Delegates and Events

C# Programming

Delegates and Events

- C# is enrich with three most innovative features: Delegates, Events and Lambda Expression
- A delegate provides a way to encapsulate a method.
- An event is a notification that some action has occurred.
- Delegates and events are related because an event is built upon a delegate.
- The lambda expression is a relatively new syntactic feature that offers a streamlined, yet powerful way to define what is, essentially, a unit of executable code.

Recalling 'C'

- What are Pointers to Functions in C?

```
main( )
{
   void display( ) ;
   void ( *p )( ) ;
   display( ) ;        // One Way
   p = display ;
   ( *p )( ) ;         // One More Way
}
void display( )
{
   printf ( "Hello" ) ;
}
```

What is a Delegate?

- A delegate is a C# language element that allows you to reference a method.
- Delegate is an object, using which one can invoke the functionality of other object without knowing its Class Name or Method Name
- Delegates are like function pointers in 'C', but are type safe
- Delegates are used for callback implementation
- Delegates is an object having reference to a method having same signature as specified in Delegate declaration

Inside Delegates

- It is a mechanism by which methods can be passed as method parameters instead of data.
- Like class and interface it is also a type and its references are also created and instantiated.
- A method can call another method through the delegate that is passed to it . This is known as an asynchronous callback. This is a common method of notifying a caller when a long process is completed

Where delegates are used?

- Delegates can be used in multithreaded programming to give the starting point of the thread execution.
- Generic library classes that has generic sort, search methods etc. use delegates to get the comparison method for user-defined object.
- Event handling mechanism also need to know which method to call when the event occurs.
- Delegates can be chained together so that multiple methods can be grouped into a single event

Presented by

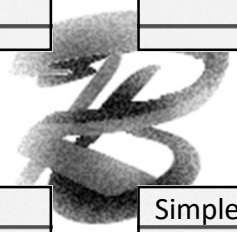
Ranjan Bhatnagar

Callback Mechanism

- Callback mechanisms can be used especially for graphical user interfaces in that controls need to invoke external methods under the correct circumstances.
- Using callbacks, programmers were able to configure one function to report back to (call back) another function in the application.
- Specifically, a delegate maintains three important pieces of information:
  - The address of the method on which it makes calls
  - The parameters (if any) of this method
  - The return type (if any) of this method

Delegate class

- Delegate types are derived from System.Delegate class in the .NET Framework.
- Delegate is an abstract class. However, only the system and compilers can derive explicitly from the Delegate class .
- After delegate is created, its instances can be created.
- Delegate types are implicitly sealed.



Syntax

- Defining

access-modifiers delegate return-type method-name (parameter-list)

- access-modifiers can be public, protected, internal, and private depending on the context in which the delegate declaration occurs, some of these modifiers may not be permitted
- The instantiated delegate is an object and so it can be passed as a parameter, or assigned to a property.

Simple Example

```
using System;
class Program
{
    static void Add(int i, int j)
    {
        Console.WriteLine("Sum :" + (i + j));
    }
    static void Sub(int i, int j)
    {
        Console.WriteLine("Sub :" + (i - j));
    }
    public delegate void reftomethod(int i, int j);
    static void Main(string[] args)
    {
        Add(10, 20);
        reftomethod r1 = Add;
        r1(10, 20);
        reftomethod r2 = Sub;
        r2.Invoke(10, 20);
    }
}
```

Delegate Base Classes

- When you build a type using the C# delegate keyword, you indirectly declare a class type that derives from System.MulticastDelegate.
- This class provides descendants with access to a list that contains the addresses of the methods maintained by the delegate object as well as several additional methods and a few overloaded operators to interact with the invocation list.
- you can never directly derive from these base classes in your code, when you use the delegate keyword, you have indirectly created a class that “is-a” MulticastDelegate.

Members of Base Classes

- **Method** This property returns a System.Reflection.MethodInfo object that represents details of a static method.
- **Target** If the method to be called is defined at the object level, Target returns an object that represents the method maintained by the delegate. If the value returned from Target equals null, the method to be called is a static member.
- **Combine()** This static method adds a method to the list maintained by the delegate. += can also be used.
- **GetInvocationList()** This method returns an array of System.Delegate objects, each representing a particular method that may be invoked.
- **Remove() / RemoveAll()** These static methods remove a method (or all methods) from the delegate’s invocation list. You can also use overloaded -= operator.

