Delegates & Events

Delegation classes : used in C# to implement pointer to methods

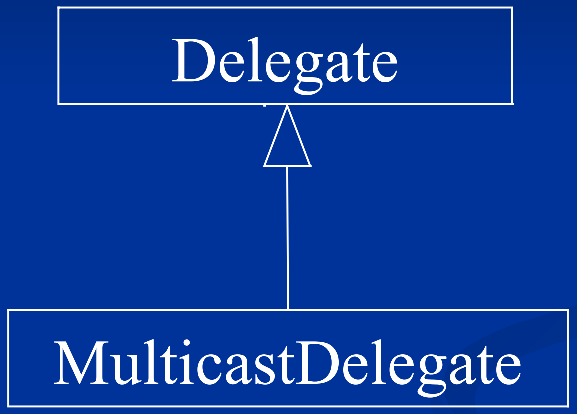
useful for event handling and event programming

a delegation class inherits from the Delegate class in a special way

a delegate is an instance of a delegation class

Delegates & Events

Delegation classes :



A delegation class is created by using the `delegate` keyword

this class is bound to a specific method signature.

Delegation class creation

```
delegate return_type className(parameters);
```

creates the className delegation class

example :

```
delegate int Deleg1(int x);
```

creates the Deleg1 class.

delegate creation

to create a delegate :

`ClassName Object = new ClassName(method);`

`method` must have a signature corresponding to the delegation class definition

example :

`Deleg1 del = new Deleg1(oups);`

with : `int oups(int param);`

A complete example

```
using System;
using utils; // remember class P

namespace delegation
{
   public delegate int Deleg1(int x); // this is a class

   class meths // this class contains only methods
   {
      static int method1(int a) {return a+1;}
      public int method2(int a) {return a+2;}
      public int method3(int a) {return a+3;}
   }

   … // the test class follows
}
```

A complete example

```
class test // also in delegation namespace
{
   static void Main(string[] args)
   {
      Deleg1 del = new Deleg1(meths.method1);
      // method0 is static
      P.rintln(meths.method1(5));
      P.rintln(del(5));

      // the two previous instructions do exactly the
   same thing
   }
}
```

A complete example

```
class test // also in delegation namespace
{
   static void Main(string[] args)
   {
      meths m = new meths();

      Deleg1 del = new Deleg1(m.method3);
      // method3 is an instance method, so a instance
      // of meths must be created before del

      P.rintln(m.method3(12));
      P.rintln(del(12));
   }
}
```

Accessing delegate information

from the delegate object : informations on the method name, the object to which the method is bound (NULL if the method is static), and the return type.

Multicast delegate :

a delegate stores information on several methods : linear list (`pointer _prev`)

Associating methods

let `del` be a delegate:

```
Deleg1 del = new Deleg1(m.method3);
```

to associate `del` to `m.method2` :

```
del = del+new Deleg1(m.method2);
```

execution is done in the same order :

calling `del(i)` calls `m.method3(i)` then `m.method2(i)`

Delegate Invocation List

calling `del(i)` returns the value computed by the last method called.

listing all the methods associated to a delegate :

`del.GetInvocationList()` returns an array of `Delegate` objects :

```
Delegate[] GetInvocationList();
```

Events

a method should be executed when some conditions are met :

- ~~wait for required conditions : blocking~~
- ~~wait in a thread : time-consuming, not coherent~~
- interruption : event programming

