HOCHSCHULE HAMM-LIPPSTADT

Praktische Informatik

Vorlesung 11

Xamarin

Zuletzt haben wir gelernt…

- Wie man mit dem ItemsControl eine Menge von Elementen anzeigt.
- Wie man über ein DataTemplate festlegt, wie ein einzelnen Element dargestellt wird.
- Wie man mit dem ItemsPanel definiert, in welchem LayoutContainer die Elemente abgelegt werden.
- Was die ListBox im Vergleich zum ItemsControl kann.
- Wie man mit der ComboBox umgehen kann.
- Wie man tabellarische Daten mit der ListView anzeigt.
- Wie man hierarchische Daten mit einem TreeView anzeigt.

Inhalt heute

- Xamarin
- Mobile Platform und Xamarin.Forms
- Beispiel Temperaturumrechner
- ContentPage, Layouts und OnPlatform
- Element Binding und Ressourcen
- Beispiel Mitgliederverwaltung

Xamarin

- Bislang haben wir mit Hilfe von C# und WPF Desktopanwendungen für das Betriebssystem Windows erstellt.
  - Mit Hilfe von **Xamarin** ist es aber möglich, mit ähnlichen Technologien auch andere Plattformen zu bedienen.

- Xamarin war ursprünglich eine eigenständige Firma mit Sitz in San Francisco.
  - Im Jahr 2014 wurde die Firma von Microsoft gekauft.

- Bei Xamarin arbeitet das Kernteam der Entwickler des Mono-Frameworks.
  - Mono ist eine plattformunabhängige Variante des .Net Frameworks.

Mobile Platform und Forms

- Neben Mono sind die wichtigsten Produkte von Xamarin die **Mobile Platform** und **Xamarin.Forms**.
- Die Xamarin Mobile Platform erlaubt es den Entwicklern mit C# Apps für iOS und Android zu entwickeln.
  - Dabei werden die plattformspezifischen SDKs in C# angesprochen und nicht in Swift oder Java.
- Xamarin.Forms geht noch einen Schritt weiter.
  - Mit diesem Framework werden die spezifischen Eigenheiten der mobilen Plattformen hinter eigenen Klassen verborgen.
- Xamarin.Forms ist zu WPF sehr ähnlich.
  - Es setzt ebenfalls auf eine Trennung von Oberfläche (XAML) und Code.
  - Es gibt aber auch viele Unterschiede.

Xamarin.Forms

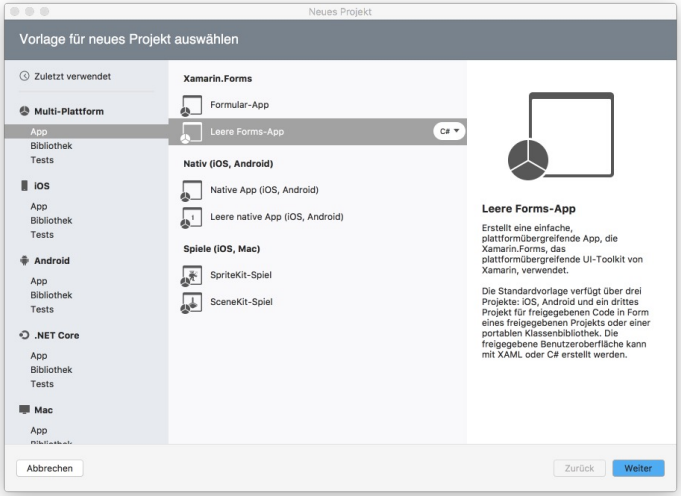
Achtung: Die Entwicklung von Apps für iOS setzt zwingend einen Mac voraus!

- Um mit Xamarin.Forms arbeiten zu können, muss eine entsprechende Entwicklungsumgebung installiert werden.
  - Unter Windows das Visual Studio mit den entsprechenden Erweiterungen.
  - Unter OS X das Visual Studio for Mac.

