Professional Software Engineering

Andrea Carrara and Patrick Berggold

Hritik Singh and Mohab Hassaan – Tutors

Chair of Computational Modeling and Simulation

andrea.carrara@tum.de; patrick.berggold@tum.de

Schedule Lecture 7

- **Functional Programming**
  - **Delegates**
  - **Anonymous Methods**
  - **Events**

andrea.carrara@tum.de; patrick.berggold@tum.de

FUNCTIONAL PROGRAMMING

Packaging chunks of code into functions...

andrea.carrara@tum.de; patrick.berggold@tum.de

Functional Programming

- It is a programming paradigm where a program is constructed by applying functions
- In C#, a function is a code snippet that receives input parameters and returns either one or no value

- We already know this, but we'll now dive deeper into it…
- C# and many other languages offer many useful functionalities

andrea.carrara@tum.de; patrick.berggold@tum.de

Functional Programming

- It is a programming paradigm where a program is constructed by applying functions
- In C#, a function is a code snippet that receives input parameters and returns either one or no value

- Syntax:

```
// what we have seen so far...
void SomeFunction(int param1){
   // something happens, no returns
}
int someVariable = 5;
SomeFunction(someVariable);
```

- We are passing a values (`param1`) into a function (`SomeFunction`), but can we actually pass functions as input parameters?

➡ Yes! Use **delegates**!

andrea.carrara@tum.de; patrick.berggold@tum.de

DELEGATES

A variable reference to methods

andrea.carrara@tum.de; patrick.berggold@tum.de

Delegates

- Delegates are specialized classes to store references to methods (hence, they are reference types)

- As long as input-types and output-type are valid, they are placeholders for any functions matching their signature

  ↳ Delegates allow us to parameterize our code!

- While creating the delegate instance, we need to pass the method as a parameter to the delegate constructor

- Multicast – Delegates contain more than one reference, thus executing a series of methods when being called

andrea.carrara@tum.de; patrick.berggold@tum.de

Creating Delegates

- `delegate` as keyword, name is normally „…Handler“
- Datatypes must match!
  - **Use any function that has the same signature**
- Invoke using `()`
- If delegates hold more than one reference, they are called multicast delegates

```
// Declare the delegate
delegate double ConversionHandler(int input);


// Declare the method
double ConvertInt2Double(int int_input) {...}


// Instantiation
ConversionHandler del_converter =
        new ConversionHandler(ConvertInt2Double);

// both return 3.0 as double
double d_num1 = del_converter(3);
double d_num2 = ConvertInt2Double(3);
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Delegate Example, Part 1

```
// declarations...
enum WorkerType { Clerk, Student, Teacher }
enum WorkType { Study, WriteExam, Eat, Sleep, GoToWork, TakeBreak, BuildStuff }

// delegate declaration
delegate void WorkHandler(int hours, WorkerType workerType, WorkType workType);

// function declaration
static void WorkPerformed(int h, WorkerType workerType, WorkType workType)
{
    // perform work
}
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Delegate Example, Part 2

```
// instantiations and use ...
WorkHandler del_student_1 = new WorkHandler(WorkPerformed);
del_student_1(10, WorkerType.Student, WorkType.Sleep); // invoke method

WorkHandler del_clerk_2 = WorkPerformed; // same thing (thanks compiler!)
del_clerk_2(1, WorkerType.Clerk, WorkType.GoToWork); // invoke method

// multicast delegate: invokes both delegates (the same input arguments)!
var del_workers = del_student_1 + del_clerk_2;
del_workers(1, WorkerType.Clerk, WorkType.GoToWork); // invoke both methods
```

Assign method to delegates

Invoke those methods

You can even add delegates and invoke all their methods at once!

