



Trainee.Project Deutsche Version

Keppler.Passwort.Generator



Für dein Praktikumsprojekt wollen wir einen Passwort-Generator mit Keppler.Systems. Benutzeroberfläche programmieren. Diesem geben wir den Namen "Keppler.Passwort.Generator".

Für das Programmieren unseres Keppler.Passwort.Generators benutzen wir die Programmiersprache **C#** und die Beschreibungssprache "Extensible **A**pplikation **M**arkup **L**anguage" (**XAML**) in der Entwicklungsumgebung "Visual Studio 2022".

Das Projekt soll als eine Einführung für simple Grundlagen von **C#** und **XAML** dienen. Wir beschäftigen uns mit **C# Variablen, Methoden, Konstrukten, Anweisungen...** und lernen **XAML Steuerelemente** und **Attribute** kennen.

Über die Programmierung hinweg gibt es mehrere Möglichkeiten, um bestimmte Funktionalitäten zu implementieren, während von Aufgabentext meistens nur ein Weg erwähnt wird. Leichte Abweichungen sind kein Problem, aber die Entscheidung über die Reihenfolge wurde bestimmt, um einen verständlichen Leitfaden zu erschaffen.

Wenn Fragen bestehen, kannst du in das unterste Kapitel navigieren und dort findest du Hilfe für **C#** und **XAML**. Solltest du auf Schwierigkeiten stoßen, mache dir bitte Notizen darüber und leite diese am Ende des Projekts an uns weiter. Wir

können dementsprechend Änderungen vornehmen und somit das Projekt optimieren.