Presented by

Ranjan Bhatnagar

Multicast delegates

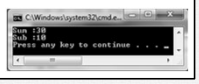
- Delegate in the previous slides were used to call only a single method.
- Delegates that can be used to call multiple methods are called multicast delegates.
- Multiple objects can be assigned to one delegate instance by using the + operator and to remove a component delegate use the - operator. += and -= are also overloaded.
- In other words multicast delegates calls a sequence of methods in the specified order.
- If one of the methods in the sequence throws an exception, the iteration stops there!

Multicast delegates signature and class

- Only delegates of the same type can be combined.
- The multicast signature should generally return void; otherwise result of the call will be the return value of the last method invoked.
- The multicast delegate types derive from System.MulticastDelegate which is inherited from System.Delegate .
- Like Delegate, this also can be derived by the Compilers and other tools only.
- All delegates are derived from MulticastDelegate.
- The delegate class created will contain a constructor (whose parameter will match the number of parameter delegate takes) and an Invoke() method. Delegate method call can be made through Invoke method also.

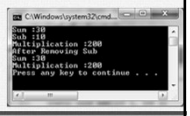
Multicast Delegate Adding Method

```
using System;
class Program
{
    static void Add(int i, int j)
    {
        Console.WriteLine("Sum :" + (i + j));
    }
    static void Sub(int i, int j)
    {
        Console.WriteLine("Sub :" + (i - j));
    }
    public delegate void reftomethod(int i, int j);
    static void Main(string[] args)
    {
        reftomethod multicast = Add;
        multicast += Sub;
        multicast(20, 10);
    }
}
```



Multicast Delegate Removing Method

```
using System;
class Program
{
    static void Add(int i, int j)
    {
        Console.WriteLine("Sum :" + (i + j));
    }
    static void Sub(int i, int j)
    {
        Console.WriteLine("Sub :" + (i - j));
    }
    static void Multiply(int i, int j)
    {
        Console.WriteLine("Multiplication :" + (i * j));
    }
    public delegate void reftomethod(int i, int j);
    static void Main(string[] args)
    {
        reftomethod multicast = Add;
        multicast += Sub;
        multicast += Multiply;
        multicast(20, 10);
        Console.WriteLine("After Removing Sub");
        multicast -= Sub;
        multicast(20, 10);
    }
}
```



Another Delegate Example

```
using System;
delegate void Print();
class Money
{
    protected uint note;
    protected uint coin;
    public Money(uint n, uint c)
    {
        this.note = n;
        this.coin = c;
    }
}
class Rupee : Money
{
    public Rupee(uint rupees, uint paise) : base(rupees, paise) { }
    public void Display()
    {
        Console.WriteLine("Rs. {0}.{1}", note, coin);
    }
}
```

Cont.

```
class Dollar : Money
{
    public Dollar(uint dollar, uint cent) : base(dollar, cent) { }
    public void Info()
    {
        Console.WriteLine("${0}.{1}", note, coin);
    }
}
class Test
{
    static void Main()
    {
        Rupee m1 = new Rupee(1000, 55);
        Dollar m2 = new Dollar(100, 75);
        Print[] p = new Print[2];
        p[0] = new Print(m1.Display);
        p[1] = new Print(m2.Info);
        write(p);
    }
    static void write(Print[] p)
    {
        p[0]();
        p[1]();
    }
}
```

Presented by

Ranjan Bhatnagar

Delegate Method Group Conversion

- Method Group Conversion is a C# language option that significantly simplifies the syntax that assigns a method to a delegate has included.
- As we had used in our first example.

```
using System;
class Program
{
    …
    public delegate void reftomethod(int i, int j);
    static void Main(string[] args)
    {
        Add(10, 20);
        reftomethod r1 = Add;
        r1(10, 20);
        reftomethod r2 = Sub;
        r2.Invoke(10, 20);
    }
}
```

Example using Method Group Conversion

- Here is an example using Method Group Conversion

```
using System;
delegate void Print();
class Money
{
    protected uint note;
    protected uint coin;
    public Money(uint n, uint c)
    {
        this.note = n;
        this.coin = c;
    }
}
class Rupee : Money
{
    public Rupee(uint rupees, uint paise) : base(rupees, paise) { }
    public void Display()
    {
        Console.WriteLine("Rs. {0}.{1}", note, coin);
    }
}
```