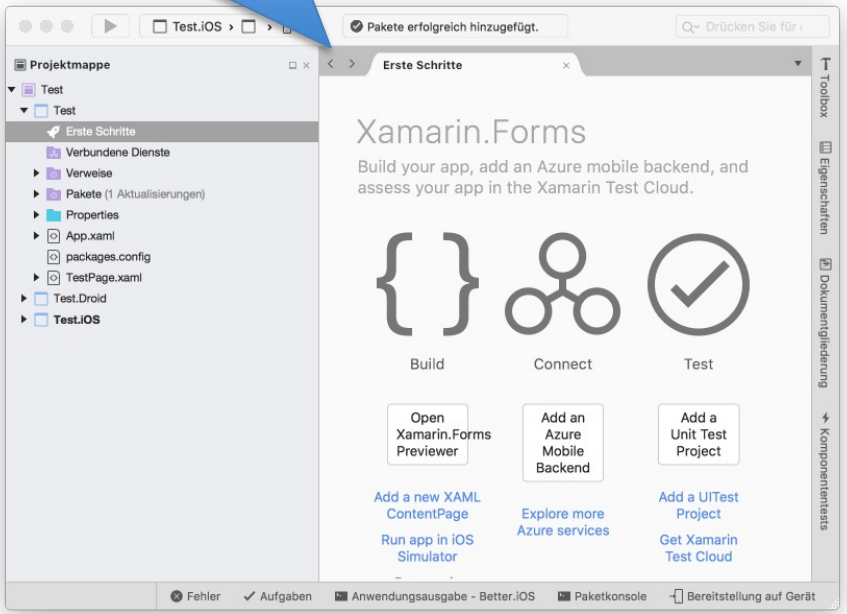
- Apps können dann sowohl lokal im Simulator getestet, als auch in den App Stores hinterlegt werden.
  - Wir werden in den folgenden Beispielen auf dem Mac entwickeln.
  - Der Programmcode ist aber für Windows identisch.

Neues Projekt erstellen

Für Xamarin.Forms muss die Projektvorlage in der Kategorie „Multi-Plattform“ mit dem Namen „Leere Forms-App“ ausgewählt werden.



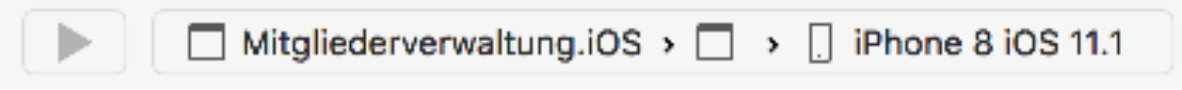
Als Ergebnis entstehen gleich **drei** Projekte. Ein Forms-Projekt und jeweils ein plattformspezifisches Projekt für iOS und Android.



Projekt starten

- Eines der beiden plattformspezifischen Projekte muss als Startprojekt ausgewählt sein.
  - Auf dem Mac ist dies natürlich das iOS Projekt.

- Am oberen Rand der Entwicklungsumgebung kann dann das Zielsystem ausgewählt werden.
  - Dies ist zunächst meist ein Simulator, z.B. iPhone 8 mit iOS 11.1.



- Wenn ein mobiles Endgerät an den Rechner angeschlossen ist, kann die App aber auch dort ausgeführt werden.

Temperatur Umrechner

- Wir erinnern uns an die Anwendung Temperaturumrechner.
  - Dort haben wir in WPF dafür gesorgt, dass wir Temperaturen von Celsius in Fahrenheit umrechnen konnten.

- Wir erstellen dafür ein neues Projekt „**Converter**“.
  - Für Xamarin.Forms Entwickler ist vor allem die sog. **portable Klassenbibliothek** wichtig.
  - Dort wird der C#-Code und alle XAML-Dateien abgelegt, die sowohl für iOS als auch für Android genutzt werden.

