C# – 3

by

Dr. Günter Kolousek

Generics

- etwa wie in Java, aber
  - auch für `int`,...
  - besser integriert, z.B. Instanzierung von Objekt eines formalen Typparameters möglich
- Generics mit
  - Klassen und Interfaces
  - Strukturen
    - wie Klassen aber ohne Vererbung
  - Methoden
  - Delegates

Generics – 2

```
using System;
using System.Collections.Generic;

class WorkQueue<T> {
    private Queue<T> queue=new Queue<T>();
    public void put(T e) { queue.Enqueue(e); }
    public T get() { return queue.Dequeue(); }
}

public class Program {
    static public void Main() {
        WorkQueue<int> wq=new WorkQueue<int>();
        wq.put(2);
        wq.put(1);
        Console.WriteLine(wq.get());  // -> 2
        Console.WriteLine(wq.get());  // -> 1
    } }
```

Generics – 3

```
using System;
using System.Collections.Generic;
class WorkStack<T> {
    //private T bottom=default(T);//”Defaultwert” setzen
    private T bottom=default;  // ab C#7.1
    private Stack<T> stack=new Stack<T>();
    public WorkStack() { stack.Push(bottom); }
    public void put(T e) { stack.Push(e); }
    public T get() { return stack.Pop(); }
}
public class Program {
    static public void Main() {
        WorkStack<int> ws=new WorkStack<int>();
        ws.put(1);
        ws.put(2);
        Console.WriteLine(ws.get());  // -> 2
        Console.WriteLine(ws.get());  // -> 1
        Console.WriteLine(ws.get());  // -> 0
    } }
```

Generics – 4

```
using System;
using System.Collections.Generic;

class WorkPacket {}
// von WorkPacket (Klasse/Interface) abgeleitet
// alternativ: struct,class,(zusätzlich) new()
class WorkQueue<T> where T:WorkPacket {
    private Queue<T> queue=new Queue<T>();
    public void put(T e) { queue.Enqueue(e); }
    public T get() { return queue.Dequeue(); }
}

public class HelloWorld {
  static public void Main() {
    WorkQueue<WorkPacket> wq=new WorkQueue<WorkPacket>()
    wq.put(new WorkPacket());
    wq.put(new WorkPacket());
    Console.WriteLine(wq.get());  // -> WorkPacket
    Console.WriteLine(wq.get());  // -> WorkPacket
  } }
```

Generics – 5

```
using System;
using System.Collections.Generic;

// new()->anlegbar (keine Parameter, nicht abstract)
class WorkStack<T> where T:new() {
    private T bottom=new T();
    private Stack<T> stack=new Stack<T>();
    public WorkStack() { stack.Push(bottom); }
    public void put(T e) { stack.Push(e); }
    public T get() { return stack.Pop(); }
}
public class HelloWorld {
    static public void Main() {
        WorkStack<int> ws=new WorkStack<int>();
        ws.put(1);
        ws.put(2);
        Console.WriteLine(ws.get());
        Console.WriteLine(ws.get());
        Console.WriteLine(ws.get());
    } }
```

Generics – 6

```
using System;

public class Program {
    // generic method
    void swap<T> (ref T x, ref T y) {
        T tmp;
        tmp = x;
        x = y;
        y = tmp;
    }

    public static void Main() {
        Program prg=new Program();
        int i=1;
        int j=2;
        prg.swap<int>(ref i, ref j);
        Console.WriteLine($"{i}, {j}");
    }
}
```

Generics – Invariance

```
using System;
using System.Collections.Generic;

interface IQueue<T> {}

// invariant interface
class Queue<T> : IQueue<T> {}

class Program {
    static void doit(IQueue<String> q) {}

    static void Main() {
        IQueue<Object> iobj=new Queue<Object>();
        IQueue<String> istr=new Queue<String>();
        // invariant!
        // will not compile -> Cannot implicitly convert type.
        iobj = istr;
        // will not compile ->  Argument 1: cannot convert fro
        doit(iobj);
    }
}
```

Generics – Covariance

```
using System;
using System.Collections.Generic;

// covariant interface
interface IQueue<out T> {}

class Queue<T> : IQueue<T> {}

class Program {
    static void Main() {
        IQueue<Object> iobj=new Queue<Object>();
        IQueue<String> istr=new Queue<String>();
        // covariant!
        // now it /will/ compile!
        iobj = istr;
    }
}
```

Generics – Contravariance

```
using System;
using System.Collections.Generic;

// contra-variant interface
interface IQueue<in T> {}

class Queue<T> : IQueue<T> {}

class Program {
    static void doit(IQueue<String> q) {}
    static void Main() {
        IQueue<Object> iobj=new Queue<Object>();
        IQueue<String> istr=new Queue<String>();
        // contra-variant!
        // now it /will/ compile!
        doit(iobj);
    }
}
```

Delegates

- typsichere ”Funktionszeiger”
- legt die Schnittstelle einer Methode fest
- Delegate ist ein Typ: Anlegen mittels `new`
- Delegates sind von der Klasse `Delegate` abgeleitet
  - …und sind `sealed` (d.h. kein Ableiten möglich)
- Verwendung: Variable, Parameter
- Achtung: Begriff wird sowohl für den Typ als auch für die Instanz verwendet!