Method Group Conversion

```
class Dollar : Money
{
    public Dollar(uint dollar, uint cent) : base(dollar, cent) { }
    public void Info()
    {
        Console.WriteLine("${0}.{1}", note, coin);
    }
}
class Test
{
    static void Main()
    {
        Rupee m1 = new Rupee(1000, 55);
        Dollar m2 = new Dollar(100, 75);
        Print GP = m1.Display;
        GP();
        GP = m2.Info;
        GP();
    }
}
```

Method Group Conversion

Anonymous methods

- An anonymous method is an unnamed block of code that is used as parameter for the delegate.
- This is very handy if we want to create a method for one time use instead of creating a separate method.
- The scope of the parameters or variable of an anonymous method is restricted to the anonymous method block.
- A jump statement inside the anonymous method block cannot go beyond the anonymous method block.
- The anonymous method can access the local variables and parameters of its enclosing method. They are called outer or captured variables of the anonymous method. But they cannot access the ref or out parameters of an enclosing scope.
- Unsafe code cannot be accessed within the anonymous-method-block.

Anonymous Method Example

- This program first declares a delegate type called CountIt that has no parameters and returns void.
- Inside Main( ), a CountIt instance called count is created, and it is passed the block of code that follows the delegate keyword.
- This block of code is the anonymous method that will be executed when count is called.
- Notice that the block of code is followed by a semicolon, which terminates the declaration statement.

```
using System;
delegate void CountIt();
class AnonMethDemo
{
    static void Main()
    {
        CountIt count = delegate
        {
            for (int i = 1; i <= 10; i++)
                Console.WriteLine(i);
        };
        count();
    }
}
```

Anonymous Method with Argument

```
using System;
class Test
{
    delegate string Cat(string[] s);
    public static void Main()
    {
        Cat c = delegate(string[] s)
        {
            string c1 = "";
            foreach (string s1 in s)
                c1 = c1 + s1;
            return c1;
        };
        string[] ss = { "C#", "IN", "ACTION" };
        Console.WriteLine(c(ss));
    }
}
```

Presented by

Ranjan Bhatnagar

Return from an Anonymous Method

```
using System;
delegate int CountIt(int end);
class AnonMethDemo
{
    static void Main()
    {
        int result;
        CountIt count = delegate(int end)
        {
            int sum = 0;
            for (int i = 0; i <= end; i++)
            {
                Console.WriteLine(i);
                sum += i;
            }
            return sum;
        };
        result = count(3);
        Console.WriteLine("Summation of 3 is " + result);
         result = count(5);
        Console.WriteLine("Summation of 5 is " + result);
    }
}
```

Care with anonymous methods

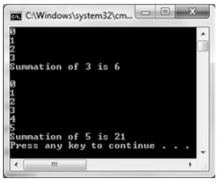
- Anonymous method makes the code slower.
- Advantage of anonymous method comes when you have to write a piece of code that will be used only in a single context. This will become more evident in event handling.
- The lambda expression improves on the concept of the anonymous method and is now the preferred approach to creating an anonymous function.

Outer Variable

- Anonymous methods are able to access the local variables of the method that defines them, such variables are termed outer variables of the anonymous method.
- Consider few points:
  - An anonymous method cannot access ref or out parameters of the defining method
  - An anonymous method cannot have a local variable with the same name as a local variable in the outer method
  - An anonymous method can access instance variables in the outer class scope
  - An anonymous method can declare local variables with the same name as outer class member variables.

Example Outer Variable

```
using System;
delegate int CountIt(int end);
class VarCapture
{
    static CountIt Counter()
    {
        int sum = 0;
        CountIt ctObj = delegate(int end)
        {
            for (int i = 0; i <= end; i++)
            {
                Console.WriteLine(i);
                sum += i;
            }
            return sum;
        };
        return ctObj;
    }
    static void Main()
    {
        CountIt count = Counter();
        int result;
        result = count(3);
        Console.WriteLine("Summation of 3 is " + result);
        Console.WriteLine();
        result = count(5);
        Console.WriteLine("Summation of 5 is " + result);
    }
}
```



Lambda Expression

- Lambda expressions are nothing more than a very concise way to author anonymous methods and ultimately simplify how we work with the .NET delegate type.
- A lambda expression is an anonymous function that you can use to create delegates or expression tree types.
- By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls.