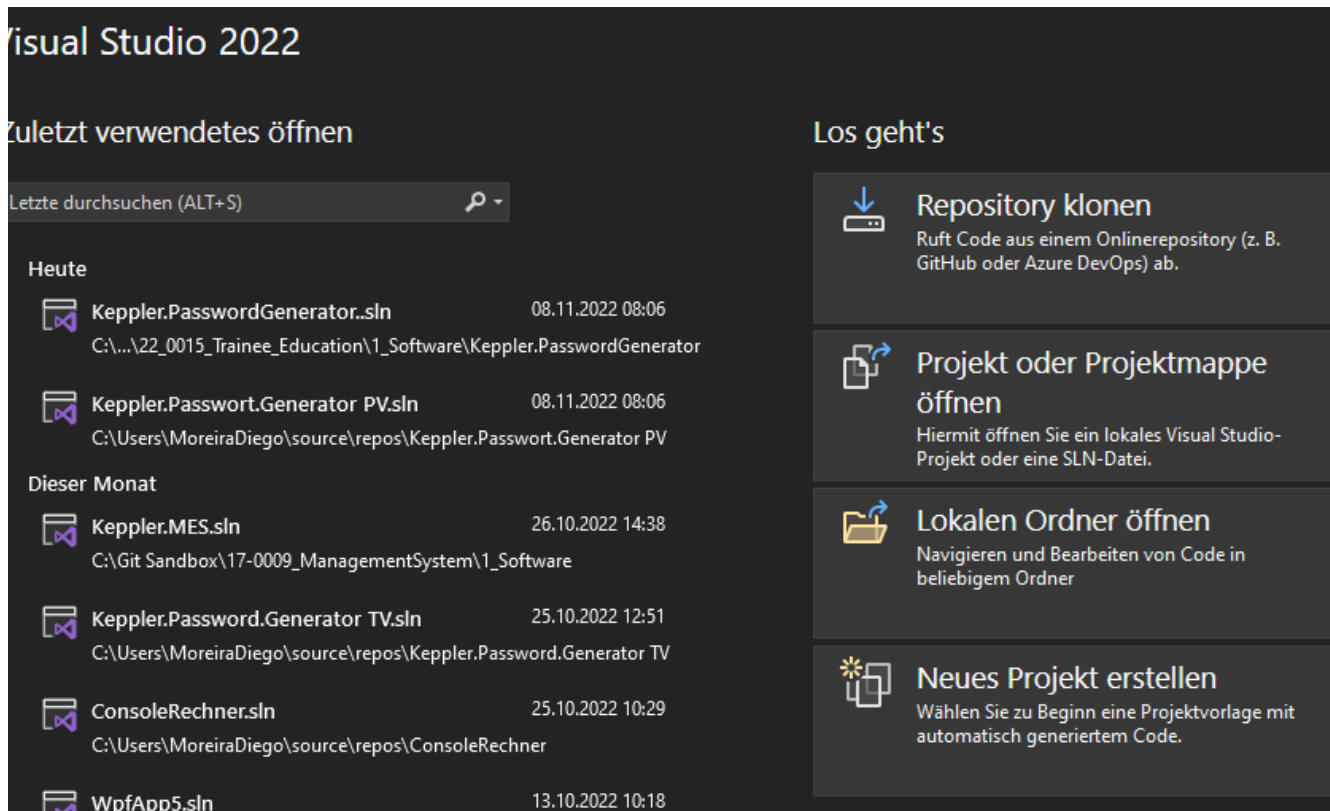


Beispiel für ein mögliches Endresultat

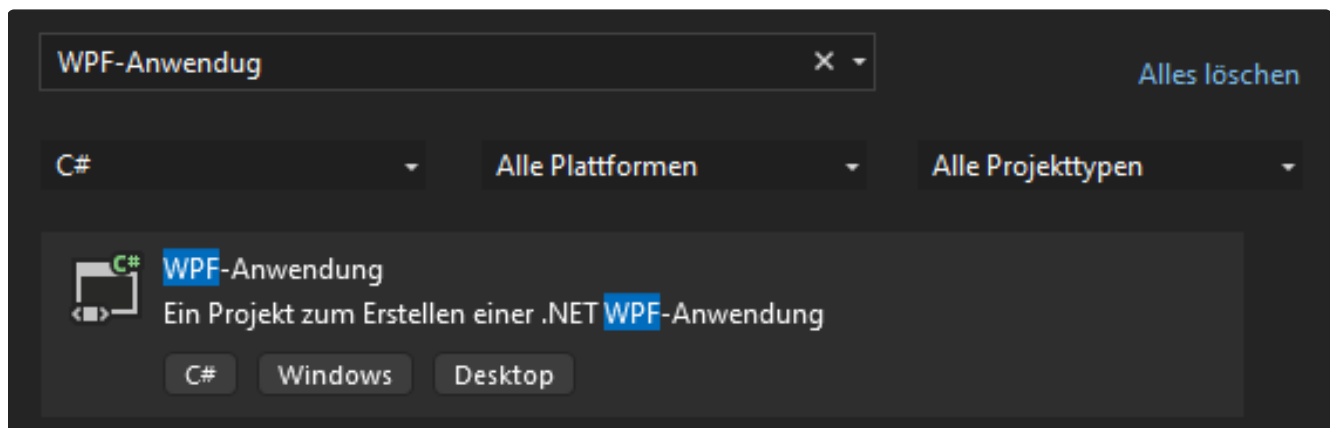
Aufgabe 1: Erstelle ein Projekt



Bevor wir mit dem Erstellen unseres Projektes beginnen, müssen wir zuerst ein neues Projekt anlegen. Dafür öffnen wir **Visual Studio** und drücken auf "Neues Projekt erstellen".



Dann suchen wir nach **WPF-Anwendung** und wählen **“WPF-Anwendung”** aus.



Danach benennen wir unser Projekt und geben einen Pfad an, auf welchem das Projekt liegt.

Neues Projekt konfigurieren

WPF-Anwendung C# Windows Desktop

Projektname
Keppler.Passwort.Generator

Ort
C:\Users\MoreiraDiego\source\repos

Name der Projektmappe ⓘ
Keppler.Passwort.Generator

☒ Platzieren Sie die Projektmappe und das Projekt im selben Verzeichnis.

Am Ende wählen wir noch unser **Framework** (Programmiergerüst) aus. Wir nutzen hierfür **".NET 6.0"**.

WPF-Anwendung C# Windows Desktop

Framework ⓘ
.NET 6.0 (Langfristige Unterstützung)

Aufgabe 2: Unsere Oberfläche



Zu Entwicklungsbeginn unseres Programms kümmern wir uns um unsere grafische Benutzeroberfläche (**GUI, Graphical User Interface**). Die **GUI** ist unsere sichtbare Oberfläche, mit welcher der Nutzer später interagieren wird.

Unsere **XAML GUI** besteht aus zwei Teilen. Erstens dem **XAML** Code und zweitens dem Entwurf, welcher sich aus dem **XAML** Code entwirft.

Eine **XAML GUI** ist aufgebaut mit verschiedenen Arten von **Steuerelementen**. Bei uns beginnt die **GUI** mit einem **Window** (Fenster).

Dieses ist das **MainWindow**. Das bedeutet, dass wenn wir unser Programm starten, dieses Fenster als erstes aufgerufen wird. In diesem Fenster finden wir ein **Grid** (Gitter).

Grids können wir in **Columns** (Spalten) und **Rows** (Reihen) aufteilen. Für diese **Columns** und **Rows** können wir eine bestimmte Breite und Höhe bestimmen. In dieses **Grid** können wir dann auch wieder neue Steuerelemente, wie z.B. ein weiteres **Grid** implementieren.

Steuerelemente die wir benötigen:

Grid:

Beispiel:

```
1 <Window>
2     <Grid>
3         <Grid.RowDefinitions>
4             <RowDefinition Height="Auto" />
5             <RowDefinition Height="*" />
6         </Grid.RowDefinitions>
7         <Grid.ColumnDefinitions>
8             <ColumnDefinition Width="*" />
9             <ColumnDefinition Width="2*" />
10        </Grid.ColumnDefinitions>
11
12        <!-- Hier kannst du weitere XAML-Elemente platzieren -->
13    </Grid>
14 </Window>
```

TextBox:

Das **TextBox**-Element stellt eine Eingabemöglichkeit für Text dar. Benutzer können Text in das Textfeld eingeben oder es kann auch programmatisch bearbeitet werden.

Beispiel:

```
1 <TextBox Text="Hallo, Welt!"/>
```

Label:

Das **Label**-Element wird verwendet, um statischen Text oder Beschriftungen anzuzeigen. Es hat keinen direkten Interaktionsfokus und wird hauptsächlich zum Anzeigen von Informationen verwendet.

Beispiel:

```
1 <Label Content="Hello"/>
```

Button:

Das **Button**-Element repräsentiert eine Schaltfläche, die Benutzer klicken können, um eine Aktion auszuführen. Du kannst Text oder ein Bild als Inhalt eines Buttons verwenden.

Beispiel:

```
1 <Button Content="Klick mich!"/>
```

CheckBox:

Das **CheckBox**-Element stellt eine Option dar, die Benutzer aktivieren oder deaktivieren können. Es wird oft für Zustimmungsabfragen oder Einstellungen verwendet.