andrea.carrara@tum.de; patrick.berggold@tum.de

Advantages of using delegates

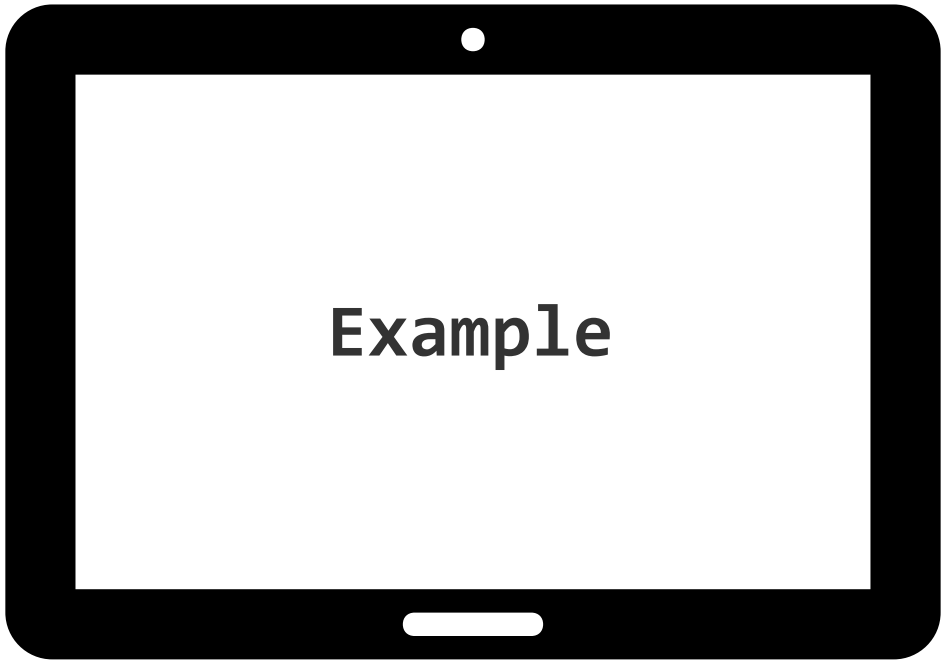
```
// function declaration
static void WorkPerformed(int h, WorkerType workerType, WorkType workType)
{
    // perform work
}

// invoking methods normally yields same results (here)...
WorkPerformed(10, WorkerType.Student, WorkType.Sleep);
WorkPerformed(1, WorkerType.Clerk, WorkType.GoToWork);
```

For simple use cases it may not matter much…

- Delegates are type-safe function pointers ⇒ useful when we don't care about concrete functions, but only about the signature
- We can pass delegates to methods as input parameters!
- We can invoke several methods at once

andrea.carrara@tum.de; patrick.berggold@tum.de

A More Advanced Delegate Example...



andrea.carrara@tum.de; patrick.berggold@tum.de

Multicast Delegate

```
// declare delegate
delegate void PrintHandler();
// some functions...
void Foo() { Console.WriteLine("Foo()"); }
void Goo() { Console.WriteLine("Goo()"); }
void Soo() { Console.WriteLine("Soo()"); }
PrintHandler del_print = Foo;
del_print += Goo; // passed as delegate
del_print += Soo;
del_print += Foo;
del_print -= Soo; // removes last function named Soo
del_print();
```

andrea.carrara@tum.de; patrick.berggold@tum.de

Multicast Delegate

```
// declare delegate
delegate void PrintHandler();
// some functions...
void Foo() { Console.WriteLine("Foo()"); }
void Goo() { Console.WriteLine("Goo()"); }
void Soo() { Console.WriteLine("Soo()"); }
PrintHandler del_print = Foo;
del_print += Goo; // passed as delegate
del_print += Soo;
del_print += Foo;
del_print -= Soo; // removes last function named Soo
del_print();
```

```
// += equal to this:
del_print = (PrintHandler) Delegate
   .Combine(del_print, new PrintHandler(Soo));
```

Static method in the Delegate class

```
// () equal to this:
del_print.Invoke();
```

Method in the Delegate class

Multicast Delegate, Part 2

```
// iterate through invocation list
foreach (PrintHandler item in del_print.GetInvocationList()){
    Console.WriteLine(item.Target + ": " + item.Method);
}
```

```
: Void Foo()
: Void Goo()
: Void Foo()
```

- `GetInvocationList()` returns invoked methods array in invocation order
- `.Target` returns the object on which the delegate invokes the method
- `.Method` returns the method represented by the delegate

andrea.carrara@tum.de; patrick.berggold@tum.de

Multicast Delegate, Part 3

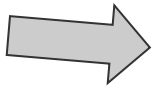
```
// yet another delegate example (sorry)...
delegate int IntegerHandler ();

int ReturnOne() {return 1;}
int ReturnTwo() {return 2;}

IntegerHandler int_del = ReturnOne;
int_del += ReturnTwo;

// Invoke this multicast delegate
int last_value = int_del(); // Returns 2!!!
```

Assigns 2 since ReturnTwo is the last function that is invoked

andrea.carrara@tum.de; patrick.berggold@tum.de

Generic Delegates – Func & Action

» Recap generics: keep input and return types generic with <T> notation



```
// Generic method
public static T SomeFunction<T>(T arg1, string arg2) {
    // ...
    // returns variable of type T!
}

int result_as_int = SomeFunction(0, "Hello there");
string result_as_str = SomeFunction("I like coding", "Hello there");
```

» We can make delegates generic as well!

