JackpotEventArgs e) =>
{
    Console.WriteLine("JACKPOT!!! " +
    e.TimeStamp.Millisecond);

};
for (int i = 0; i < 100; i++)
```

Accessing Outer Variables

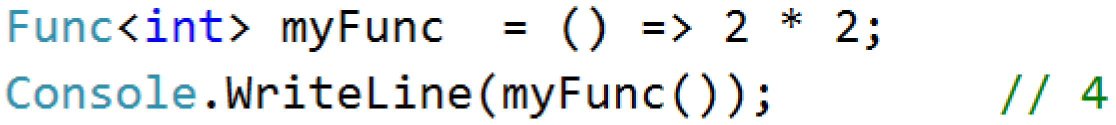
- Anonymous methods can access *“outer variables”* outside the anonymous method itself



```
Dice d = new Dice();
int eventOccurrences = 0;

d.Jackpot += (object sender, JackpotEventArgs e) =>
{
    Console.WriteLine("JACKPOT!!! " +
    e.TimeStamp.Millisecond);
    eventOccurrences++;
};
```

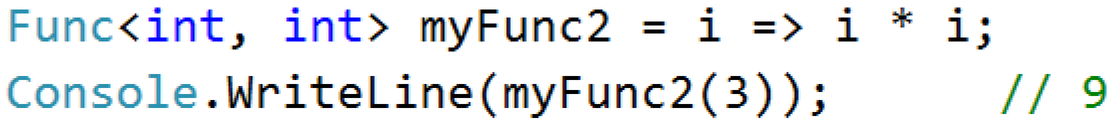
Expressions with
Zero or One Parameters

- Lambda expressions could be parameterless



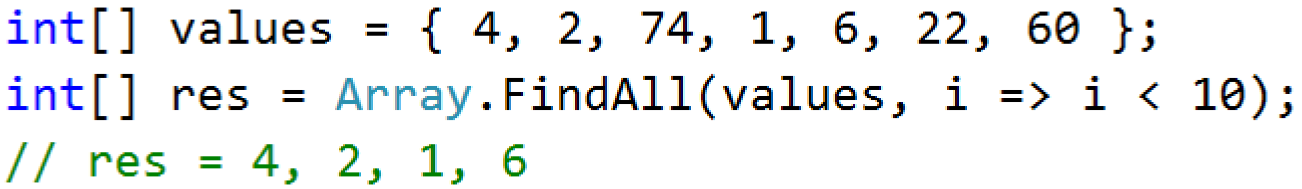
```
Func<int> myFunc  = () => 2 * 2;
Console.WriteLine(myFunc());        // 4
```

- The parentheses can be left out altogether if exactly one parameter



```
Func<int, int> myFunc2 = i => i * i;
Console.WriteLine(myFunc2(3));      // 9
```

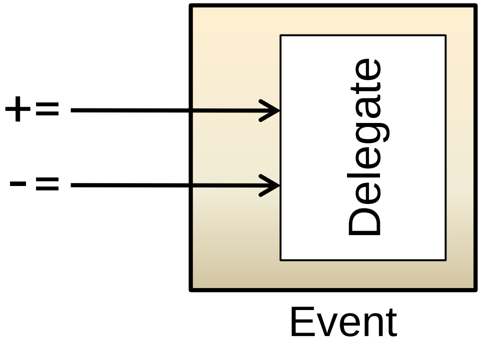
- `Array.FindAll()` works perfectly with predicates