Beispiel:

```
1 <CheckBox Content="Klick mich"/>
```

Image:

Das **Image**-Element ermöglicht das Anzeigen von Bildern in deiner Benutzeroberfläche. Du kannst die Quelle des Bildes angeben, um es darzustellen.

Beispiel:

```
1 <Image Source="pfad/zum/bild.png"/>
```

Um die Steuerelemente in unserem **Grid** richtig zu platzieren, haben wir verschiedene **Attribute**, die wir dafür nutzen können.

Wir haben **Grid.Column** und **Grid.Row**. Damit können wir entscheiden in welcher **Spalte** und welcher **Reihe** unser **Element** platziert wird.

ACHTUNG! Beachte, dass wenn wir bei unserem **Element** z.B. die **Spalte** 4 und **Reihe** 1 angeben, unser Element in der 5. **Spalte** und 2. **Reihe** landen wird.

Beispiel:

```
1 <Grid>
2   <Grid.ColumnDefinitions>
3     <ColumnDefinition/>
4     <ColumnDefinition/>
5     <ColumnDefinition/>
6     <ColumnDefinition/>
7     <ColumnDefinition/> <!-- Diese Spalte landet Element
8   </Grid.ColumnDefinitions>
9   <Grid.RowDefinitions>
10    <RowDefintion/>
11    <RowDefintion/> <!-- Diese Reihe landet Element
12    <RowDefintion/>
13  </Grid.RowDefinitions>
14
15    <Element Grid.Column="4" Grid.Row="1"/>
16 </Grid>
```

Dann gibt es auch noch den **Grid.ColumnSpan** und **Grid.RowSpan**.

Über dieses **Attribut** können wir bestimmen, über wie viele **Spalten/Reihen** sich das Element ziehen soll.

Beispiel:

```
1 <Grid>
2   <Grid.ColumnDefinitions>
```

```

3      <ColumnDefinition Width="10"/>
4      <ColumnDefinition Width="20"/> <---
5      <ColumnDefinition Width="30"/> <--- Streckt sich über 3 Spalten
6      <ColumnDefinition Width="40"/> <---
7      <ColumnDefinition Width="40"/>
8  </Grid.ColumnDefinitions>
9  <Grid.RowDefinitions>
10     <RowDefintion/>
11     <RowDefintion/> <--- Streckt sich über 1 Reihe
12     <RowDefintion/>
13 </Grid.RowDefinitions>
14
15 <Element Grid.Column="1" Grid.Row="2" Grid.ColumnSpan="3"
16         Grid.RowSpan="1"/>
17 </Grid>

```

Ebenfalls gibt es noch die **Attribute HorizontalAlignment (Horizontale Ausrichtung)** und **VerticalAlignment (Vertikale Ausrichtung)**.

Mit diesen beiden Attributen können wir die Ausrichtung auf der horizontalen und vertikalen Achse in unseren Spalten und Reihen bestimmen.

Für unser **HorizontalAlignment** haben wir die Möglichkeiten **Center, Left, Right, Stretch**.

Und für unser **VerticalAlignment** haben wir **Center, Top, Bottom** und **Stretch**.

Beispiel:

```

1 <Grid>
2   <Grid.ColumnDefinitions>
3     <ColumnDefinition Width="10"/>
4     <ColumnDefinition Width="20"/> <---
5     <ColumnDefinition Width="30"/> <--- Befindet sich im Mittelpunkt der 3 Spalten
6     <ColumnDefinition Width="40"/> <---
7     <ColumnDefinition Width="40"/>
8   </Grid.ColumnDefinitions>
9   <Grid.RowDefinitions>
10    <RowDefintion/>
11    <RowDefintion/> <--- Befindet sich unten in der Reihe
12    <RowDefintion/>
13  </Grid.RowDefinitions>
14
15  <Element Grid.Column="1" Grid.Row="2" Grid.ColumnSpan="3"
16          Grid.RowSpan="1" HorizontalAlignment="Left"
17          VerticalAlignment="Bottom"/>
18 </Grid>

```

Und dann haben wir noch den **Margin**.

Mit dem **Margin** können wir den Abstand um ein **Steuerelement** festlegen.

Diese Abstände definieren den Raum zwischen dem Rand des **Steuerelements** und seinen umgebenden Elementen.

Der **Margin** wird im Normalfall aus 4 Werten festgelegt.

Diese definieren den Abstand in den Richtungen links, oben, rechts und unten (In dieser Reihenfolge).

Beispiel:

```
1 <Element Margin="2,4,8,16"/> <!-- Links = 2, Oben = 4, Rechts = 8 und Unten = 16-->
```

Neben der Positionierung der Steuerelemente können wir auch die Größe eines Elements anpassen. Dafür nutzen wir die Attribute **Height** und **Width**.

Beispiel:

```
1 <Grid Height="25" Width="50"/>
```

Und für die Schrift haben wir mehrere **Font-** Attribute:

FontSize, welche die Größe der Schrift bestimmt.

Beispiel:

```
1 <TextBox FontSize="25"/>
```

FontFamily, welche die Schriftart bestimmt, z.B. "Arial" und "Verdana".

Beispiel:

```
1 <TextBox FontFamily="Algerian"/>
```

FontStretch, welche die Streckung der Schrift bestimmt, z.B. "SemiCondensed" und "UltraExpanded".

