

# Praktische Informatik

Vorlesung 03

Delegates und Ereignisse

# Zuletzt haben wir gelernt...

- Welche Strategien für die Übersetzung von Quellcode existieren und wie dies auf der .Net Plattform geschieht.
- Was Assemblies sind.
- Wie man Bibliotheken erstellt und diese in Projekten zur Wiederverwendung von Programmcode einsetzt.
- Was ein Paketmanager leistet.
- Wie man NuGet einsetzt, um eine externe Bibliothek zu einem Projekt hinzuzufügen.
- Was Buildwerkzeuge sind und warum diese gebraucht werden.

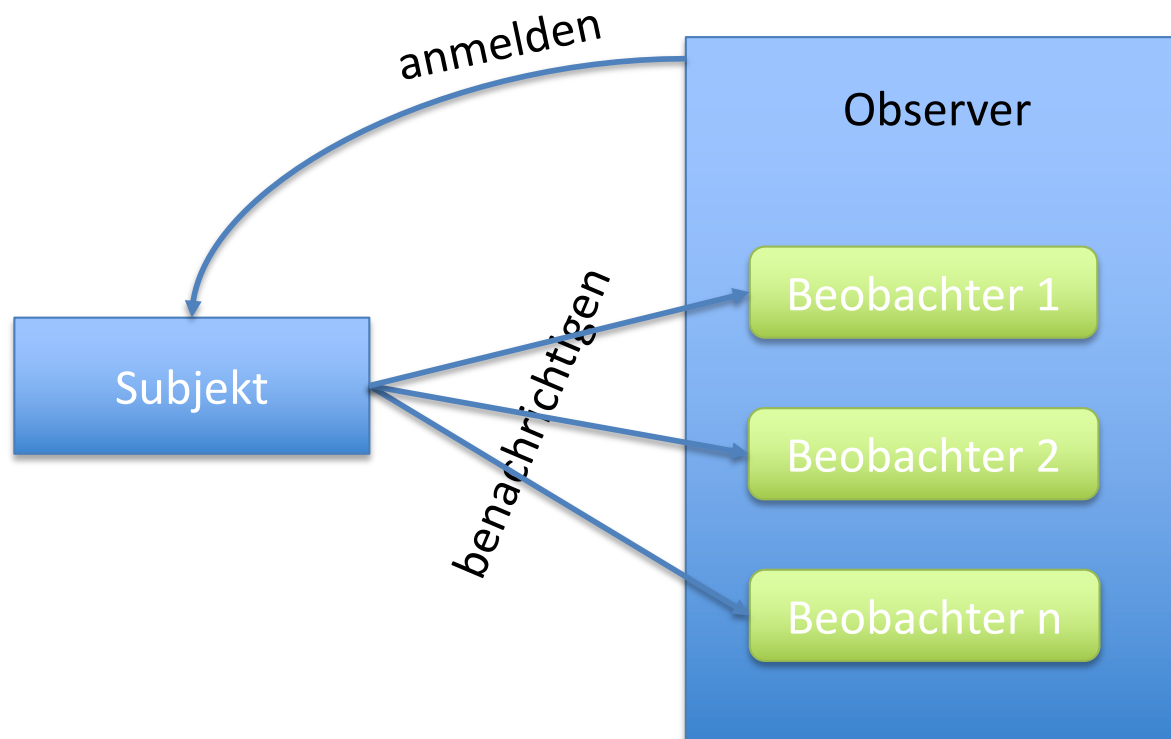
# Inhalt heute

- Das Beobachter Muster
- Ereignis-Ziel mit Delegates
- Ereignisquelle mit Event
- Ereignisse im .Net-Framework
- Die INotifyPropertyChanged Schnittstelle

# Beobachter Muster

- Im letzten Semester haben wir das **Beobachter-Muster** kennen gelernt.
  - Mit dessen Hilfe konnten wir Beobachter **benachrichtigen**, sobald bestimmte **Ereignisse** eingetreten waren.
  - Wir konnten z.B. verschiedene Beobachter darüber informieren, dass ein neues Objekt in die Mitgliederliste unserer Vereinsverwaltung eingefügt wurde.
- Dazu mussten zwei Schnittstellen implementiert werden.
  - Beim beobachteten Subjekt sorgte **IObservable** dafür, dass man Beobachter an-/abmelden und eine Benachrichtigung absetzen konnte.
  - Bei den Beobachtern sorgte **IObserver** dafür, dass eine Benachrichtigung auch an eine Methode weitergeleitet werden konnte.