```
int[] values = { 4, 2, 74, 1, 6, 22, 60 };
int[] res = Array.FindAll(values, i => i < 10);
// res = 4, 2, 1, 6
```

Introducing Events

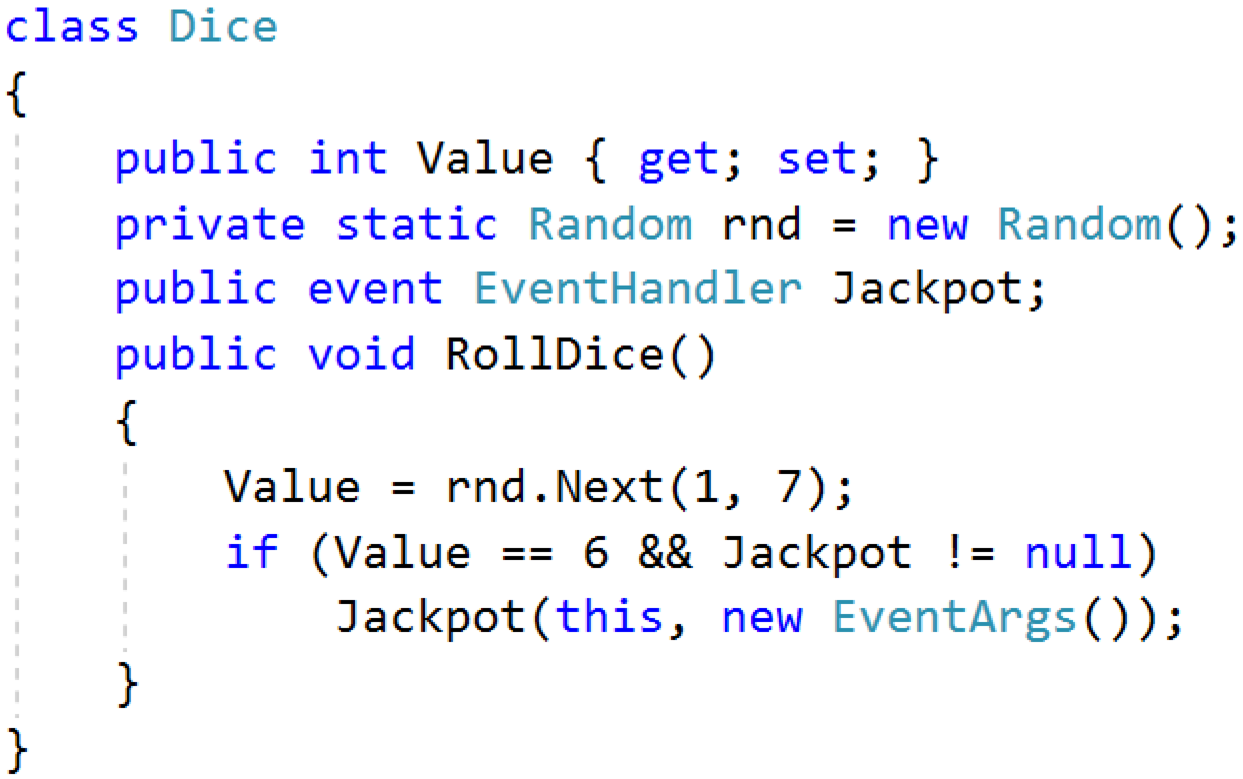
- Modern programming is event-driven
  - Occurrences of events trigger certain actions
  - Publisher-Subscriber scenario
  - E.g. button clicks in Windows applications
- Can delegates facilitate this kind of scenario?
  - Well… Yes, but…



- Events provide a convenient wrapper around delegates!

The event Keyword

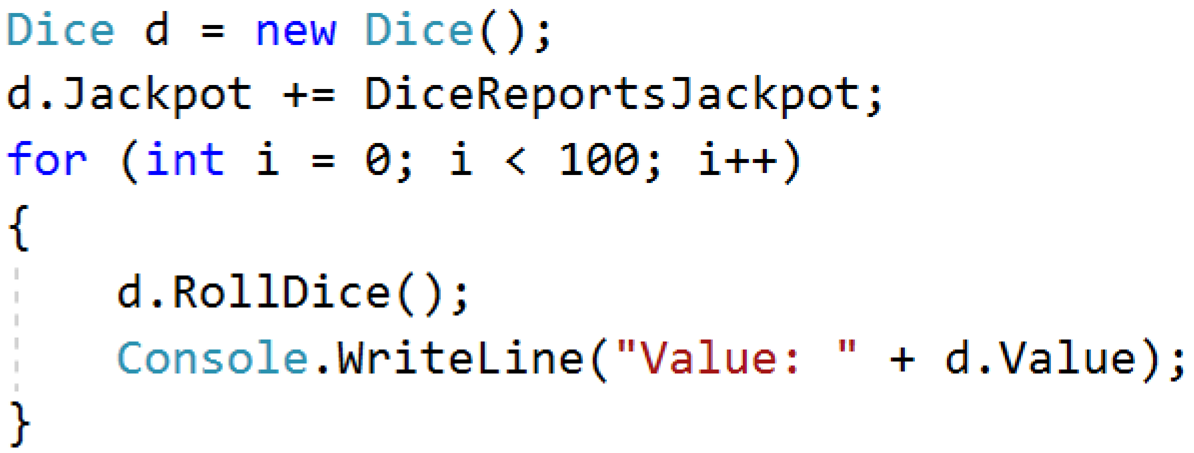
- Events are constructed from some delegate signature with the event keyword
  - EventHandler and EventHandler<> is the commonly used delegate



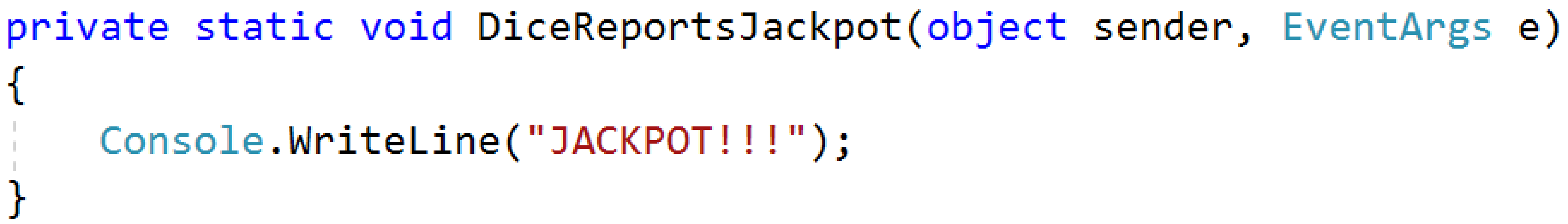
```
class Dice
{
    public int Value { get; set; }
    private static Random rnd = new Random();
    public event EventHandler Jackpot;