Using Delegate

```
using System;
class Program
{
    static void Main(string[] args)
    {
        double a = Area(5.5);
        Console.WriteLine("Area:" + a);
    }
    static double Area(double r)
    {
        return 3.14 * r * r;
    }
}
```

Using Delegate

```
using System;
class Program
{
    delegate double CalcArea(double r);
    static CalcArea del = Area;
    static void Main(string[] args)
    {
        double a = del.Invoke(5.5);
        Console.WriteLine("Area:" + a);
    }
    static double Area(double r)
    {
        return 3.14 * r * r;
    }
}
```

Presented by

Ranjan Bhatnagar

Using Anonymous

```
using System;
class Program
{
    static void Main(string[] args)
    {
        double a = Area(5.5);
        Console.WriteLine("Area:" + a);
    }
    static double Area(double r)
    {
        return 3.14 * r * r;
    }
}
```

Using Anonymous

```
using System;
class Program
{
    delegate double CalcArea(double r);
    static void Main(string[] args)
    {
        CalcArea del = new CalcArea(
            delegate(double r)
            {
                return 3.14 * r * r;
            });
        double area = del(5.5);
        Console.WriteLine("Area:" + area);
    }
}
```

Made Simple with Lambda

Using Core Method

```
using System;
class Program
{
    static void Main(string[] args)
    {
        double a = Area(5.5);
        Console.WriteLine("Area:" + a);
    }
    static double Area(double r)
    {
        return 3.14 * r * r;
    }
}
```

Using Delegate

```
using System;
class Program
{
    delegate double CalcArea(double r);
    static CalcArea del = Area;
    static void Main(string[] args)
    {
        double a = del.Invoke(5.5);
        Console.WriteLine("Area:" + a);
    }
    static double Area(double r)
    {
        return 3.14 * r * r;
    }
}
```

Using Anonymous

```
using System;
class Program
{
    delegate double CalcArea(double r);
    static void Main(string[] args)
    {
        CalcArea del = new CalcArea(
            delegate(double r)
            {
                return 3.14 * r * r;
            });
        double area = del(5.5);
        Console.WriteLine("Area:" + area);
    }
}
```

```
using System;
class Program
{
    delegate double
              CalcArea(double r);
    static void Main
              (string[] args)
    {
        CalcArea del =
              r => 3.14 * r * r;
        double area = del(5.5);
        Console.WriteLine
              ("Area:" + area);
    }
}
```

- All lambda expressions use the lambda operator, which is =>.

Func<T, TResult> Delegate

- Encapsulates a method that has one parameter and returns a value of the type specified by the TResult parameter.
- You can use this delegate to represent a method that can be passed as a parameter without explicitly declaring a custom delegate.
- The encapsulated method must correspond to the method signature that is defined by this delegate.
- This means that the encapsulated method must have one parameter that is passed to it by value, and that it must return a value.

Using Func<T, TResult>

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Func<double, double> del = r => 3.14 * r * r;
        double area = del(5.5);
        Console.WriteLine(area);
    }
}
```

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Func<string, string> convert = s => s.ToUpper();
        string name = "India";
        Console.WriteLine(convert(name));
    }
}
```

Using Action<t>

- Action<t> is also generic delegate that Encapsulates a method that has a single parameter and does not return a value.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Action<string> MyMsg = n => Console.WriteLine(n);
        MyMsg("India is Great");
    }
}
```

Using Predicate<t>

- Represents the method that defines a set of criteria and determines whether the specified object meets those criteria.
- Predicate always return type of bool so it is used for checking.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Predicate<string> CheckPassLength = x => x.Length > 8;
        Console.WriteLine(CheckPassLength("passtheapp"));
    }
}
```

Presented by

Ranjan Bhatnagar

Expression Trees

- Expression Trees are special kind of binary trees.
- A binary tree is a tree in which all the nodes contain zero, one, or two children.
- The upper limit of steps necessary to find the required information in binary trees equals to $\log_2 N$, where N denotes the number of all nodes in a tree.
- Consider a simple declaration of Lambda Expression that takes two numbers as parameters and returns true whenever the first number is less than the second one and false otherwise.

```
Func<int, int, bool> f = (a, b) => a < b;
```