Beispiel:

```
1 <TextBox FontStretch="Condensed"/>
```

FontStyle, welche den Stil der Schrift bestimmt, z.B. "Normal" und "Oblique".

Beispiel:

```
1 <TextBox FontStyle="Italic"/>
```

FontWeight, welche die Schriftstärke bestimmt, z.B. "Bold" und "UltraLight".

Beispiel:

```
1 <TextBox FontWeight="Normal"/>
```

Und für die **Kolorierung** unserer Elemente gibt es ebenfalls mehrere **Attribute**.

Diese sind aber eher elementspezifisch im Gegensatz zu den anderen Attributen, die wir bisher besprochen. Für das Definieren einer Farbe gibt es ebenfalls mehrere Möglichkeiten.

Farbnamen:

Von XAML gibt es vordefinierte bekannte Farben:

Beispiel:

```
1 <TextBox Text="Hallo" Foreground="Red"/> <--- Farbe Red wird genutzt
```

Hexadezimalwerte:

Man kann Farben auch mithilfe von **Hexadezimalwerten** erzeugen.

Der **Hexadezimalwert** besteht aus sechs Zeichen, die die Intensitäten der Farben Rot, Grün und Blau repräsentieren.

Beispiel:

```
1 <Button Foreground="#719AB2"/> <!-- Intensität: Rot=71, Grün=9A, Blau=B2 -->
```

Die Farbnamen sind zwar sehr viel simpler, decken dafür aber nur einen kleinen Teil der möglichen Farben ab. Es gibt noch weitaus mehr Möglichkeiten für die Kolorierung der Steuerelemente, wir benötigen aber erstmal Hexadezimalwerte und Farbnamen.

Es gibt 3 Attribute die wir für das Kolorieren unserer **Steuerelement** benutzen werden.

Foreground:

Dieses Attribut legt die Vordergrundfarbe eines **Steuerelements** fest; normalerweise für einen Textinhalt.

Beispiel:

```
1 <Label Content="Hello" Foreground="#1657E6"/> <--- Schrift "Hello" färbt sich zur Farbe #1657E6
```

Background:

Dieses Attribut legt die Hintergrundfarbe eines **Steuerelementes** fest.

Beispiel:

```
1 <Button Background="SnowWhite"/> <--- Hintergrund vom Button färbt sich zur Farbe "SnowWhite"
```

BorderBrush:

Dieses Attribut legt die Farbe des Randes eines **Steuerelements** fest.

Beispiel:

```
1 <Button BorderBrush="Black"/> <!-- Rand färbt sich zur Farbe "Black"
```

Es gibt noch viele weitere Attribute für das Kolorieren von unseren **Steuerelementen**, aber diese 3 sind jetzt erstmal die Wichtigsten.

Nun brauchen wir jetzt noch die **Text** und **Content (Inhalt)** Attribute.

Text:

Benutzt für **TextBox**, **TextBlock** ...

Beispiel:

```
1 <TextBox Text="Hallo, ich bin eine TextBox"/>
```

Content: Benutzt für **Label**, **Button**, **CheckBox** ...

Beispiel:

```
1 <Label Content="Hallo, ich bin ein Label"/>
2 <Button Content="Hallo, ich bin ein Button"/>
3 <CheckBox Content="Hallo, ich bin eine CheckBox"/>
```

Und das letzte was wir brauchen, ist das **Name** Attribut. Mit diesem kann man die **Steuerelemente** durch ihren Namen differenzieren und später auf diese zugreifen.

```
1 <TextBox Name="txt_Name" Text="Name"/>
2
3 <Label x:Name="lbl_Nachname" Content="Nachname"/>
```

Das wären jetzt erst mal alle Attribute, die wir benötigen. Die Anzahl an existierenden Attributen ist um ein vielfaches höher.

Folgende Steuerelemente benötigen wir nun:

- **Checkboxes** * 4 die wir "cb_CaseOne" - "cb_CaseFour" nennen.
- **TextBoxen** * 2 die wir "txt_GeneratedPasswort" und "txt_Amount" nennen.
- **Labels** * 2 die wir lbl_Anzahl und lbl_ErrorMessage nennen.
- **Buttons** * 2 die wir "btn_Generate" und "btn_Reset" nennen.
- **Image** * 1 das wir "img_Logo" nennen.

Zugang zu unseren Steuerelementen bekommen wir zum einen durch den **XAML** Code, und zum anderen auch aus der **Toolbox**, welche man mit Strg+W+X öffnen kann. Wir nutzen unseren **XAML** Code für unsere Steuerelemente.

Für unser **Image** müssen wir erstmal das Keppler Logo als eine **png (Portable Network Graphics)** abspeichern.