    public void RollDice()
    {
        Value = rnd.Next(1, 7);
        if (Value == 6 && Jackpot != null)
            Jackpot(this, new EventArgs());
    }
}
```

The event Keyword

- Subscribers can now subscribe and unsubscribe to the event with += and -=



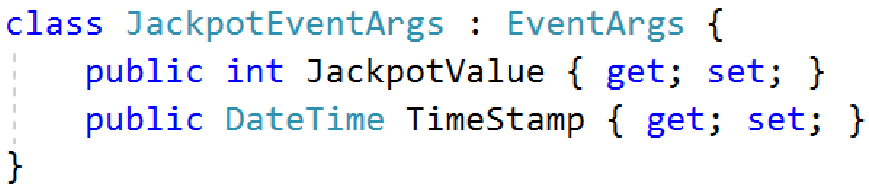
```
Dice d = new Dice();
d.Jackpot += DiceReportsJackpot;
for (int i = 0; i < 100; i++)
{
    d.RollDice();
    Console.WriteLine("Value: " + d.Value);
}
```



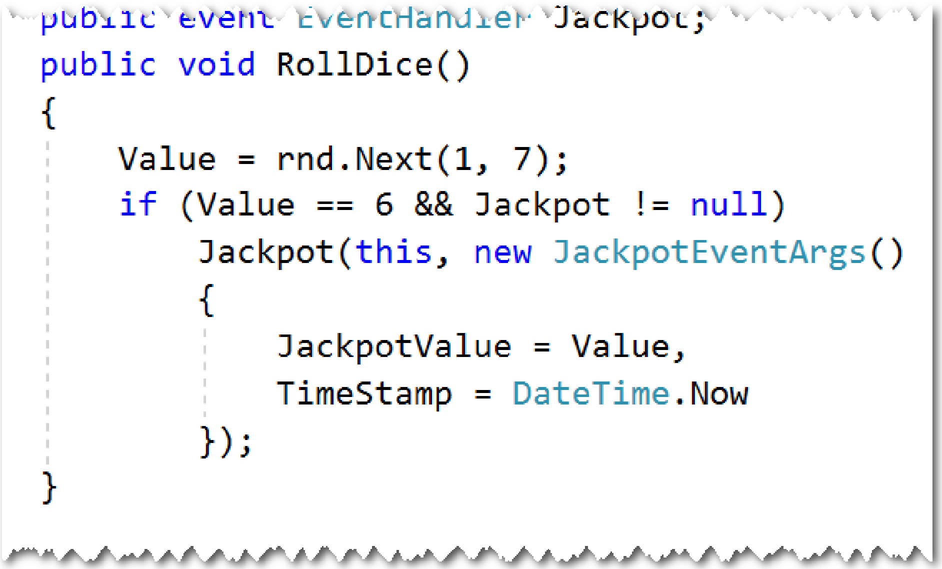
```
private static void DiceReportsJackpot(object sender, EventArgs e)
{
    Console.WriteLine("JACKPOT!!!");
}
```

Event Arguments

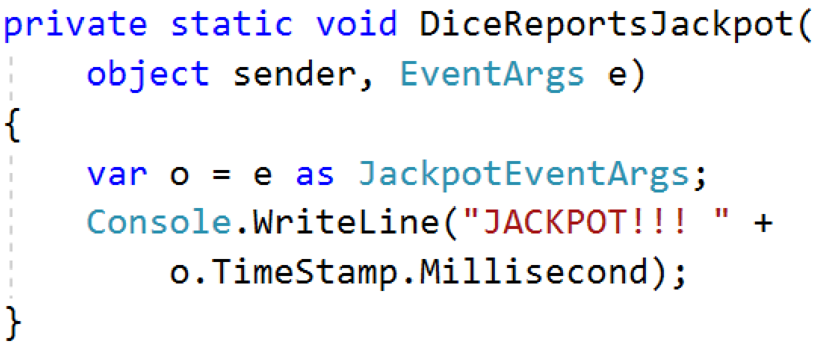
- The recommended event pattern is that the parameters consists of
  - object raising the event
  - Subclass of System.EventArgs
- The event info class name is to be called *event name* + *“EventArgs”*