- Ein neues Xamarin.Forms Projekt ist nicht gänzlich leer.
  - Das Projekt enthält die Dateien App.xaml und App.xaml.cs als Einsprungspunkt in die Anwendung.
  - Zudem ist mit den Dateien ConverterPage.xaml und ConverterPage.xaml.cs bereits eine erste Seite vorhanden.

ContentPage

- Die Datei ConverterPage ist die erste Seite, die in der App angezeigt wird.
  - Dies wird im Konstruktor der App.xaml.cs so definiert.

- Uns fällt auf, dass die Klasse `ConverterPage` nicht von Window ableitet.
  - Stattdessen leitet sie von der Klasse `ContentPage` ab.
  - Dies ist die Basisklasse für alle einfachen Views in Xamarin.Forms.

- Ansonsten kommt uns der XAML-Code in der Datei ConverterPage.xaml sehr bekannt vor.
  - Zunächst ist ein Label vorhanden, welches wir löschen können.

Layout

- Wir wollen nun auf unserer Oberfläche einen `Slider` und zwei `Labels` anzeigen.
  - Die Labels sollen jeweils den Wert in Celsius und Fahrenheit ausgeben.

- Auch in Xamarin.Forms müssen Interaktionselemente in Layout-Containern platziert werden.
  - Die Klassen `StackLayout` und `Grid` funktionieren genau so, wie bereits gesehen.

- Da auf mobilen Endgeräten viel weniger Platz zur Verfügung steht, als auf einem Desktop-System, stehen hier aber auch einige andere Layout-Container bereit.
  - `AbsoluteLayout` als Ersatz zum `Canvas`.
  - Zum noch `RelativeLayout`, `ScrollView`, ...

ConverterPage.xaml

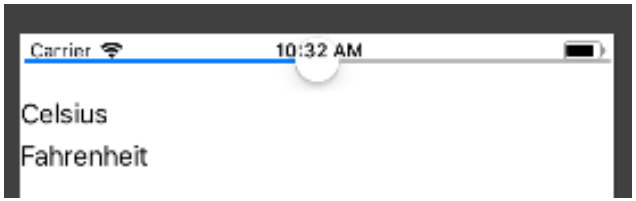
- Wir benutzen zunächst ein einfaches StackLayout:

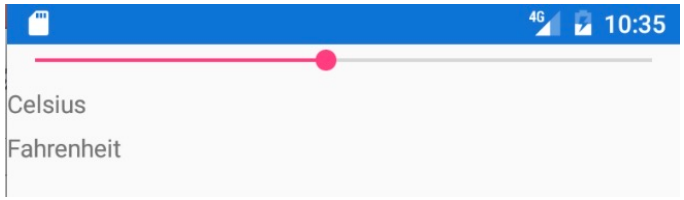
```
<StackLayout>
    <Slider x:Name="slider"
       Minimum="-100" Value="0" Maximum="100" />
    <Label Text="Celsius" />
    <Label Text="Fahrenheit" />
</StackLayout>
```

- Die Klassen `Slider` und `Label` lassen sich genau so konfigurieren, wie wir das aus der WPF gewohnt sind.
  - Damit das Data Binding später umgesetzt werden kann, geben wir dem `Slider` auch über `x:Name` einen Namen.

Plattformabhängigkeit

- Wir starten die Anwendung nun im Simulator.
  - Dabei stellen wir unter iOS fest, dass der `Slider` direkt am oberen Rand des Bildschirms angehängt ist und die Uhr überdeckt.
  - Unter Android wird hingegen ein Abstand zum Rand eingehalten.





- Wir haben es hier mit einer unterschiedlichen Darstellung der App auf den Plattformen zu tun.
  - Dies ist natürlich sehr ungünstig.
  - Xamarin stellt aber Konzepte bereit, um damit umzugehen.

OnPlatform

- Die Klasse `OnPlatform` bietet unter Xamarin.Forms die Möglichkeit, Werte in Abhängigkeit der aktuellen Plattform zu setzen.
  - Dies kann sowohl in XAML als auch im Code Behind genutzt werden.