Delegates - statische Methode

```
using System;

delegate double Mean(double a, double b);

public class Program {
    public static double arith_mean(double a, double b)
        => (a + b) / 2;

    public static void Main() {
        Mean mean = arith_mean;

        Console.WriteLine(mean(3, 5));
    }
}
```

Delegates - Instanzmethode

```
public delegate double Mean(double a, double b);

public class MeanCalculator {
    public double arith_mean(double a, double b) {
        return (a + b) / 2;
    }

    public double geom_mean(double a, double b) {
        return Math.Sqrt(a * b);
    }
}

public static void Main() {
    // ... vorherige Folie, dann:
    MeanCalculator calc=new MeanCalculator();
    mean = calc.arith_mean;
    Console.WriteLine(mean(4, 6));
    mean = calc.geom_mean;
    Console.WriteLine(mean(6, 6));
}
```

Delegates - Multicasting

- Mehr als eine Methode ($\rightarrow$ `MulticastDelegate`)
  - `+` und `+=`: hinzufügen
  - `-` und `-=`: entfernen

```
using System;
public class Program {
    public delegate void Log(string msg);

    public static void error_log(string msg) {
        Console.Error.WriteLine(msg); }
    public static void out_log(string msg) {
        Console.WriteLine(msg); }
    public static void debug_log(string msg) {
        Console.Error.WriteLine($"Debug: {msg}"); }
    public static void Main() {
        Log log = error_log;
        log = log + out_log;
        log += debug_log;
        log("Hello, world");  } }
```

Delegates & Lamda Ausdrücke

```
using System;

public class Program {
  public static void Main() {
      // Action: predefined without a return value
      Action<string> log=msg => Console.WriteLine(msg);
      log(”Hello, World”);

      // Func: predefined with a return value
      // x, y, return
      Func<double, double, double> mean=
        (x, y) => (x + y) / 2;
      Console.WriteLine(mean(3, 5));
  }
}
```

Delegates & Lambda Anweisung – 2

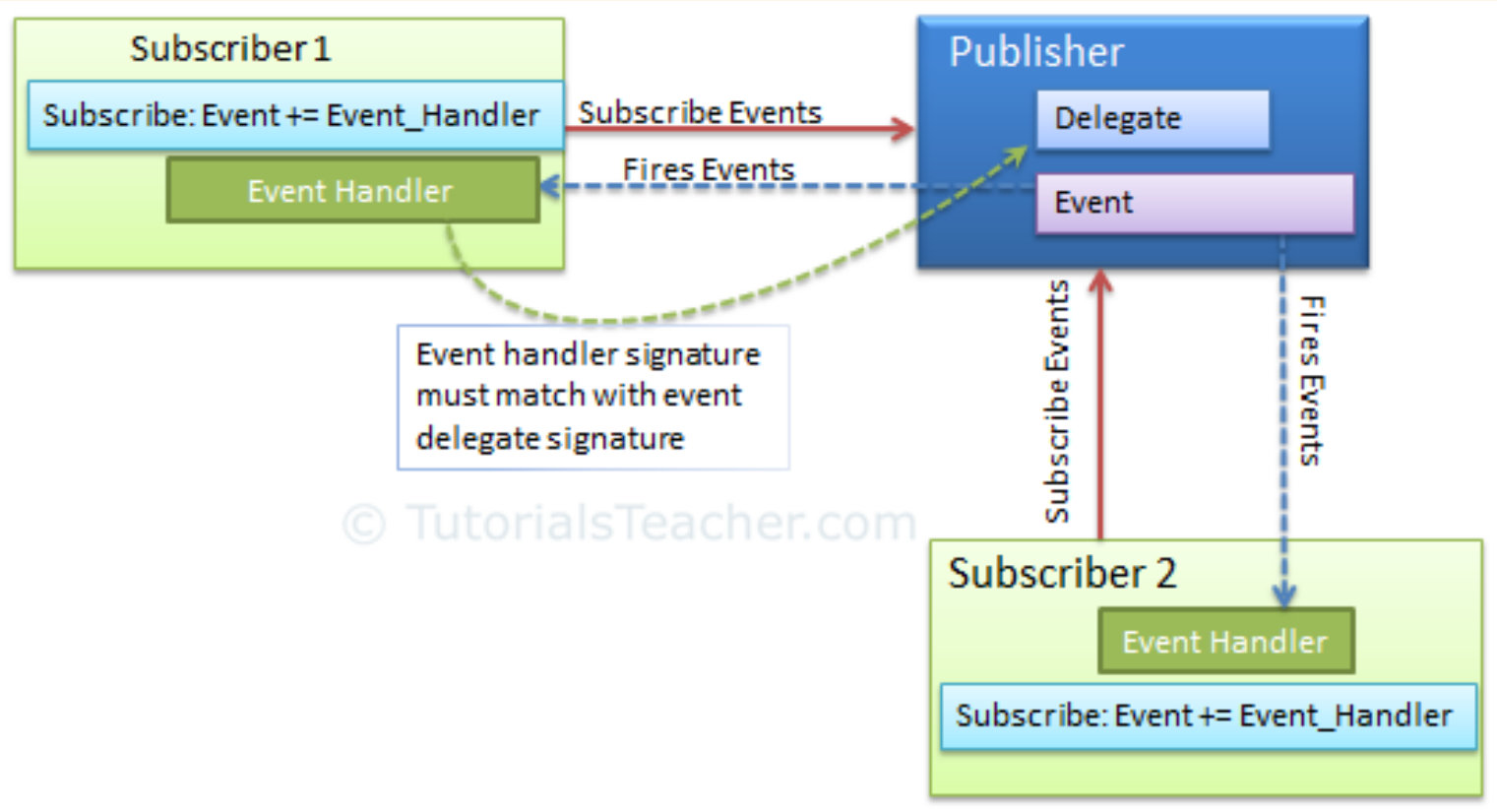
```
using System;

public class Program {
    public static void Main() {
        // multiple code lines
        Func<string, string> output=msg => {
            string ret=”Hello, ”;
            ret += msg;
            return ret;
        };
        Console.WriteLine(output(”World”));
        // closures
        double offset=1;
        Func<double, double> adder=param => param + offset;
        Console.WriteLine(adder(1));  // -> 2
        offset = 2;
        Console.WriteLine(adder(1));  // -> 3
    }
}
```