- `Action` **delegate encapsulates a void function with no input parameters**
- `Action<T1, T2, ...>` **delegate encapsulates a void function with input parameters of types** `T1`, `T2`, ...
- `Func<T>` **delegate encapsulates a non-void function that returns type** `T`
- `Func<T1, T2, ..., T>` **delegate encapsulates a non-void function that returns type** `T` **with input parameters of types** `T1`, `T2`, ...

`Action` and `Func` are generic built-in delegates

andrea.carrara@tum.de; patrick.berggold@tum.de

Generic Delegates – Func & Action

» We can make delegates generic as well!

- `Action`: for void functions (no input)
- `Action<T1, T2, ...>`: for void functions (with input parameters of types `T1`, `T2`, …)
- `Func<T>`: for non-void functions (no input), return type `T`
- `Func<T1, T2, ..., T>`: for non-void functions (with input parameters of types `T1`, `T2`, …), return type `T`

```
void VoidMethod<T>(T param) { Console.WriteLine(param.ToString()); }
// Using an Action
Action<int> PrintAction_a = VoidMethod;
PrintAction_a(10); // Prints 10

// Using a generic delegate
delegate void MyActionDelegate<T>(T item);
MyActionDelegate<int> PrintAction_d = VoidMethod;
PrintAction_d(10); // Prints 10
```

Using generic delegates is equal to using Func and Action

andrea.carrara@tum.de; patrick.berggold@tum.de

Generic Delegates – Func & Action

» We can make delegates generic as well!

- `Action`: for void functions (no input)
- `Action<T1, T2, ...>`: for void functions (with input parameters of types `T1`, `T2`, …)
- `Func<T>`: for non-void functions (no input), return type `T`
- `Func<T1, T2, ..., T>`: for non-void functions (with input parameters of types `T1`, `T2`, …), return type `T`

```
string ReturnString<T1, T2>(T1 param1, T2 param2){return "Something";}
// Using a Func
Func<int, bool, string> ReturnFunc_f = ReturnString;
ReturnFunc_f(3, true); // returns "Something"

// same as using a generic delegate!
delegate string MyFuncDelegate<T1, T2>(T1 in1, T2 in2);
MyFuncDelegate<int, string> ReturnFunc_d = ReturnString;
ReturnFunc_d(5, "false"); // returns "Something"
```

Using generic delegates is equal to using Func and Action

andrea.carrara@tum.de; patrick.berggold@tum.de

ANONYMOUS METHODS

A feature of delegates to speed up coding

andrea.carrara@tum.de; patrick.berggold@tum.de

Anonymous Methods

- Delegates can also be used to declare a function without name („anonymous“)
- The function then can‘t be called again – so anonymous methods are used if a block of code is just meant to be used once
- There are two syntaxes for anonymous methods:
  - **One using the delegate-keyword**
  - **Another one using the Lambda-Operator (‘=>‘)**

andrea.carrara@tum.de; patrick.berggold@tum.de

Anonymous Methods via delegate keyword

```
// delegate declaration
delegate void PrintHandler(int value);

PrintHandler print = delegate(int val) {
     Console.WriteLine("First Anonymous call, value: {0}", val);
};
print += delegate(int val) {
     Console.WriteLine("Second Anonymous call, value: {0}", val);
};
print(0);
```

This is where the method name used to be

```
First Anonymous call, value: 0
Second Anonymous call, value: 0
```

- No access modifier, no name, and no return statement
- Only uses a method body and the `delegate` keyword... Get rid of the keyword via Lambda operator!

andrea.carrara@tum.de; patrick.berggold@tum.de

Anonymous Methods via lambda expressions

- Syntax: Lambda operator (=>), separates input (left) from the output (right)
- Specification of datatypes not needed! Will be figured out by the compiler…
- Two types of Lambda expressions:
  - **Expression Lambda**
  - **Statement Lambda**
- Use {} for multi-line definitions
- () => for no parameters!

```
// Expression Lambda

input => expression;

// Statement Lambda

input => {statements};
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Anonymous Methods via lambda expressions