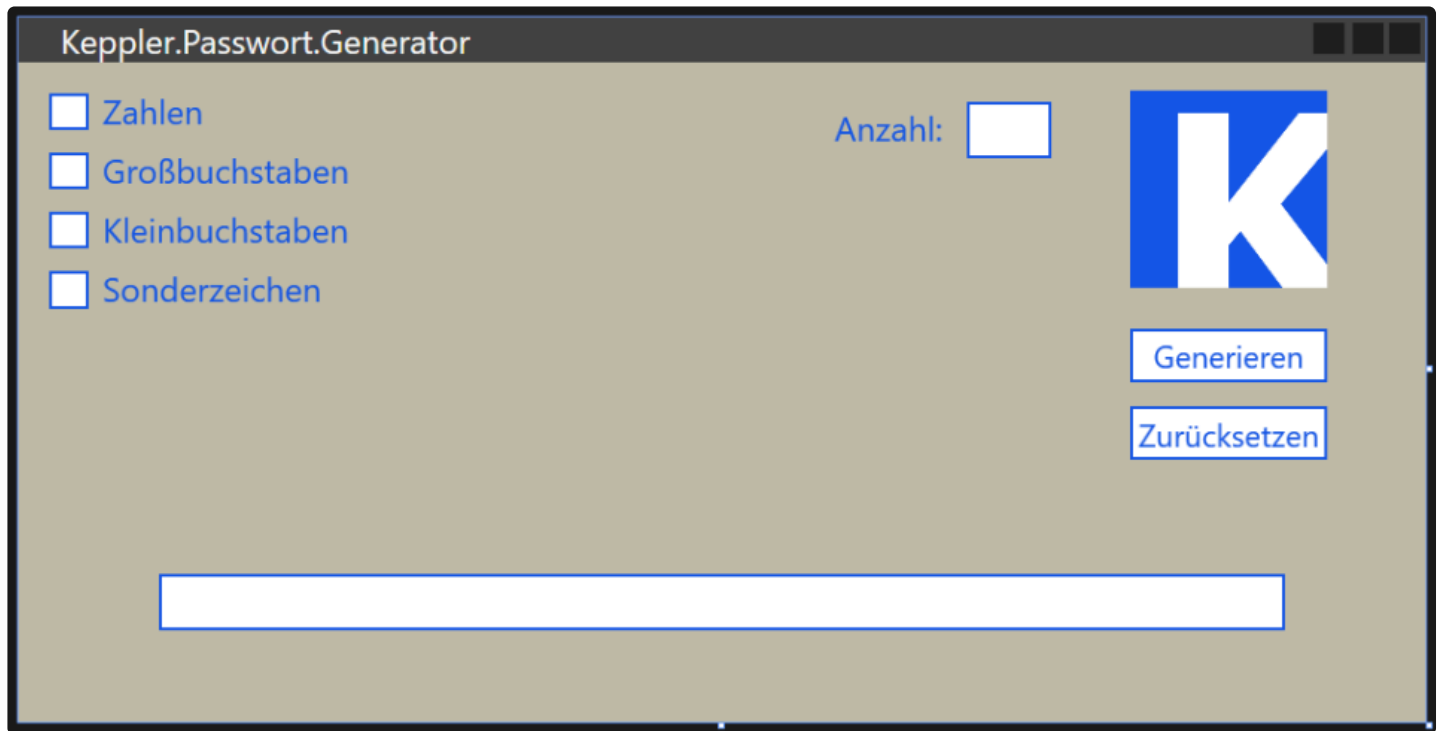
Dann müssen wir in unserem **Projektmappen Explorer**, welchen wir mit Strg+W+S öffnen unser Bild einfügen.

Wenn wir dann die Eigenschaften des Bildes öffnen, wechseln wir den **Buildvorgang** des Bildes auf "Ressource".,

In unserem Image können wir daraufhin das **Attribut** Source nutzen, und in diesem den Pfad des Bildes angeben.

Jetzt können wir den Rest unserer **GUI** mit unseren **Steuerelementen** und deren Attributen sauber anordnen.

Unser **Window** sollte am Ende im Groben so aussehen:



Der **Hex-Code** für das Keppler.Blau ist **#1657E6**.

Aufgabe 3: Beginn des Backend Codes



Nachdem wir uns um die **XAML Code (Frontend)** gekümmert haben, müssen wir uns jetzt um den **C# Code (Backend)** kümmern.

Zuerst wollen wir **Ereignisse** für unsere **CheckBoxen** erzeugen. Dafür gehen wir nochmal kurz in den **XAML Code** und nutzen das Attribut **Checked** in unserer ersten **CheckBox** und schreiben einen **neuen Ereignishandler**:

```
1 <CheckBox Checked="<Neuer Ereignishandler>"/>
```

Das Gleiche machen wir für alle 4 Checkboxes. Schauen wir nun in unseren **Backend Code**, können wir sehen, dass vier neue sogenannte **Methoden** erstellt wurden.

```
1 private void cbCaseOne_Checked(object sender, RoutedEventArgs e)
2 {
```

```
3  
4 }
```

Methoden dienen dazu, Code zu organisieren, strukturieren und wiederzuverwenden. Methoden können, sobald sie erstellt sind, immer wieder aufgerufen werden. Es gibt verschiedene Typen und weitere Komponenten von Methoden. Ein paar davon werden wir auch im Laufe der Entwicklung noch kennenlernen.

In diesen **Checked Methoden** können wir entscheiden, was passieren soll, wenn wir die Checkboxen ankreuzen.

In unserem Fall möchten wir ein Objekt erstellen, mit welchem wir die Information weitergeben, ob eine, mehrere oder keine Checkboxen angekreuzt sind. Hierfür nutzen wir eine Variable vom Typ **bool**.

Der Datentyp **bool** steht für **boolean (Boolescher Wert)**. Er wird verwendet, um Wahrheitswerte darzustellen, also Zustände wie wahr oder falsch.