# Beobachter Muster



# Grafische Benutzeroberflächen

- Ereignisse treten in **grafischen Benutzeroberflächen (GUIs)** sehr oft ein.
  - Wir wollen benachrichtigt werden, wenn z.B. ein Button geklickt wird.
  - Da wir demnächst solche GUIs erstellen wollen, müssen wir uns überlegen, wie man effektiv mit Ereignissen umgeht.
- Auf der .Net-Plattform gibt es neben dem Beobachter-Muster noch eine andere, bessere Möglichkeit, Ereignisse zu behandeln.
  - Mit Hilfe von Delegaten können wir Zeiger auf Methoden realisieren.
  - Mit Hilfe von Events können wir Ereignisse typsicher anbieten.
- Diese Sprachmittel werden an vielen Stellen des .Net-Frameworks genutzt.
  - Insbesondere bei den GUI-Technologien WPF und Windows Forms.
  - Aber auch an vielen anderen Stellen.

# Delegaten

- Sehen wir uns zunächst die sog. Delegaten (vom engl. Wort für Delegierter) an.
  - Ein Delegat ist ein **Datentyp**, der einen **Zeiger auf eine Methode** beschreibt.
- Aus einem Delegat kann ein Delegat-Objekt erzeugt werden, der auf eine oder mehrere Methode zeigt.
  - Mit Hilfe von Delegaten können Funktionen/Methoden wie Variablen übergeben werden.
- Dies hilft uns, wenn wir bei einem Ereignis festlegen wollen, wer überhaupt benachrichtigt werden soll.

# Delegaten definieren

- Ein Delegat wird mit Hilfe des Schlüsselwortes **delegate** definiert.
  - Delegaten sind typsichere Methodenzeiger.
  - Der folgende Delegat mit dem Namen `CalculateHandler` darf ausschließlich auf Methoden zeigen, die einen `int`-Wert als Antwort liefern und zwei `int`-Werte als Parameter erwarten:

```
public delegate int CalculateHandler(int a, int b);
```

- Ein Delegat definiert einen neuen Datentyp.
  - Die Definition eines Delegaten darf überall dort stehen, wo auch die Definition einer Klasse stehen darf.
  - Ein Delegat ist selber noch keine Variable/Objekt, sondern eine Art Vorlage dafür.



# Delegat-Objekte erzeugen

- Aus einem Delegaten erzeugt man Delegaten-Objekte.
  - Dies funktioniert genau so, wie bei einer Klasse.
  - Schritt 1: Eine Objektvariable definieren.
  - Schritt 2: Mit Hilfe von „new“ ein Exemplar erzeugen.
- Der Delegat besitzt genau einen Konstruktor.
  - Dieser Konstruktor erwartet nur einen Parameter.
  - Den Namen der Methoden, auf den das Objekt verweisen soll.
  - Die Methode muss natürlich eine kompatible Signatur aufweisen.

```
CalculateHandler c = new CalculateHandler(sum);
```

- Es geht aber sogar noch einfacher:

```
CalculateHandler c = sum;
```

# Aufgabe

- Es soll ein Taschenrechner programmiert werden.
  - Es werden zwei Zahlen von der Konsole eingelesen.
  - Anschließend soll der Operator eingegeben werden:  
+, -, \*, ...
  - Das Ergebnis soll in Abhängigkeit der Rechenmethode berechnet und ausgegeben werden.
- Es soll ein entsprechender Delegat definiert und benutzt werden.