- In unserem Fall wollen wir das `Padding` des `StackLayout` unter iOS am oberen Rand vergrößern.
  - Im XAML wird dies wie folgt umgesetzt:

```
<StackLayout.Padding>
    <OnPlatform x:TypeArguments="Thickness">
        <On Platform="iOS" Value="0, 20, 0, 0" />
    </OnPlatform>
</StackLayout.Padding>
```

Element Binding

- Als nächstes wollen wir das Element Binding umsetzen.
  - Der Wert des `Sliders` soll in den `Label`n angezeigt werden.
  - Dazu muss im ersten Label das Data Binding konfiguriert werden.

- Wir stellen allerdings fest, dass die Klasse Binding in Xamarin die Eigenschaft `ElementName` nicht unterstützt.
  - Stattdessen können wir aber die Eigenschaft BindingContext (in WPF DataContext) des Labels auf den Slider setzen.
  - Dadurch bezieht sich das Binding automatisch auf den Slider.

- Im Binding muss dann nur die Quell-Eigenschaft gesetzt werden.

```
<Label
   BindingContext="{x:Reference slider}"
   Text="{Binding Value, StringFormat='{0:F2} Grad Celsius'}" />
```

Convert-Klasse

- Im zweiten Label wollen wir nun den in Fahrenheit umgerechneten Wert anzeigen.
  - Dazu können wir, wie in WPF, eine Converter-Klasse erstellen.

- Diese Klasse muss wieder die Schnittstelle `IValueConverter` implementieren.
  - Die Klasse bietet die beiden Methoden `Convert` und `ConvertBack` an.
  - Wir können diese Klasse aus unserem bestehenden Projekt 1:1 importieren.

```
public class CelsiusToFahrenheitConverter : IValueConverter
{
    ...
}
```

Ressource definieren

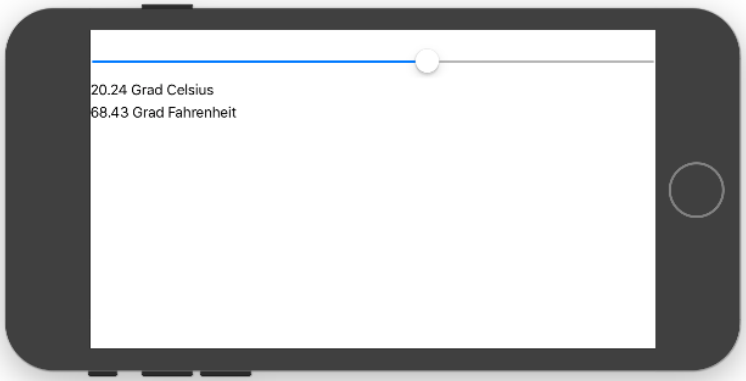
- Von der Converter-Klasse müssen wir nun ein Objekt als Ressource definieren.
  - Auch hier gibt es wieder einen kleinen Unterschied zu WPF.
  - Ressourcen müssen in einem `ResourceDictionary` definiert werden.

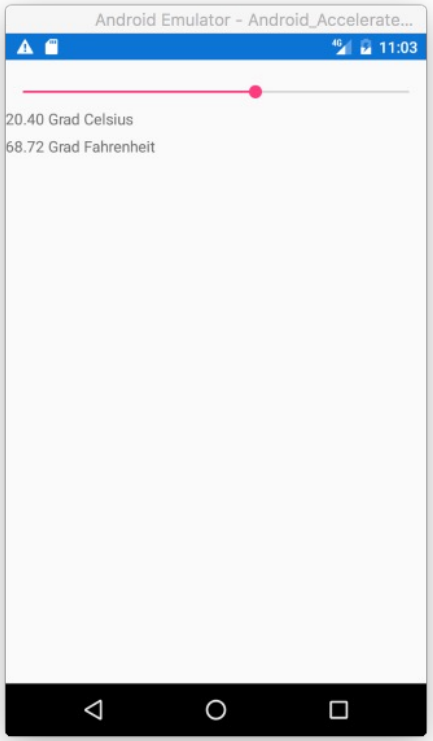
```
<ContentPage.Resources>
    <ResourceDictionary>
        <local:CelsiusToFahrenheitConverter x:Key="converter" />
    </ResourceDictionary>
</ContentPage.Resources>
```