Beispiel:

```
1 bool wahr = true;  
2 bool falsch = false;
```

Ein **bool** kann einen von 2 Werten haben. **True** und **false**. Wir deklarieren die **bool** Variablen nicht als einfache **bools**, sondern wir nutzen hier den **Zugriffsmodifikator public**. (Für alle Variablen, die wir im Laufe des Projekts deklarieren werden, nutzen wir den **Zugriffsmodifikator public**, außer wenn der **Zugriffsmodifikator** explizit erwähnt wird)

Unsere **bools** deklarieren wir mit den Namen "_numberTrueOrFalse", "_letterBigTrueOrFalse" "_letterSmallTrueOrFalse" und "_specialCharacterTrueOrFalse".

Ebenfalls muss beachtet werden, dass wir die **public Variablen** nicht in den Methoden deklarieren, sondern außerhalb. Optimalerweise deklarieren wir sie alle am Anfang der Klasse für eine bessere Übersichtlichkeit. Solche Variablen nennt man auch globale Felder.

Beispiel:

```
1 public bool _numberTrueOrFalse;
```

Nun müssen wir unseren deklarierten Variablen in den **Checked** Methoden den Wert **true** geben:

```
1 private void cbCaseOne_Checked(object sender, RoutedEventArgs e)
2 {
3     _isNumberTrueFalse = true;
4 }
5
6 private void cbCaseOne_Unchecked(object sender, RoutedEventArgs e)
7 {
8     _isNumberTrueFalse = false;
9 }
```

Was wir jetzt für die **Checked** Ereignisse gemacht haben, müssen wir nun auch nochmal für die **Unchecked** Ereignisses machen. Der einzige Unterschied besteht darin, dass wir unseren **bools** den Wert **false** anstatt **true** geben.

Im nächsten Schritt wollen wir eine Methode erstellen, welche uns am Ende der Methode ein Zeichen für unser Passwort zurück geben soll. Für diese Methode nehmen wir uns die sogenannte **ASCII Tabelle** zur Hand.

Die **ASCII Tabelle (American Standard Code for Information Interchange)** besteht aus 256 Reihen. Diese sind durchnummeriert von 0 bis 255. Jeder Zahl von 0 bis 255 ist ein Zeichen zugewiesen. Beispiel: **33=!** oder **65=A**.

Nun deklarieren wir auch 2 neue Variablen. Eine Variable vom Typ **int** die wir "_number" nennen und eine Variable vom Typ **string** die wir "_character" nennen.

Der Datentyp **int** steht für **integer** (Ganzzahl). Er wird verwendet, um ganze Zahlen ohne Dezimalstellen darzustellen. Der Zahlenbereich von einem **int** ist -2,147,483,648 bis 2,147,483,647. Das sind 32 Bit/4 Bytes.

Beispiel:

```
1 int zahl = 4; // zahl wird mit dem Wert 4 initialisiert
```


Mathematische Operatoren können auf **int**-Variablen angewendet werden.

Der Datentyp **string** steht für Zeichenketten. Er wird verwendet, um Text oder eine Sequenz von Zeichen darzustellen.

Beispiel:

```
1 string name = "Keppler"; // zahl wird mit dem Wert Keppler initialisiert
```

Für einen **string** kann man jede Art von Zeichen benutzen.

Die **int** Variable benötigen wir später, da in dieser Variable der Zahlenwert aus der **ASCII Tabelle** gespeichert sein soll. Die **string** Variable nutzen wir für das Abspeichern des Zeichens aus der **ASCII Tabelle**.

Aufgabe 4: Passwortgenerator



Jetzt kümmern wir uns um den Zufallsgenerator selbst.

Den Aufbau des Generators werden wir in 3 Teile aufteilen.

1. Wir benötigen die vorher angesprochene neue **Methode**. In dieser Methode wollen wir nach einem numerischen Wert aus der **ASCII Tabelle** fragen, damit uns dann das zugewiesene Zeichen aus der Tabelle zurückgegeben wird.
2. Durch unsere 4 **Checkboxen** haben wir 16 verschiedene Möglichkeiten für das zufällige Generieren eines Passworts. Für jede Möglichkeit wird ein Zahlenbereich bestimmt, aus dem dann eine Zahl zufällig generiert wird.

Die Zahlenbereiche entnehmen wir aus der **ASCII Tabelle siehe Kapitel 4.1**.

Da wir nicht nur einstellige Passwörter wollen, müssen wir auch die gewünschte Länge des Passwortes angeben. Hierfür verwenden wir die Anzahl **TextBox**. Die dort eingetragene Zahl, lesen wir aus konvertieren sie von string nach int und

übernehmen sie in eine Variable. Die eben gefüllte Variable verwenden wir in einer Schleife, welche dann bestimmt, wie oft ein Zeichen zufällig generiert werden soll.

Diese zufällig generierte Zeichenkette soll dann in die unten angelegte **TextBox** geschrieben werden.

3. Zuletzt kümmern wir uns noch um Fehlermeldungen und das Zurücksetzen der Einstellungen, Auftreten von Fehlern und Drücken des zweiten Buttons.

4.1:



Für diesen Schritt benötigen wir erstmal eine neue Methode. Dieser Methode geben wir den Namen "GeneratingPasswordCharacterASCII".

Im Gegensatz zu den vorigen **Checked** und **Unchecked** Methoden, nutzen wir jetzt nicht den Typ **void** sondern **string**.

Bei dieser Art von Methode müssen wir am Ende der Methode einen **Rückgabewert** vom Typ **string** angeben. Das macht man, indem man am Ende **return** und entweder einen direkten Wert oder eine Variable schreibt.

Als **return** Wert nutzen wir unsere **_character** Variable.