Events

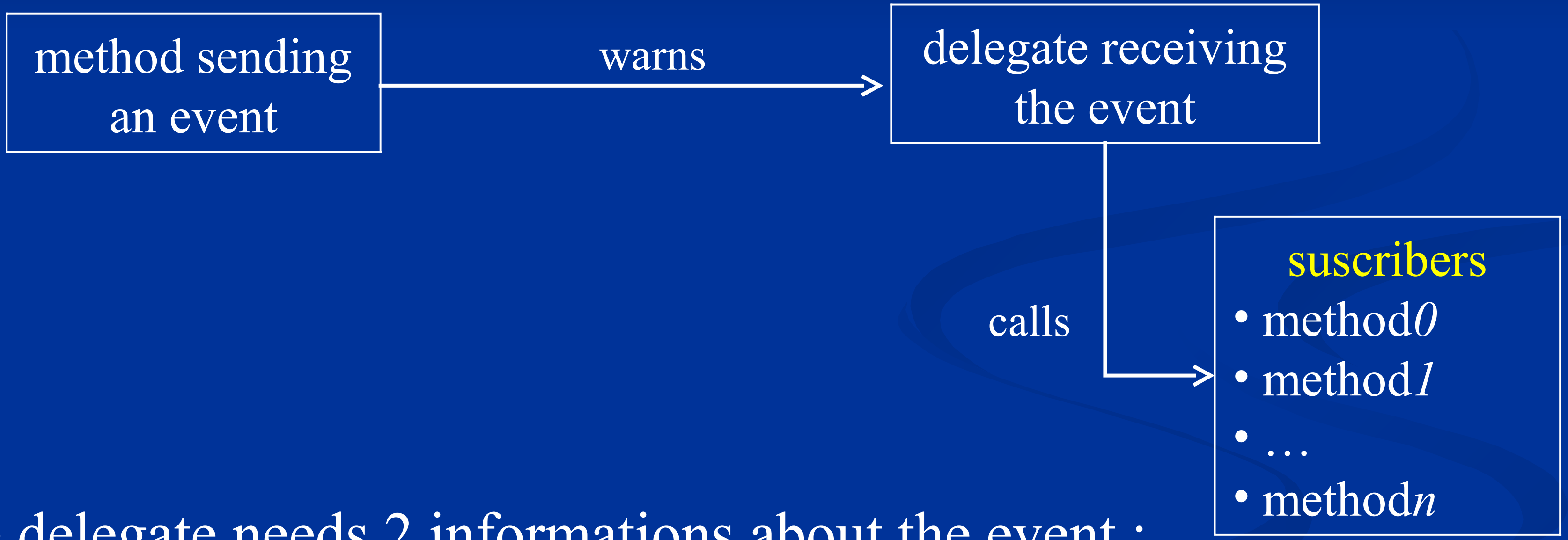
an object `O` raises an event :

some other objects R_i must react to this event

done through delegates

R_i have to suscribe to the the event raised by `O`

Event communication



the delegate needs 2 informations about the event :

- its source
- its nature

Event communication

source : sender object

nature : information of the event : EventArgs class

first step : create a delegation class

```
public delegate void myDelegateClass(object
   sender, EventArgs e);
```

Event storing

second step : create an event in a class

```
class transmittor
{
public event myDelegateClass MyDel;
```

myDel stores the delegate to be warned : it is an object

third step : create a method raising the event

Events and Delegates

- An event keyword is a scope modifier for the delegate !
- Invocation access to the multicast delegate is limited to the declaring class
- The behaviour is as though the delegate were private for invocation

Event raising

for inheritance purposes, first create a protected virtual method raising the event :

```
protected virtual void onMyDel(object
  sender, EventArgs e)
{
  if (myDel != null) // check for suscribers
  {
     myDel(sender,e);
  }
}
```

Event raising

now create a public method to raise the event

```
public void raiseEvent()
{
  onMyDel(this,EventArgs.Empty);
}
```

you may also create your own EventArgs with inheritance

that's all for the transmittor class

Event Handling

next step : create a class reacting to the event :

(or several classes)

```
class receiver
{
   public void action(object sender,
   EventArgs e)
   {
      Console.Writeline("event caught");
      // and some interesting things
   }
}
```

Event handling

last step : write the test class

```
class test
{
  static void Main(string[] args)
  {
     transmittor t = new transmittor();
     receiver r = new receiver();

     // subscription

     t.MyDel += new myDelegateClass(r.action);
     // to be continued...
```

Event handling

last step : write the test class

```
class test
{
   // …

   t.RaiseEvent(); //

   // r.action is called by the delegate

   Console.Read(); // pause

}
```

Subscription

```
subscribing :

t.MyDel += new myDelegateClass(r.action);

unsubscribing :

t.MyDel -= r.action;
```