```
delegate void PrintHandler(int value);
// using delegate keyword
PrintHandler print_d = delegate(int val) {
     Console.WriteLine("First Anonymous call, value: {0}", val);
};
// using lambda expression
PrintHandler print_l = (val) => Console.WriteLine("First Anonymous call, value: {0}", val);
```

» From delegate to lambda:

```
delegate(int val) { =>
     Console.WriteLine("First Anonymous call, value: {0}", val);
};
```

```
(val) => Console.WriteLine("First Anonymous call, value: {0}", val);
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Using Lambdas to filter/query data

» Lambdas are a powerful tool to filter and query data structures

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n<6);
// print numbers as long as condition is violated
// print code...
```

`TakeWhile`:
Returns elements from a sequence as long as a specified condition is true, and then skips the remaining elements.

```
5, 4, 1, 3
```

» Quite useful for iterating through database… See LINQ lecture!

andrea.carrara@tum.de; patrick.berggold@tum.de

An advanced use-case of delegates

EVENTS

andrea.carrara@tum.de; patrick.berggold@tum.de

Overview - Events

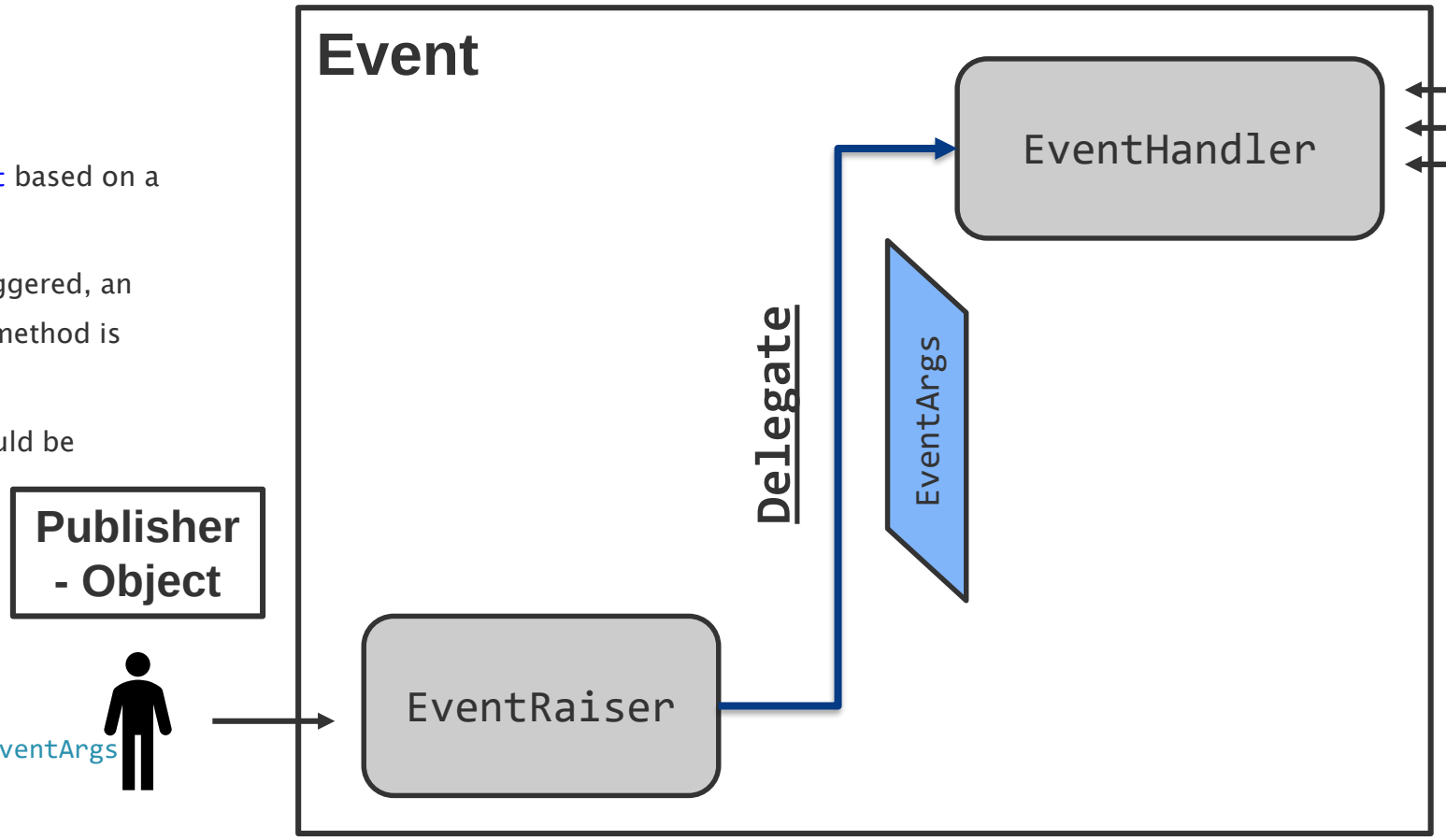
- Events are notification mechanisms whose communication (their signature) depends on delegates
  - **The sender object is called “Publisher”, the receiving object “subscriber” or “listener”**
  - **Event-based communication in C# is implemented very similarly to other popular languages**
- Main difference between events & delegates:
  - **Delegates are independent, events always depend on delegates**
  - **Event methods cannot be invoked directly outside of their class (but += and -= operators work)**
- Example: Click a Button!

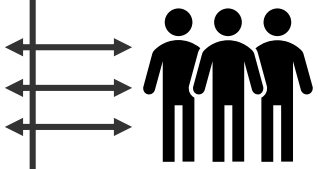
→ “Improves” delegate’s security by ensuring methods are called only when an event is triggered

andrea.carrara@tum.de; patrick.berggold@tum.de

Publisher class:

- Declare an `event` based on a `delegate`
- If an event is triggered, an **On***Something*() method is invoked
- Event trigger could be anything: add to list, send a signal, etc.
- Check if there are subscribers and if so, pass `EventArgs`





Subscriber - Objects

Subscriber class:

- Subscribe to an event by accessing the event of the publisher instance

andrea.carrara@tum.de; patrick.berggold@tum.de

Event Syntax – Main Example