Beispiel:

```
1 public int Methode()  
2 {  
3  
4     return 1;  
5 }
```

Eine weitere Besonderheit, die wir bei dieser **Methode** beachten müssen, ist, dass wir sogenannte **Übergabeparameter** benutzen wollen.

Um **Übergabeparameter** zu nutzen, muss in der runden Klammer ein Typ, Klasse, ... angegeben werden und zusätzlich ein Name für den Parameter.

Beispiel:

```
1 private void Methodenname(string name)
2 {
3
4 }
```

Übergabeparameter werden benutzt, um beim Aufrufen der Methode dieser einen Wert zu geben, welcher dann in der Methode benutzt werden kann. Einen Übergabeparameter kann man z.B. wie eine Variable aufrufen.

Für unsere Methode deklarieren wir als **Übergabeparameter** eine **int Variable** und nennen diesen "**number**".

Unsere jetzt noch leere Methode sollte dann so aussehen:

```
1 public string GeneratingPasswordCharacterASCII(int number)
2 {
3     return _character;
4 }
```

Unsere Methode müssen wir nun mit der sogenannten **if/else** Bedingungsanweisung füllen.

Die **if**-Anweisung ermöglicht es dir, Code auszuführen, wenn eine bestimmte Bedingung erfüllt ist.

Beispiel:

```
1 if (Bedingung)
2 {
3     // Code, der ausgeführt wird, wenn die Bedingung wahr ist
4 }
5
```

Der Code innerhalb der geschweiften Klammern wird nur ausgeführt, wenn die angegebene Bedingung wahr ist.

Die **else**-Anweisung wird in Verbindung mit der **if**-Anweisung verwendet, um Code auszuführen, wenn die Bedingung der **if**-Anweisung nicht erfüllt ist.

Beispiel:

```
1 if (Bedingung)
2 {
3     // Code, der ausgeführt wird, wenn die Bedingung wahr ist
4 }
5 else
6 {
7     // Code, der ausgeführt wird, wenn die Bedingung nicht wahr ist
8 }
9
```

Der Code innerhalb der ersten geschweiften Klammer wird ausgeführt, wenn die Bedingung der **if**-Anweisung wahr ist, andernfalls wird der Code innerhalb der zweiten geschweiften Klammer ausgeführt.

Die **else if**-Anweisung wird verwendet, wenn du mehrere Bedingungen überprüfen möchtest. Sie wird zwischen einer **if**-Anweisung und einer **else**-Anweisung platziert.

Beispiel:

```
1 if (Bedingung1)
2 {
3     // Code, der ausgeführt wird, wenn Bedingung1 wahr ist
4 }
5 else if (Bedingung2)
6 {
7     // Code, der ausgeführt wird, wenn Bedingung2 wahr ist
8 }
9 else
10 {
11     // Code, der ausgeführt wird, wenn keine der Bedingungen wahr ist
12 }
13
```

Die Bedingungen werden nacheinander überprüft. Wenn Bedingung 1 wahr ist, wird der dazugehörige Code ausgeführt. Wenn nicht, wird Bedingung 2 überprüft, und wenn sie wahr ist, wird der zugehörige Code ausgeführt. Ist keine der Bedingungen wahr, wird der Code im **else**-Block ausgeführt.

Für die **if**-Anweisung die wir jetzt benötigen, wird die zufällig generierte Zahl mit den Zahlenwerten aus der **ASCII Tabelle** verglichen. Sobald eine Bedingung erfüllt ist, wird der Wert der **ASCII Tabelle** in eine Variable geschrieben.

[illegible]

2/2 von ASCII Tabelle

4.2:



Als nächstes müssen wir erstmal eine neue Funktion erstellen, die einen **string** zurückgibt. Wir geben den neuen Funktion Namen "GeneratePassword" geben. Als **Rückgabewert** nutzen wir für diese Methode wieder die "_character" Variable.

```
1 public string GeneratePassword()
2 {
3
4     return _character;
5 }
```

Für unsere Methode brauchen wir jetzt einen **switch-case**.

Das **switch-case**-Konstrukt wird verwendet, um eine bestimmte Aktion abhängig von einem Wert auszuführen. Es ist eine alternative Möglichkeit, bedingte Verzweigungen zu steuern, wenn du mehrere Optionen für den Wert hast.

Wir haben einen **switch** und einen oder mehrere **cases**. Beim **switch** geben wir unser Datenobjekt an und wenn der Wert unseres **cases** mit dem Wert des Datenobjektes übereinstimmt wird dieser **bestimmte case** ausgeführt.

Neben unseren **cases** mit bestimmten Werten können wir zusätzlich auch einen **default case** erstellen, welcher eintritt, wenn keiner der bestimmten Werte mit dem Datenobjekt übereinstimmt.

Einen **break** benötigen wir am Ende jedes **cases**. Nach dem **break Statement** verlassen wir diesen **case** und fahren mit dem restlichen Code fort.