- Anschließend können wir den Converter im Element Binding des zweiten Labels einsetzen:

```
<Label BindingContext="{x:Reference slider}“
Text="{Binding Value, Converter={StaticResource converter}}“ />
```

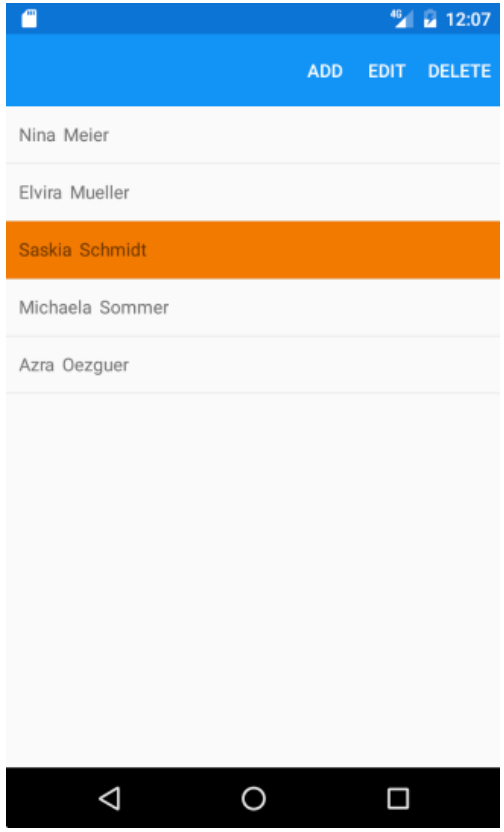
Ergebnis


20.24 Grad Celsius
68.43 Grad Fahrenheit

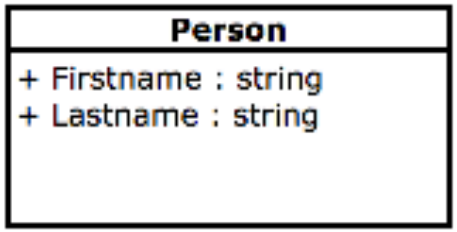

Android Emulator - Android_Accelerate...
11:03
20.40 Grad Celsius
68.72 Grad Fahrenheit

Mitgliederverwaltung

- Wir wollen noch ein zweites Beispiel erstellen.
  - Wir wollen in einer App die Mitglieder eines Sportvereins verwalten können.
- Die Mitglieder (Objekte) werden in der Oberfläche in einer Liste angezeigt.
  - Es soll zudem die Möglichkeit existieren, Mitglieder hinzuzufügen, zu bearbeiten und zu löschen.
- Wir erstellen dazu ein neues Xamari.Forms Projekt.



Datenmodell



- Wir wollen uns zunächst dem Datenmodell widmen.
  - Wir benötigen eine Klasse „Person“ mit entsprechenden Eigenschaften, wie z.B. Vorname und Nachname.

- Ein Objekt dieser Klasse soll später in einem eigenen Formular bearbeitet oder erstellt werden können.
  - Die Eigenschaften des Objekts (Vorname, Nachname, ...) werden dann mit Hilfe von Data Binding an Eingabefelder gebunden.
  - Alle Änderungen in den Eingabefeldern werden dadurch direkt mit dem Objekt synchronisiert.

- Die Klasse Person sollte auch die Schnittstelle INotifyPropertyChanged implementieren.
  - Dadurch kann ein Objekt dieser Klasse die Oberfläche über Änderungen informieren.

Problem mit Data Binding

- Beim Data Binding eines Person-Objektes zeigt sich in einer bestimmten Situation problematisches Verhalten.
  - Klickt der Benutzer im Formular auf Abbrechen, behält das Objekt die bereits durchgeführten Änderungen.