```
class JackpotEventArgs : EventArgs {
    public int JackpotValue { get; set; }
    public DateTime TimeStamp { get; set; }
}
```



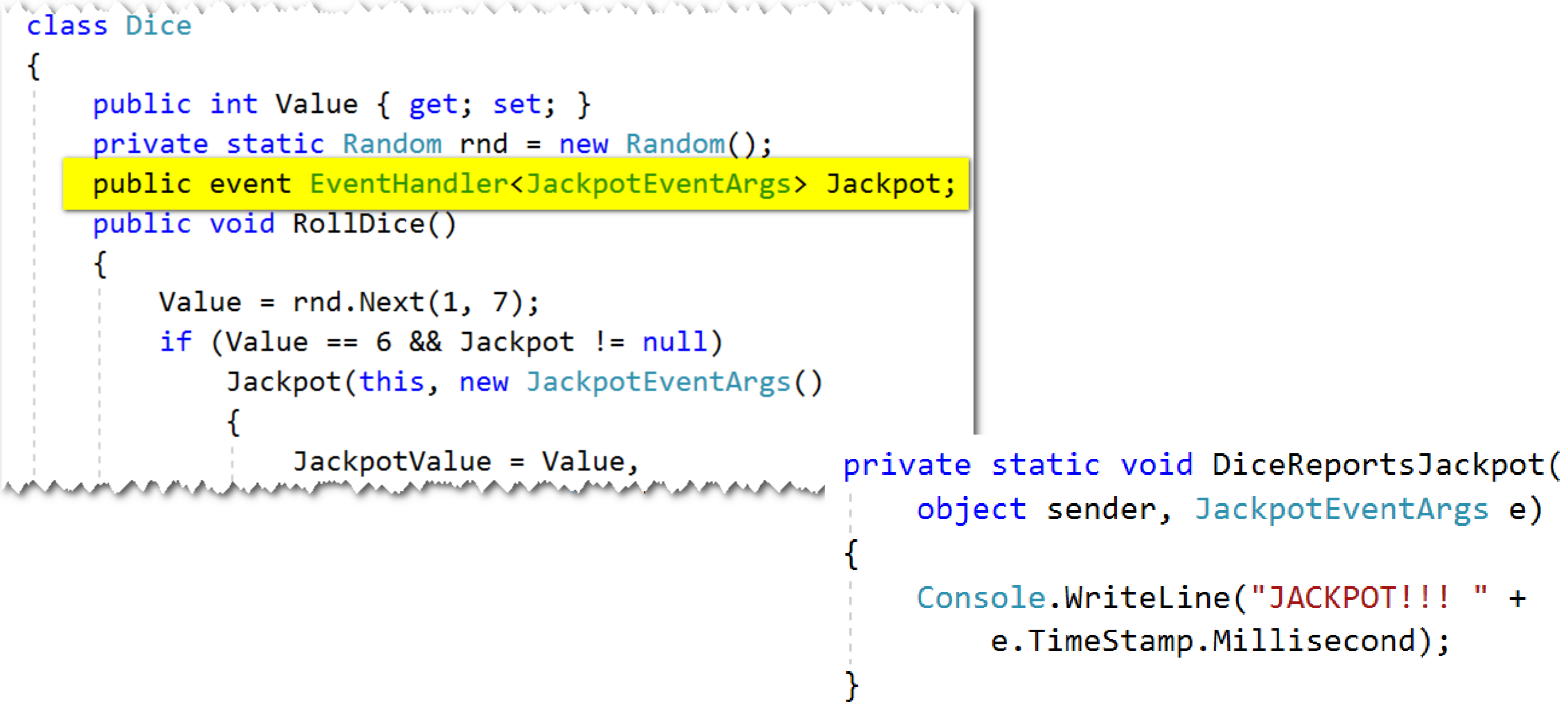
```
public event EventHandler Jackpot;
public void RollDice()
{
    Value = rnd.Next(1, 7);
    if (Value == 6 && Jackpot != null)
        Jackpot(this, new JackpotEventArgs()
        {
            JackpotValue = Value,
            TimeStamp = DateTime.Now
        });
}
```



