Module 12

"Delegates, Events, and Lambda Expressions"

TEKNOLOGISK
INSTITUT

Agenda

- **Delegates**
- Events
- Anonymous Methods and Lambda Expressions
- Lab 12
- Discussion and Review

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

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

- Instances of this type are references to methods with this signature

```
class SimpleMath
{
   public static void Add( int i, int j ) { ... }
}
```

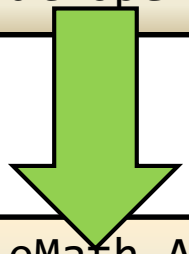
```
MathOperation m = new MathOperation( SimpleMath.Add );
```

- You can define delegates with any legal signature
- Delegates can reference both static and instance methods with the same syntax

Method Group Conversions

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

```
MathOperation m = new MathOperation( SimpleMath.Add );
```



```
MathOperation m = SimpleMath.Add;
```

- 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

```
MathOperation m = SimpleMath.Add;
...
m( 5, 7 );
```

```
m.Invoke( 5, 7 );
```

- And return values are used like conventional methods

```
public static string SayHello( string name )
{
   return string.Format( "Hello, {0}", name );
}
...
public delegate string HelloDelegate( string s );
```

```
HelloDelegate hello = SayHello;
Console.WriteLine( hello( "World" ) );
```

Multicasting Delegates

- C# delegates are in fact multicasting

```
MathOperation m = SimpleMath.Add;
m += SimpleMath.Multiply;
m( 5, 7 );
```

- Each delegate actually references a *list of methods* to be invoked – not just a single method!
- It has an internal invocation list

```
foreach( Delegate d in m.GetInvocationList() )
{
   Console.WriteLine( "Method Name: {0}", d.Method );
   Console.WriteLine( "Type Name: {0}", d.Target );
}
```

Removing Targets from Invocation List

- As demonstrated earlier, the **+=** operator adds a target to the invocation list.

```
MathOperation m = null;
m += SimpleMath.Add;
m += SimpleMath.Multiply;
...
m -= SimpleMath.Add;
m( 5, 7 );
```

- In a similar vein, the **-=** operator removes targets from the invocation list

- Note: It doesn't have to be the exact same reference which was added.
  - So you don't have to store original reference
  - Equality will ensure that the correct target gets removed

Delegates as Parameters

- Delegates can be supplied as parameters to methods

```
static void ShowInvocationList( Delegate del )
{
   foreach( Delegate d in del.GetInvocationList() )
   {
      Console.WriteLine( "Method Name: {0}", d.Method );
      Console.WriteLine( "Type Name: {0}", d.Target );
   }
}
```

- Similarly, delegates can be returned from methods

```
static MathOperation CreateDelegate()
{
   return SimpleMath.Add;
}
```

Generic Delegates

```
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 );
}
```