Constructing Expression Trees

- To create an expression tree that refers to the previous example, we have to use the following syntax:

```
Expression<Func<int, int, bool>> f = (a, b) => a < b;
```

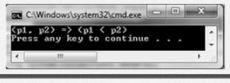
- The only difference is the use of the Expression<T> class.
- This class is somewhat handy as it contains four interesting properties holding useful information about the underlying expression tree:
  - Body
  - NodeType
  - Parameters
  - Type

Expression Tree Steps

- Alternatively, you could also build the same expression tree in another way – step by step:

```
Expression<Func<int, int, bool>> f = (a, b) => a < b;

        ParameterExpression par1 =
              Expression.Parameter(typeof(int), "p1");
        ParameterExpression par2 =
              Expression.Parameter(typeof(int), "p2");
        BinaryExpression expr = Expression.LessThan(par1, par2);
        var func = Expression.Lambda<Func<int, int, bool>>(expr,
                          new ParameterExpression[] { par1, par2
});
Console.WriteLine(func);
```



Using Expression Trees

- To use an expression tree, you must first compile it using Compile().

```
using System;
using System.Linq.Expressions;
class Program
{
    static void Main(string[] args)
    {
        Expression<Func<int, int, bool>> f = (a, b) => a < b;
        var func = f.Compile();
        var result = func(2, 5);
        Console.WriteLine(result);
    }
}
```

Understanding Events

- Delegates are fairly interesting constructs in that they enable objects in memory to engage in a two-way conversation.
- Moreover, when you use delegates in the raw as your application's callback mechanism, if you do not define a class's delegate member variables as private, the caller will have direct access to the delegate objects.
- In this case, the caller could reassign the variable to a new delegate object (effectively deleting the current list of functions to call) and, worse yet, the caller would be able to directly invoke the delegate's invocation list.

Consider Following Code Block

```
using System;
public class Car
{
    public delegate void CarEngineHandler(string msgForCaller);
    // Now a public member!
    public CarEngineHandler listOfHandlers;
    // Just fire out the Exploded notification.
    public void Accelerate(int delta)
    {
        if (listOfHandlers != null)
            listOfHandlers("Sorry, this car is dead...");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Agh! No Encapsulation! *****\n");
        // Make a Car.
        Car myCar = new Car();
        // We have direct access to the delegate!
        myCar.listOfHandlers =
               new Car.CarEngineHandler(CallWhenExploded);
        myCar.Accelerate(10);
```

Presented by

Ranjan Bhatnagar

Cont.

```
        // We can now assign to a whole new object...
        // confusing at best.
        myCar.listOfHandlers = new Car.CarEngineHandler(CallHereToo);
        myCar.Accelerate(10);
        // The caller can also directly invoke the delegate!
        myCar.listOfHandlers.Invoke("hee, hee, hee...");
        Console.ReadLine();
    }
    static void CallWhenExploded(string msg)
    { Console.WriteLine(msg); }
    static void CallHereToo(string msg)
    { Console.WriteLine(msg); }
}
```

- Exposing public delegate members breaks encapsulation, which not only can lead to code that is hard to maintain (and debug), but could also open your application to possible security risks!

The C# event Keyword

- C# provides the event keyword, so you don't have to build custom methods to add or remove methods to a delegate's invocation list,
- When the compiler processes the event keyword, you are automatically provided with registration and unregistration methods as well as any necessary member variables for your delegate types.
- These delegate member variables are always declared private, and therefore they are not directly exposed from the object firing the event.

Events

- An event is a way by which an object (publisher) can intimate/notify any other objects (subscribe) when some interesting thing happens to it.
- Events are declared using delegates.
- Events happen when a control is used by a user in a C# Windows Forms or Web application.
- The form or the web application first has to subscribe to events raised by controls such as buttons and list boxes.
- The Delegate used to write up such an event is called EventHandler delegate.