# Lösung

```
public delegate int CalculateHandler(int a, int b);
```

Delegat definieren.

```
class Program
```

```
{  
    public static int sum(int a, int b) { return a + b; }  
    public static int diff(int a, int b) { return a - b; }  
    public static int mult(int a, int b) { return a * b; }
```

```
    static void Main(string[] args)
```

```
    {  
        int a = Convert.ToInt32(Console.ReadLine());  
        int b = Convert.ToInt32(Console.ReadLine());  
        string method = Console.ReadLine();
```

```
        CalculateHandler calculate = sum;
```

Delegat-Objekt erzeugen.

```
        switch (method)  
        {  
            case "-": calculate = diff; break;  
            case "*": calculate = mult; break;  
        }
```

```
        Console.WriteLine(calculate(a, b));  
        Console.ReadLine();
```

Delegat aufrufen.

```
    }  
}
```

# Delegat-Objekt als Parameter

- Ein Exemplar eines Delegaten ist ein ganz normales Objekt.
  - Es besitzt die Methoden Equals, ToString, ...
- Ein solches Objekt kann als Parameter oder Ergebnis an/von einer Methode übergeben werden.
  - Dadurch können Funktionen/Methoden als Parameter an Methoden übergeben werden.
- Eine Methoden „rechne“ kann z.B. ein Delegat-Objekt erwarten, die es dann selber zum Rechnen aufruft:

```
public static void rechne(CalculateHandler calculator, int a, int b)
{
    Console.WriteLine(calculator(a, b));
}
```

# Multicast Delegaten

- Ein Exemplar eines Delegaten kann nicht nur auf eine einzige Methode verweisen.
  - Mit Hilfe des Operators += lassen sich an ein Delegaten-Objekt noch weitere Methoden-Verweise anhängen.

```
calculate += new CalculateHandler(diff);  
  
// Oder noch einfacher:  
calculate += sum;
```

- Dadurch entstehen sog. Multicast-Delegates.
  - Beim Aufruf des Delegaten werden **alle Methoden** in der entsprechenden Reihenfolge aufgerufen.
  - Nur das Ergebnis der letzten Methode wird aber zurückgegeben.

# Anonyme Methoden

- Delegat-Objekte können auf Methoden verweisen.
  - Bislang waren dies immer Methoden, die wir zuvor normal definiert hatten.
- Delegat-Objekte können aber auch auf **anonyme Methoden** verweisen.
  - Anonyme Methoden wurden nicht vorher definiert und besitzen keinen Namen.
- Anonyme Methoden können Parameter erwarten und Ergebnisse liefern.
  - Zusätzlich können sie aber auf die lokalen Variablen des aktuellen Code-Blocks zugreifen.
  - Daher dürfen die Parameter nicht den Namen von bereits definierten lokalen Variablen tragen.

# Anonyme Methoden benutzen

- Anonyme Methoden werden ebenfalls mit dem Schlüsselwort **delegate** definiert.
  - Es muss zuvor ein passender Delegat definiert worden sein.
- Das folgende Delegat-Objekt verweist auf eine anonyme Methode.
  - Die Methode bekommt zwei int-Zahlen als Parameter geliefert und liefert die Summe der beiden Zahlen als Ergebnis zurück.

```
CalculateHandler calculate = delegate (int x, int y) { return x + y; };
```

- Das daraus entstandene Delegat-Objekt kann nun benutzt werden, um die anonyme Methode aufzurufen.

```
var ergebnis = calculate(5, 10);
```

# Anpassen der Lösung

- Mit Hilfe anonymer Methoden können wir unsere Lösung verschlanken:

```
public delegate int CalculateHandler(int a, int b);

class Program
{
    static void Main(string[] args)
    {
        int a = Convert.ToInt32(Console.ReadLine());
        int b = Convert.ToInt32(Console.ReadLine());
        string method = Console.ReadLine();

        CalculateHandler calculate = delegate (int x, int y) { return x + y; };

        switch (method)
        {
            case "-": calculate = delegate(int x, int y) { return x - y; } ; break;
            case "*": calculate = delegate(int x, int y) { return x * y; }; break;
        }

        Console.WriteLine(calculate(a, b));
        Console.ReadLine();
    }
}
```



# Ereignisse mit event

- Mit Hilfe von Delegaten kann ein Methoden-Zeiger in C# typsicher definiert werden.
  - Dadurch können wir uns das **Ziel einer Benachrichtigung** merken.
  - Eine solche Methode wird auch als **Ereignis-Routine (engl. event handler)** bezeichnet.
- Wir könnten nun in einer Klasse ein Delegat-Objekt öffentlich zugänglich machen.
  - Dadurch können sich dann Beobachter an dieses Objekt anhängen.
  - Entsprechend können wir diese Beobachter dann im Falle eines Ereignisses benachrichtigen, indem wir das Delegat-Objekt aufrufen.