```
private static void DiceReportsJackpot(
    object sender, EventArgs e)
{
    var o = e as JackpotEventArgs;
    Console.WriteLine("JACKPOT!!! " +
        o.TimeStamp.Millisecond);
}
```

The `EventHandler<T>` Delegate

- Since all event delegates preferably obey the same pattern, this is captured in a generic eventhandler delegate which you should always use!

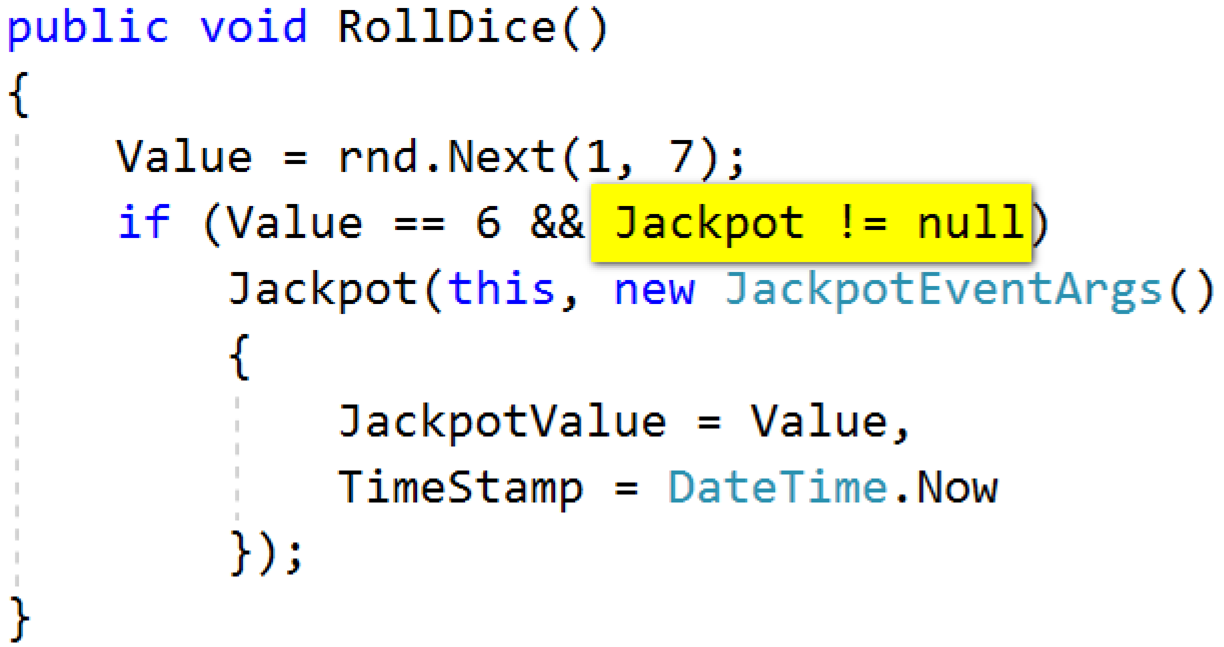


```
class Dice
{
    public int Value { get; set; }
    private static Random rnd = new Random();
    public event EventHandler<JackpotEventArgs> Jackpot;
    public void RollDice()
    {
        Value = rnd.Next(1, 7);
        if (Value == 6 && Jackpot != null)
            Jackpot(this, new JackpotEventArgs()
            {
                JackpotValue = Value,
```

```
private static void DiceReportsJackpot(
    object sender, JackpotEventArgs e)
{
    Console.WriteLine("JACKPOT!!! " +
        e.TimeStamp.Millisecond);
}
```

Raising Events

- Events are raised by treating the event as the underlying delegate
- Remember to check whether event is null
  - This checks if there are any subscribers



```
public void RollDice()
{
    Value = rnd.Next(1, 7);
    if (Value == 6 && Jackpot != null)
        Jackpot(this, new JackpotEventArgs()
        {
            JackpotValue = Value,
            TimeStamp = DateTime.Now
        });
}
```