- Das Objekt muss dann eigentlich den Zustand vor den Änderungen wieder herstellen.
  - Diese Fähigkeit muss noch nachgerüstet werden.

IEditableObject

- Um diese Fähigkeit nachzurüsten existiert die Schnittstelle `IEditableObject`.
  - Diese Schnittstelle steht auch in WPF zur Verfügung.

- IEditableObject sorgt dafür, dass verschiedene Methoden angeboten werden müssen.
  - BeginEdit soll den alten Wert aller Eigenschaften zur späteren Rekonstruktion speichern.
  - CancelEdit soll den zuvor gespeicherten Zustand wieder herstellen.
  - EndEdit beendet den Editiervorgang und löscht die gespeicherten Werte.

Ausschnitt aus Klasse Person

```
public class Person : INotifyPropertyChanged, IEditableObject
{
    ...

    public string Firstname
    {
        get { return firstname; }
        set
        {
            firstname = value;
            PropertyChanged?.Invoke(this, ...);
        }
    }

    public void BeginEdit()
    {
        firstname_backup = firstname;
        lastname_backup = lastname;
        is_editing = true;
    }

    ...
```

PersonViewModel

- Die Klasse Person reicht noch nicht aus.
  - Wir benötigen noch eine Klasse, um eine Liste von Person-Objekten zu verwalten.
  - Die Liste ist vom Type `ObservableCollection`, so dass Änderungen an der Liste auch in der Oberfläche durchschlagen.

- Zudem soll in der Klasse das aktuell ausgewählte Objekt vorgehalten werden.
  - Entsprechend wir eine Eigenschaftsmethode „Selected“ bereitgestellt, die das aktuelle Person-Objekt vorhält.

- Wir nennen diese Klasse **PersonViewModel**.
  - Ein Objekt dieser Klasse dient als Quelle des Data Binding in unserer Liste.

Ausschnitt aus PersonViewModel

```
public class PersonViewModel : INotifyPropertyChanged
{
    private Person selected = null;

    public PersonViewModel()
    {
        Members = new ObservableCollection<Person>();
    }

    public ObservableCollection<Person> Members
    {
        get; private set;
    }

    ...
}
```

Datenmodell anbinden

- Nachdem das Datenmodell fertiggestellt ist, kann nun die Oberfläche erstellt werden.
  - Im Code Behind der Startseite wird der `BindingContext` auf ein Objekt der Klasse `PersonViewModel` gesetzt.
  - Zusätzlich fügen wir testweise ein neues Objekt in die Liste ein.

```
public partial class MitgliederverwaltungPage : ContentPage
{
    private PersonViewModel model = new PersonViewModel();

    public MitgliederverwaltungPage()
    {
        InitializeComponent();
        model.Members.Add(new Person() {
          Firstname="Maria", Lastname="Oezguer" });
        BindingContext = model;
    }
}
```

XAML

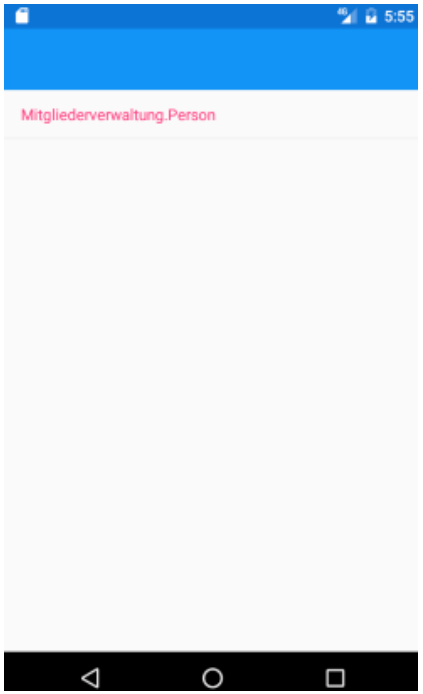
- Im XAML der Startseite kann das Datenmodell nun per Data Binding angebunden werden.
  - Die Person-Objekte sollen in einer ListView angezeigt werden.