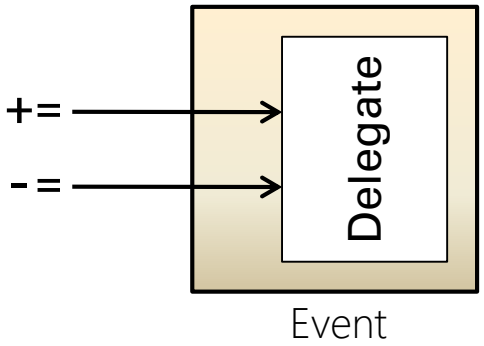
```
MyGenericDelegate<int> it = IntTarget;
it( 87 );

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

Agenda

- Delegates
- **Events**
- Anonymous Methods and Lambda Expressions
- Lab 12
- Discussion and Review

Introducing Events

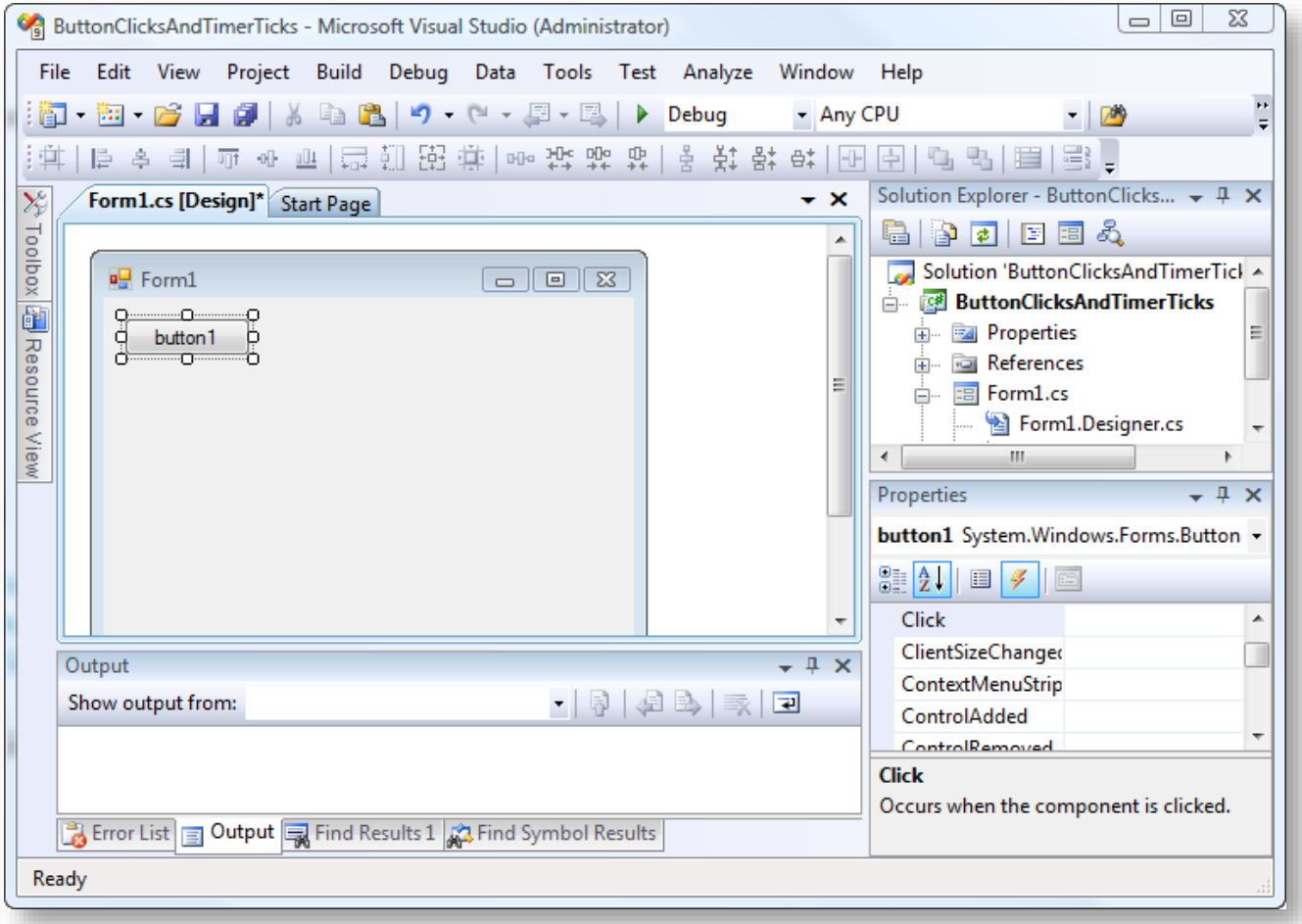
- Modern programming is event-driven
  - Occurrences of events trigger certain actions
  - Publisher-Subscriber scenario
  - E.g. button clicks in Windows applications
- Can delegates facilitate this kind of scenario?
  - Well… Yes, but…



- Events provide a convenient wrapper around delegates!

TEKNOLOGISK INSTITUT

Button Clicks and Timer Ticks



The **event** Keyword

- Events are constructed from some delegate signature with the **event** keyword

```
delegate void SubscriberDelegate( object publisher,
                                  SubscriptionInfo info );
```

```
public class Publisher
{  ...
   public event SubscriberDelegate NewInfo;
}
```

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

```
Publisher p = new Publisher();
Subscriber s1 = new Subscriber( "Ted" );
p.NewInfo += new SubscriberDelegate( s1.PublisherUpdated );
...
p.NewInfo -= new SubscriberDelegate( s1.PublisherUpdated );
```

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"*
- The delegate name is to be called *event name + "EventHandler"*

```
delegate void NewInfoEventHandler( object sender,
                                   NewInfoEventArgs args );
public class NewInfoEventArgs : EventArgs
{
   public NewInfoEventArgs() { timeStamp = DateTime.Now; }
   public DateTime timeStamp;
}
```

```
public class Publisher
{
   public event NewInfoEventHandler NewInfo;
}
```

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!