```
static void Main(string[] args) {
    PublisherList publisherList = new PublisherList();
    Listener listener1 = new Listener(publisherList);
    Listener listener2 = new Listener(publisherList);
    Listener listener3 = new Listener(publisherList);
    publisherList.Add("Some element");
}
```

```
List was changed!
List was changed!
List was changed!
```


andrea.carrara@tum.de; patrick.berggold@tum.de

Event Syntax – Publisher Example

```
public class PublisherList : ArrayList {
    public delegate void ListEventHandler(object sender, EventArgs e);
    public event ListEventHandler ChangedList;

    protected virtual void OnChangedList(EventArgs e) {
        if(ChangedList != null) {
            ChangedList(this, e);
        }
    }
    // override the ArrayList method Add()
    public override int Add(object value) {
        OnChangedList(EventArgs.Empty);
        return base.Add(value);
    }
}
```

(these parameters are optional)

Delegate is the 'blueprint' for the event

dotnet convention: event publisher methods should be `protected`, `virtual`, `void` and named On+ *<name of the event>*

Check if event has any subscribers – if there were no subscribers, there would be no method to invoke

Trigger and event when a value is added to the list by invoking `OnChangedList`

andrea.carrara@tum.de; patrick.berggold@tum.de

Event Syntax – Listener Example

```
public class Listener {
    public Listener(PublisherList publisherList) {
        this.pubList = publisherList;
        this.pubList.ChangedList += listWasChanged;
    }
    public PublisherList pubList;

    private void listWasChanged(object sender, EventArgs args) {
        Console.WriteLine("List was changed!");
    }
}
```

Subscribe to the event `ChangedList` by adding a method reference that is executed when `ChangedList` is invoked in any publisher instance

Print whenever `listWasChanged()` is called

andrea.carrara@tum.de; patrick.berggold@tum.de

Event Syntax – Main Example

```
static void Main(string[] args) {
    PublisherList publisherList = new PublisherList();
    Listener listener1 = new Listener(publisherList);
    Listener listener2 = new Listener(publisherList);
    Listener listener3 = new Listener(publisherList);
    publisherList.Add("Some element");
}
```

```
List was changed!
List was changed!
List was changed!
```

`PublisherList::Add()` is called

⇩

`PublisherList::OnChangedList()` is called

⇩

Event `PublisherList::ChangedList` is called which references three `Listener::listWasChanged()` methods that are invoked sequentially

andrea.carrara@tum.de; patrick.berggold@tum.de

Events – Webserver Example

andrea.carrara@tum.de; patrick.berggold@tum.de

THANK YOU !

andrea.carrara@tum.de; patrick.berggold@tum.de