# Aufgabe

- Wir wollen eine Klasse Bankkonto erstellen.
  - Ein Bankkonto hat einen (nur lesbaren) Kontostand.
  - Auf ein Konto kann man Geld einzahlen und abheben.
- Wenn sich der Kontostand ändert, soll es möglich sein, dass Beobachter darüber informiert werden.
- Wir wollen dies nun mit Hilfe von Delegation umsetzen.

# Bankkonto

Der Delegat definiert, wie das Ziel der Benachrichtigung aussehen darf.

```
public delegate void KontostandGeandertHandler(double neuer_kontostand);
```

```
class Bankkonto
```

```
{
    private double kontostand = 0;
```

```
    public double Kontostand
```

```
{
        get { return kontostand; }
    }
```

```
    public KontostandGeandertHandler KontostandGeandert;
```

```
    public void einzahlen(double betrag)
```

```
{
        kontostand += betrag;
        KontostandGeandert(kontostand);
    }
```

```
    public void abheben(double betrag)
```

```
{
        kontostand -= betrag;
        KontostandGeandert(kontostand);
    }
```

```
}
```

Ein Objekt vom Typ des Delegaten ist öffentlich zugänglich, damit sich Beobachter anmelden können.

Beobachter werden über den neuen Kontostand informieren.

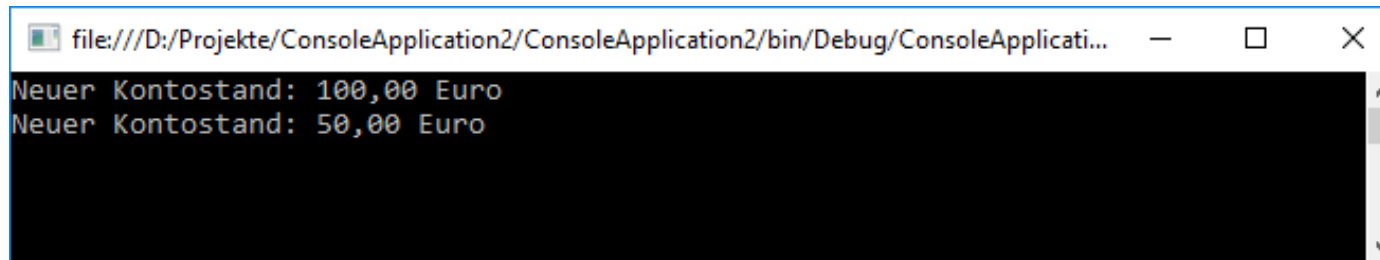
# Nutzung des Bankkontos

```
class Program
{
    static void Main(string[] args)
    {
        Bankkonto k = new Bankkonto();
        k.KontostandGeandert += Kontostand_Aenderung_eingetreten;

        k.einzahlen(100);
        k.abheben(50);

        Console.ReadLine();
    }

    private static void Kontostand_Aenderung_eingetreten(double neuer_kontostand)
    {
        Console.WriteLine("Neuer Kontostand: {0:F2} Euro", neuer_kontostand);
    }
}
```

A screenshot of a Windows console application window. The title bar shows the file path: file:///D:/Projekte/ConsoleApplication2/ConsoleApplication2/bin/Debug/ConsoleApplicati... The console output displays two lines: "Neuer Kontostand: 100,00 Euro" and "Neuer Kontostand: 50,00 Euro". The text is in a monospaced font, with the first line in green and the second line in red. The window has standard Windows controls (minimize, maximize, close) in the top right corner.

```
file:///D:/Projekte/ConsoleApplication2/ConsoleApplication2/bin/Debug/ConsoleApplicati...
Neuer Kontostand: 100,00 Euro
Neuer Kontostand: 50,00 Euro
```