- Die Eigenschaften `ItemSource` und `SelectedItem` werden über Data Binding mit dem Datenmodell verbunden.

```
<ListView
   ItemsSource="{Binding Members}"
   SelectedItem="{Binding Selected, Mode=TwoWay}">
</ListView>
```

DataTemplate

- In dieser Konfiguration zeigt die `ListView` leider nur den Klassennamen des Objektes in der Liste an.
  - Wie wir bereits gelernt haben, muss eine ListView über ein Template erst gezeigt bekommen, wie ein Person-Objekt dargestellt werden soll.
  - Die Syntax dazu ist in Xamarin sehr ähnlich zu WPF.



- Der Eigenschaft `ItemTemplate` wird ein Objekt vom Type `DataTemplate` zugewiesen.
  - In unserem Fall besteht das Template aus einem einfachen StackLayout mit zwei Labeln.

Vollständiges ListView

```
<ListView
   ItemsSource="{Binding Members}"
   SelectedItem="{Binding Selected, Mode=TwoWay}">
   <ListView.ItemTemplate>
      <DataTemplate>
         <ViewCell>
            <StackLayout Orientation="Horizontal">
               <Label Text="{Binding Firstname}" />
               <Label Text="{Binding Lastname}" />
            </StackLayout>
         </ViewCell>
      </DataTemplate>
   </ListView.ItemTemplate>
</ListView>
```

Toolbar

- Zuletzt wollen wir eine Toolbar einfügen.
  - Dazu kann der Eigenschaft ToolbarItems im XAML der Startseite eine Liste von ToolbarItem-Objekten hinzugefügt werden.

- Exemplarisch im Folgenden für den Edit-Button:

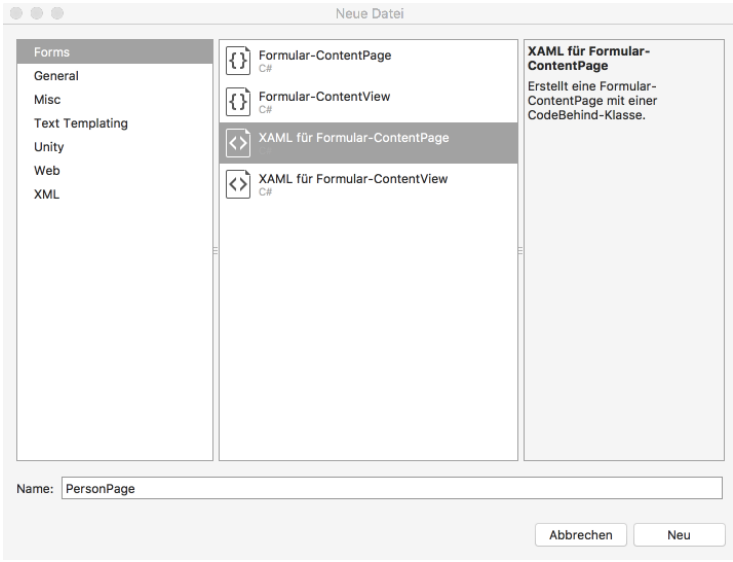
```
<ContentPage.ToolbarItems>
   <ToolbarItem Name="Edit"
      Clicked="Edit_Clicked"
      IsEnabled="{Binding IsSelected}"/>
</ContentPage.ToolbarItems>
```

Der Button ist nur dann aktiviert, wenn auch ein Element ausgewählt ist.

- Jedem ToolbarItem wird auch ein Event-Handler zugewiesen.
  - Dieser wird aufgerufen, sobald der entsprechend Button geklickt wird.