```
public delegate void EventHandler<T>( object sender, T e )
   where T: EventArgs
```

- Thus

```
delegate void NewInfoEventHandler( object sender,
                                   NewInfoEventArgs args );
```

```
public class Publisher
{
   public event EventHandler<NewInfoEventArgs> NewInfo;
}
```

Raising Events

- Events are raised by treating the event as the underlying delegate

```
if( NewInfo != null )
{
   NewInfo( this, new NewInfoEventArgs() );
}
```

- Remember to check whether event is null
  - This checks if there are any subscribers

- In C# 6.0 the Null-conditional Operator **?.** makes raising events cleaner
  - Note: **.Invoke()** must be explicitly applied for delegates

```
NewInfo?.Invoke( this, new NewInfoEventArgs() );
```

Agenda

- Delegates
- Events
- **Anonymous Methods and Lambda Expressions**
- Lab 12
- Discussion and Review

Defining Anonymous Methods

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

```
p.NewInfo += delegate
{
   Console.WriteLine( "NewInfo event was raised" );
};
```

- Note: The final "**;**" must be present!

- Parameters can be supplied to anonymous methods as usual

```
p.NewInfo += delegate( object sender, NewInfoEventArgs e )
{
   Console.WriteLine( "New info: {0}", e.TimeStamp );
};
```

Accessing Outer Variables

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

```
int eventOccurrences = 0;
...
p.NewInfo += delegate( object sender, NewInfoEventArgs e )
{
   Console.WriteLine( "New info: {0}", e.TimeStamp );
   eventOccurrences++;
};
```

- Note that these are shared!
  - Shared by all invocations
  - Can be modified in between invocations of the anonymous method by somebody else

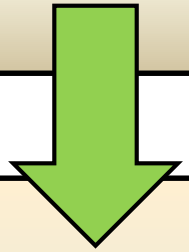
Defining Lambda Expressions

- Lambda expressions are a compact notation of the form

```
( Type1 arg1, ..., Typen argn ) =>
   Statements to Process Arguments
```

- They are just short-hands for anonymous methods

```
p.NewInfo += delegate( object sender, NewInfoEventArgs e )
{
   Console.WriteLine( "New info: {0}", e.TimeStamp );
};
```



```
p.NewInfo +=
   ( sender, e ) => Console.WriteLine( "New info: {0}",
                                       e.TimeStamp );
```

Arguments with Multiple Statements

- You can use multiple statements for argument processing by enclosing them in statement blocks, i.e. `{ … }`

```
p.NewInfo += ( sender, e ) =>
{
   Console.WriteLine( "New info: {0}", e.TimeStamp );
   eventOccurrences++;
};
```

- Outer variables can be accessed exactly as for anonymous methods

Expressions with Zero or One Parameters

- Lambda expressions could be parameterless

```
public delegate int SimpleNumberDelegate();
SimpleNumberDelegate d = () => 87;
Console.WriteLine( d.Invoke() );
```

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

```
// Built into .NET
public delegate bool Predicate<T>( T obj );

Predicate<int> p = ( i => i == 87 );
```

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

- This is where Lambda Expressions really rock! ☺

Expression-bodied Members

- In C# 6.0 all functionality-based members can have expression bodies
  - Methods
  - Properties
  - Indexers

```
public class Person
{
   public string FirstName { get; set; } = "Jens";
   public string LastName { get; set; } = "Jensen";
   public int Age { get; set; } = 0;

   public string FullName => $"{FirstName} {LastName}";

   public override string ToString() =>
      $"{Name} is {Age} year{(Age == 1 ? "" : "s")} old";
}
```

More Expression-bodied Members

- In C# 7.0 all remaining syntactic constructs can have expression bodies
  - **get**, **set** separately
  - Constructors
  - Destructors

```
public class Person
{
    ...
    public Person( string name ) => Names.Add(_id, name);

    ~Person() => Names.Remove(_id);

    public string Name
    {
        get => Names[_id];
        set => Names[_id] = value;
    }
}
```

Throw Expressions

- In C# 6 one could not easily just throw an exception in an expression-bodied member
- C# 7 allows **`throw`** expressions as subexpressions
  - Also outside of expression-bodied members..!

```
public class PersonRepository
{
    private readonly List<Person> _persons;
    ...
    public void Add( Person person ) =>
        _persons.Add(person ??
            throw new ArgumentNullException(nameof(person)));
}
```

- Note that a **`throw`** expression does not have an expression type as such…

Quiz: Lambda Expressions – Right or Wrong?

TEKNOLOGISK INSTITUT

```
p.NewInfo += e => Console.WriteLine( "New info: {0}", e.TimeStamp );
```

```
Predicate<int> p = ( i => i * 42 );
```

```
List<int> list = new List<int>{ 42, 87, 112, 59, 33, 128 };
List<int> unfiltered = list.FindAll( _ => true );
```

```
List<int> unfiltered = list.FindAll( () => true );
```

```
List<int> filtered = list.FindAll( i => { Console.WriteLine( i );
                                          return i < 87; } );
```

```
int j = 112;
List<int> filtered = list.FindAll( i =>
{
   return i != j;
} );
```

TEKNOLOGISK INSTITUT

Lab 12: Using Delegates, Events and Lambda Expressions

Discussion and Review

- Delegates
- Events
- Anonymous Methods and Lambda Expressions


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Hasselvangen 243
8355 Solbjerg
Denmark