# Bewertung der Lösung

- Dies ist kein guter Weg!
  - Das Delegat-Objekt ist im Bankkonto öffentlich zugänglich.
  - Wir verstoßen damit gegen das **Geheimnisprinzip**.
  - Wir können von außen dem Delegat-Objekt verbotene Werte zuweisen, z.B. null.
- Wir könnten nun eine Eigenschaftsmethode erstellen.
  - Diese kapselt den Zugriff auf das Delegat-Objekt.
  - Es wird dadurch sichergestellt, dass keine unerlaubten Werte zugewiesen werden.
- Es geht aber auch einfacher.
  - Es existiert das Schlüsselwort **event**.
  - Ein event ist vom Typ eines Delegaten und Teil einer Klassendefinition.
- Das Schlüsselwort event erstellt quasi eine Eigenschaftsmethode zu einem Delegat-Objekt.
  - Es wird sichergestellt, dass dem Delegat-Objekt kein unerlaubter Wert zugewiesen werden kann.

# Lösung anpassen

```
public delegate void KontostandAenderungHandler(double neuerBetrag);
```

```
class Bankkonto
```

```
{
```

```
    private double kontostand = 0;
```

```
    public event KontostandAenderungHandler KontostandGeaendert;
```

```
    public void einzahlen(double betrag)
```

```
    {
```

```
        kontostand += betrag;
```

```
        KontostandGeaendert(kontostand);
```

```
    }
```

```
    public void abheben(double betrag)
```

```
    {
```

```
        kontostand -= betrag;
```

```
        KontostandGeaendert(kontostand);
```

```
    }
```

```
}
```

Das Bankkonto bietet eine Ereignisquelle namens KontostandGeaendert an.

Das Ereignis wird ausgelöst, wenn sich der Kontostand ändert.

# Ereignisse abonnieren

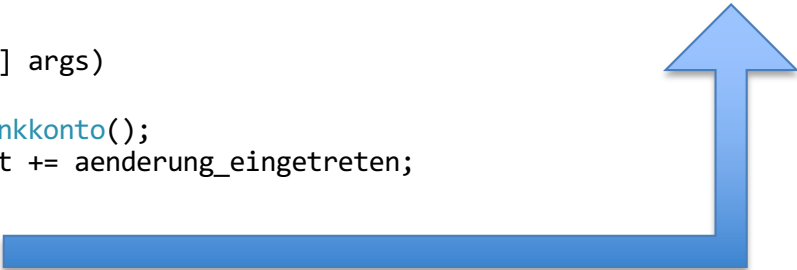
- Ein Bankkonto-Objekt kann nun über Änderungen am Kontostand informieren.
  - Dazu muss der Ereignisquelle lediglich ein Delegat auf eine Methode angehängt werden.

```
class Program
{
    public static void aenderung_eingetreten(double neuer_betrag)
    {
        Console.WriteLine("Kontostand geändert! Neuer Betrag={0}", neuer_betrag);
    }

    static void Main(string[] args)
    {
        Bankkonto k = new Bankkonto();
        k.KontostandGeaendert += aenderung_eingetreten;

        k.einzahlen(100);
        k.abheben(50);

        Console.ReadLine();
    }
}
```



# Eigenschaften von Ereignissen

- Ereignisse müssen von keinem Beobachter abonniert werden.
  - Wenn ein Ereignis ausgelöst wird, werden diejenigen Delegaten-Objekte aufgerufen, die zuvor angemeldet wurden.
  - Das können im Prinzip beliebig viele sein, also auch keine.
- Ereignisse können (dank der Delegaten) Parameter und Ergebnisse definieren, müssen es aber nicht.
  - Es hängt ganz davon ab, ob dies sinnvoll ist.
  - Man legt dies mit Hilfe des Delegaten fest, der zur Definition der Ereignisquelle gehört.
- Ereignisse (events) können Teil einer Schnittstelle sein.
  - Dadurch wird dann ebenfalls sicher gestellt, dass die implementierende Klasse über dieses Ereignis verfügt.