Events

- Events und Delegates sind dem Observer Pattern nachempfunden $\rightarrow$ Sinn ist das Benachrichtigen bei Änderungen!
- Events sind eine Art von Delegate
  - Rückgabewert immer `void`!
- Events werden mittels eines Delegate definiert
- Events können mehrere Eventhandler zugeordnet werden, die dem definierten Delegate entsprechen müssen
- Wenn ein Event gefeuert wird, dann werden die zugeordneten Eventhandler aufgerufen
- Events bilden eine zusätzliche Abstraktion über Delegates
- Events bieten einen zusätzlichen Schutz gegenüber Delegates
  - nur += und −=

Events – 2


Subscriber 1
Subscribe: Event += Event_Handler
Event Handler
Subscribe Events
Fires Events
Publisher
Delegate
Event
Event handler signature must match with event delegate signature
© TutorialsTeacher.com
Subscribe Events
Fires Events
Subscriber 2
Event Handler
Subscribe: Event += Event_Handler

Events – 3

```
using System;

public class Program {
    public delegate void AlarmHandler(string msg);
    public static event AlarmHandler alarm;

    public static void alarm_handler1(string msg) {
        Console.WriteLine($"Handler1: {msg}");
    }
    public static void alarm_handler2(string msg) {
        Console.WriteLine($"Handler2: {msg}");
    }
    public static void Main() {
        // alarm += new AlarmHandler(alarm_handler1);
        alarm += alarm_handler1;
        alarm += alarm_handler2;

        alarm("Feuer!");
    } }
```

Events – 4

- anstatt eigenem `delegate` gibt es vordefiniert `System.EventHandler<TEventArgs>`
- Eventhandler hat dann folgende Signatur: `void handler(object sender, EventArgs args)`
- Definition der Argumentklasse

```
using System;
using static System.Console;

public class AlarmEvtArgs : EventArgs {
    public AlarmEvtArgs(string _msg) {
        msg = _msg;
    }
    public string msg { get; }
}
```

Events – 5

- Definition des Events-Teils

```
public class AlarmMachine {
  public AlarmMachine(string _id) { id = _id; }

  public event EventHandler<AlarmEvtArgs> alarm_event;

  public void alarm_handler1(object s, AlarmEvtArgs e) {
    WriteLine($"1: {((AlarmMachine)s).id}: {e.msg}");
  }

  public void alarm_handler2(object s, AlarmEvtArgs e) {
    WriteLine($"2: {((AlarmMachine)s).id}: {e.msg}");
  }

  public void alarm() {
    alarm_event(this, new AlarmEvtArgs("Feuer!"));
  }
  private string id;
}
```

Events – 6

- Definition der Verwendung

```
public class Program {
    public static void Main() {
        AlarmMachine am=new AlarmMachine(”am1”);
        am.alarm_event += am.alarm_handler1;
        am.alarm_event += am.alarm_handler2;

        am.alarm();
    }
}
```