Simple Event Example

```
using System;
// Declare a delegate type for an
event.
delegate void MyEventHandler();
// Declare a class that contains
an event.
class MyEvent
{
    public event
       MyEventHandler SomeEvent;
       // This is called
       //to raise the event.
    public void OnSomeEvent()
    {
        if (SomeEvent != null)
            SomeEvent();
    }
}
```

```
class EventDemo
{
    // An event handler.
    static void Handler()
    {
        Console.WriteLine
              ("Event occurred");
    }
    static void Main()
    {
        MyEvent evt
              = new MyEvent();
        // Add Handler()
        //to the event list.
        evt.SomeEvent += Handler;
        // Raise the event.
        evt.OnSomeEvent();
    }
}
```

Producer, Consumer scenario

- Producer: Mangoes are produced by the mango farm.
- Consumer: The consumer needs to be intimated as soon as fresh lot of mangoes are plucked.
- Consumer has to register with Producer if it is interested in receiving intimation when the mangoes are produced.
- Producer class defines a delegate that is event based.
- Event delegate is defined with event keyword as
- public event EventHandler<TEventArgs> someEvent;
- The TEventArgs should inherits from EventArgs
- General Delegate Event Handler signature in consumer.
- public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
- The Producer calls the consumer method through the delegate when the event happens.

Producer Code

```
using System;
class ProduceMango
{
    public event EventHandler<MangoEventArgs> MangoInfo;
    string Mango;
    public ProduceMango(string type)
    {
        Mango = type;
    }
    public void FreshLot()
    {
        int i = new Random().Next(1000);
        string mangoinfo = i + " " + Mango + " mangoes produced ";
        Console.WriteLine(mangoinfo);
        if (MangoInfo != null)
        {
            MangoInfo(this, new MangoEventArgs(Mango, i));
        }
    }
}
```

Presented by

Ranjan Bhatnagar

Consumer Code

```
class MangoEventArgs : EventArgs
{
    public MangoEventArgs(string type, int number)
    {
        MangoInfo = type;
        Number = number;
    }
    public string MangoInfo { get; private set; }
    public int Number { get; private set; }
}
class ConsumeMango
{
    public void SqueeezeMango(object sender, MangoEventArgs e)
    {
        Console.WriteLine("Squeezing " + e.Number
               + " of " + e.MangoInfo + " mangoes");
    }
}
```

Code implementing producer and consumer

```
class Program
{
    static void Main(string[] args)
    {
        ConsumeMango Slice = new ConsumeMango();
        ProduceMango SalemFarms =
                new ProduceMango("Alphanso");
        // Slice registers event with SalemFarms
        SalemFarms.MangoInfo += Slice.SqueeezeMango;
        SalemFarms.FreshLot();
    }
}
```

Another Event Example

```
using System;
class MeltdownEventArgs : EventArgs
{
    private string message;
    public MeltdownEventArgs(string message)
    {
        this.message = message;
    }
    public string Message
    {
        get
        {
            return message;
        }
    }
}
```

Another Event Example

```
class Reactor
{
    private int temperature;
    public delegate void MeltdownHandler(
    object reactor, MeltdownEventArgs myMEA);
    public event MeltdownHandler OnMeltdown;
    public int Temperature
    {
        set
        {
            temperature = value;
            if (temperature > 1000)
            {
                MeltdownEventArgs myMEA = new
MeltdownEventArgs("Reactor meltdown in progress!");
                OnMeltdown(this, myMEA);
            }
        }
    }
}
```

Another Event Example

```
class ReactorMonitor
{
    public ReactorMonitor(Reactor myReactor)
    {
        myReactor.OnMeltdown +=
                new Reactor.MeltdownHandler(DisplayMessage);
    }
    public void DisplayMessage(object myReactor,
                        MeltdownEventArgs myMEA)
    {
        Console.WriteLine(myMEA.Message);
    }
}
```

Another Event Example

```
public class Program
{
    public static void Main()
    {
        Reactor myReactor = new Reactor();
        ReactorMonitor myReactorMonitor =
                    new ReactorMonitor(myReactor);
        Console.WriteLine("Setting reactor temperature
                                to 100 degrees Centigrade");
        myReactor.Temperature = 100;
        Console.WriteLine("Setting reactor temperature
                                to 500 degrees Centigrade");
        myReactor.Temperature = 500;
        Console.WriteLine("Setting reactor temperature
                                to 2000 degrees Centigrade");
        myReactor.Temperature = 2000;
    }
}
```

Thanks!