# Ereignisse im Visual Studio

- Das Visual Studio hilft uns dabei, Ereignisse zu abonnieren.
  - Eine zum Delegaten der Ereignisquelle passende Methode kann automatisch generiert werden.
- Beim Zugriff auf eine Ereignisquelle muss nur die Tab-Taste gedrückt werden.

```
class Program
{
    static void Main(string[] args)
    {
        Bankkonto k = new Bankkonto();
        k.KontostandAenderung += |
        k.einzahlen(100);
        k.abheben(50);
        Console.ReadLine();
    }
}
```

K\_KontostandAenderung;(Zum Einfügen TAB-TASTE drücken)

# Ereignis-Abonnement aufheben

- Das Abonnement eines Ereignisses kann auch wieder aufgehoben werden.
  - Entsprechend entfernt man das Delegaten-Objekt der Ereignis-Routine wieder aus der Ereignisquelle.
  - Anschließend wird man nicht mehr über eintretende Ereignisse informiert.
- In unserem Beispiel wollen wir evtl. nicht mehr über Änderungen am Kontostand informiert werden.

```
k.KontostandAenderung -= aenderung_eingetreten;
```

# Ereignis-Routinen im .Net-Framework

- Im .Net-Framework weisen alle Ereignis-Routinen eine gemeinsame Struktur auf.
  - Die Struktur wird durch die entsprechenden Delegaten definiert.
- Die Ereignis-Routinen besitzen grundsätzlich zwei Parameter.
  - Der erste Parameter ist das Objekt, welches das Ereignis ausgelöst hat.
  - Der zweite Parameter kapselt ereignisspezifische Daten, z.B. die Höhe des Betrags einer Einzahlung oder Abbuchung.
- Für die ereignisspezifischen Daten wird meist eine eigene Klasse definiert, die von der Klasse **EventArgs** erbt.
  - Dort werden dann notwendige Datenfelder hinzugefügt.

# Ereignisdaten abbilden

- Wir wollen unsere Bankkonto-Lösung gemäß des .Net-Frameworks anpassen.
  - Dafür erstellen wir zunächst eine neue Klasse, um die ereignisspezifischen Informationen abbilden zu können.

```
public class BankkontoEventArgs : EventArgs
{
    public double Betrag { get; set; }
    public bool Einzahlung { get; set; }
    public bool Abbuchung { get; set; }

    public BankkontoEventArgs(double betrag, bool einzahlung)
    {
        Betrag = betrag;
        Einzahlung = einzahlung;
        Abbuchung = !einzahlung;
    }

    public override string ToString()
    {
        return string.Format((Einzahlung ? "Einzahlung" : "Auszahlung") + " von {0:F2} Euro.", Betrag);
    }
}
```

# Delegate anpassen

- Als nächstes passen wir unseren Delegaten so an, dass er den Vorgaben entspricht.

```
public delegate void KontostandAenderungHandler(object sender, BankkontoEventArgs e);
```

- Entsprechend müssen wir auch in unserer Bankkonto-Klasse das Ereignis anders auslösen:

```
public void einzahlen(double betrag)
{
    kontostand += betrag;
    KontostandAenderung(this, new BankkontoEventArgs(betrag, true));
}

public void abheben(double betrag)
{
    kontostand -= betrag;
    KontostandAenderung(this, new BankkontoEventArgs(betrag, false));
}
```