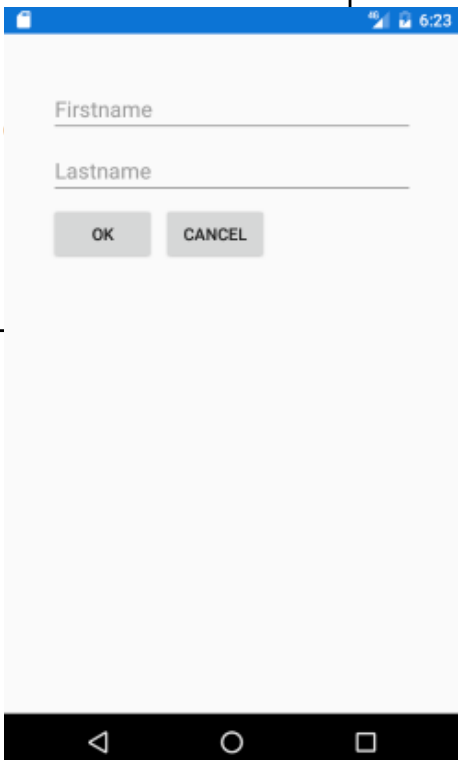
PersonPage

- Damit wir ein Person-Objekt bearbeiten können, benötigen wir ein eigenes Formular.
- Wir fügen dazu dem Projekt eine neue ContentPage hinzu.
- Im XAML der neuen Seite kann dann ein Formular zur Bearbeitung eines Person-Objektes aufgebaut werden.



XAML der PersonPage

```
<ContentPage
    ...
    <StackLayout Padding="40,40,40,40">
        <Entry Placeholder="Firstname" Text="{Binding Firstname}"/>
        <Entry Placeholder="Lastname" Text="{Binding Lastname}"/>
        <StackLayout Orientation="Horizontal">
            <Button Text="Ok" Clicked="Ok_Clicked" />
            <Button Text="Cancel" Clicked="Cancel_Clicke
        </StackLayout>
    </StackLayout>
</ContentPage>
```



Navigation in Xamarin

- In WPF würden wir nun die Methoden Show oder ShowDialog benutzen.
  - Es würde dann ein weiteres Fenster angezeigt.
- In Xamarin geht dies so nicht.
  - Es ist nicht genug Platz vorhanden, um ein zusätzliches Fenster auf dem Bildschirm anzuzeigen.
- Wir können aber zu einer anderen Seite (Klasse Page) navigieren.
  - Das Konzept ist ähnlich zu Web-Seiten im Browser.
  - Mann kann eine neue Seite auf den Navigationsstack legen und navigiert dadurch zu dieser Seite.
  - Entsprechend kann auch eine Seite vom Navigationsstack entnommen werden und man navigiert wieder zurück.
- Die Klasse Navigation bietet entsprechende statische Methoden an.
  - **PushAsync** → legt eine neue Seite auf den Navigationsstack
  - **PopAsync** → entnimmt eine Seite vom Navigationsstack

Edit_Clicked

- Das Formular `PersonPage` wollen wir anzeigen, sobald ein Nutzer die Buttons Add oder Edit klickt.
  - Dazu navigieren wir mit PushModalAsync zu dieser Seite.

- Dem Konstruktor der Klasse übergeben wir das zu bearbeitende Objekt.
  - Dieses wird dann per Data Binding angebunden.

```
private async void Add_Clicked(object sender, System.EventArgs e)
{
    var obj = new Person();
    var page = new PersonPage(obj);
    await Navigation.PushModalAsync(page);
}
```

Wir haben heute gelernt…

- Wie die Plattform Xamarin funktioniert.
- Wie man mit Xamarin.Forms plattformunabhängige Apps entwickeln kann.
- Die Unterschiede zu WPF anhand des Beispiels Temperaturumrechner.
- Was die Klasse ContentPage, Layouts und die Klasse OnPlatform leisten.
- Wie man in Xamarin.Forms mit Element Binding und Ressourcen umgeht.
- Weitere Unterschiede zur WPF anhand des Beispiels Mitgliederverwaltung.