# Auf Ereignis reagieren

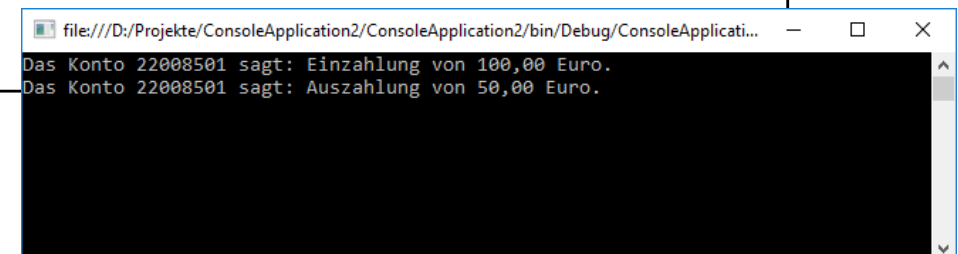
- Das Ereignis des Bankkontos können wir wieder abonnieren.
  - Es werden nun aber wesentlich mehr Informationen mitgeliefert.

```
class Program
{
    static void Main(string[] args)
    {
        Bankkonto k = new Bankkonto();
        k.KontostandAenderung += Kontostand_Aenderung_eingetreten;

        k.einzahlen(100);
        k.abheben(50);

        Console.ReadLine();
    }

    private static void Kontostand_Aenderung_eingetreten(object sender, BankkontoEventArgs e)
    {
        Console.WriteLine("Das Konto " + sender.GetHashCode() + " sagt: " + e);
    }
}
```



```
file:///D:/Projekte/ConsoleApplication2/ConsoleApplication2/bin/Debug/ConsoleApplicati...
Das Konto 22008501 sagt: Einzahlung von 100,00 Euro.
Das Konto 22008501 sagt: Auszahlung von 50,00 Euro.
```

# INotifyPropertyChanged

- In den kommenden Vorlesungen werden wir grafische Benutzeroberflächen gestalten.
  - In den Oberflächen werden oft die Daten aus Objekten angezeigt, z.B. der Kontostand eines Bankkontos.
- Wir werden noch sehen, dass die Benutzeroberfläche sich selbstständig aktualisieren kann, wenn Änderungen eintreten.
  - Dies wird als **Datenbindung (engl. data binding)** bezeichnet.
- In diesem Zusammenhang müssen die Objekte eine bestimmte Schnittstelle implementieren.
  - Die Schnittstelle **INotifyPropertyChanged** sorgt dafür, dass Objekte ein bestimmtes Ereignis werfen, wenn sich Eigenschaften ändern.
  - Dieses Ereignis wird durch die Benutzeroberfläche abonniert und zur Aktualisierung der Daten benutzt.

# INotifyPropertyChanged implementieren

- Wir wollen unsere Bankkonto-Klasse erneut anpassen.
  - Dazu erstellen wir eine nur lesbare Eigenschaftsmethode „Kontostand“.
  - Daneben implementieren wir die Schnittstelle INotifyPropertyChanged und werfen das Ereignis, wenn nötig.
- Das Ereignis liefert wiederum zwei Parameter mit.
  - Der erste Parameter ist das auslösende Objekt, also ein Bankkonto-Objekt.
  - Der zweite Parameter ist vom Typ PropertyChangedEventArgs, in dessen Objekt man den Namen der geänderten Eigenschaft übergibt.
- Dadurch kann die Oberfläche die geänderte Eigenschaft auslesen und so die Daten aktualisieren.



# Bankkonto anpassen

```
class Bankkonto : INotifyPropertyChanged
{
    private double kontostand = 0;

    public double Kontostand
    {
        get { return kontostand; }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    public void einzahlen(double betrag)
    {
        kontostand += betrag;
        PropertyChanged(this, new PropertyChangedEventArgs("Kontostand"));
    }

    public void abheben(double betrag)
    {
        kontostand -= betrag;
        PropertyChanged(this, new PropertyChangedEventArgs("Kontostand"));
    }
}
```

Das Ereignis  
PropertyChanged wird  
durch die Schnittstelle  
vorgegeben.

Bei Änderungen wird  
das Ereignis ausgelöst.

# Wir haben heute gelernt...

- Wie man Ereignisse mit Hilfe des Beobachter Musters umsetzen konnte.
- Wie man ein Ereignis-Ziel mit Hilfe von Delegaten realisieren kann.
- Wie man Ereignisquellen mit dem Schlüsselwort event umsetzt.
- Wie Ereignisse im .Net-Framework umgesetzt werden.
- Warum die Schnittstelle INotifyPropertyChanged benutzt wird, um Änderungen zu signalisieren.