Beispiel:

```
1 switch (Wert)
2 {
3     case Wert1:
4         // Code, der ausgeführt wird, wenn Wert == Wert1
5         break;
6     case Wert2:
7         // Code, der ausgeführt wird, wenn Wert == Wert2
8         break;
9     // ...
10    default:
11        // Code, der ausgeführt wird, wenn keiner der vorherigen Fälle zutrifft
12        break;
13 }
```

Für unseren **switch-case** brauchen wir erstmal eine neue **int** Variable die wir "_switch" nennen. Diese Variable nutzen wir als Datenobjekt für den **switch**. Wir benötigen 16 **cases** denen wir die Zahlen von 0-15 zuweisen. Weiter benötigen wir 16 **cases** für die 16 Möglichkeiten fürs Generieren des Passworts.

```
1 public string GeneratePassword()
2 {
3     switch (_switch)
4     {
5         case 1:
6             //Möglichkeit 1
7             break;
8         case 2:
9             //Möglichkeit 2
10            break;
11        case 3:
12            //Möglichkeit 3
13            break;
14            //.....
15    }
16    return _character;
17 }
```

Nachdem wir alle **cases** angelegt haben, lassen wir diese Methode kurz ruhen und kümmern uns um das Festlegen von den Werten für unseren **switch-case**.

Unser Programm soll immer ein Passwort generieren, wenn wir auf unseren Knopf drücken. Dafür benötigen wir eine neue **Ereignis-Methode** für unseren **Button**.

Hierfür gehen wir in unseren **XAML-Code** und erzeugen ein **Click** Event für unseren "btnGenerate".

Die Methode, die durch das Erstellen des Events erzeugt wurde, wird, wie man auch schon vermuten könnte, ausgeführt, wenn wir den zugewiesenen Button anklicken.

Für diese Methode benötigen wir jetzt eine neue **if/else**-Anweisung. Hierfür brauchen wir nur ein **if** und ein **else**.

Unser **if** lassen wir vorerst mal leer und als Bedingung können wir erst mal einen Platzhalter nutzen.

In unserem **else** benötigen wir jedoch eine **Schleife**. Um die Anzahl der Zeichen zu generieren wollen wir diese Schleife nutzen.

Es gibt mehrere Arten von Schleifen, die in verschiedenen Situation besser oder schlechter sind. Beispiele hierfür sind die **while**- oder **foreach**-Schleife.

In unserem Fall nutzen wir die sogenannte **for**-Schleife.

Bevor wir aber die Schleife in unseren Code einbauen, müssen wir erst mal die Anzahl aus der **TextBox** "txtAmount" auslesen.

Dafür deklarieren wir 2 neue Variablen. Einen **string** welchen wir "_anzahlText" nennen und einen **int** den wir "_amount" nennen.

Dann lesen wir den Text aus der **TextBox** aus und geben den Wert an unseren **string**:

```
1 _anzahlText = txtAmount.Text;
```

Anschließend müssen wir den Wert aus unserem **string** in unseren **int** konvertieren:

```
1 _amount = Convert.ToInt32(anzahlText);
```


Nun wollen wir noch die **TextBox** einmal leeren, bevor das Passwort generiert wird. Warum wir dies tun, wird dir später klar werden.

```
1 txtGeneratedPassword.Text = string.Empty;
```

Da wir jetzt unsere Anzahl an Zeichen, die zufällig generiert werden sollen, auslesen können, können wir uns nun der **for**-Schleife widmen.

Die **for**-Schleife wird verwendet, um eine bestimmte Anzahl von Wiederholungen durchzuführen. Sie besteht aus einem **Initialisierungsausdruck**, einem **Bedingungsausdruck** und einem **Iterationsausdruck**. Der Code im Schleifenkörper wird solange ausgeführt, wie die Bedingung wahr ist.

```
1 for (int i = 0; i < 40; i++)
2 {
3     // Code, der wiederholt ausgeführt wird, solange die Bedingung erfüllt ist
4 }
5
6
7 (int i = 0 /*Initialisierungsausdruck*/;
8  i < 40 /*Bedingungsausdruck*/;
9  i++ /*Iterationsausdruck*/)
```

In unserem Schleifenkopf deklarieren wir eine neue, aber auch eine **lokale int** Variable, die wir "index" nennen. Dieser Variablen geben wir den Wert 0 (**Initialisierungsausdruck**).

Die Schleife soll wiederholt werden, solange der Wert in der "_amount" Variable höher als unser eben deklarierter **int** "index" ist (**Bedingungsausdruck**).

Am Ende jeder Wiederholung soll der Wert vom "index" um 1 hochgezählt werden (**Iterationsdruck**).

```
1 for (int index = 0; index < _amount; index++)
2 {
3
4 }
```

Da wir mit unserer Schleife fertig sind, kümmern wir uns nun um den zu wiederholenden Code. Wir benötigen jetzt eine neue **if/else** Anweisung.

In diesem **if/else** wollen wir durch unsere Bedingungen herausfinden, wie der Status der **Checkboxen** ist; also ob sie angekreuzt sind oder nicht (**true** oder **false**).

Abhängend vom Status der Checkboxen, wird der "_switch" Variablen ein Wert von 0-15 gegeben.

Ein **if/else if** sollte dann so aussehen:

```
1 if (_isNumberTrueFalse && _isLetterBigTrueFalse &&
2    !_isLetterSmallTrueFalse && _isSpecialCharacterTrueFalse) //Yes/Yes/No/Yes
3 {
4
5 }
```

Ist aber auch so möglich:

```
1 if (cbCaseOne.IsChecked == true && cbCaseTwo.IsChecked == false &&
2    cbCaseThree.IsChecked == true && cbCaseFour.IsChecked == false) //Yes/No/Yes/No
3 {
4
5 }
```

Nun müssen wir noch den **if's** und **else if's** die Aufgabe zuteilen, dass sie beim Ausführen der _switch Variablen einen Wert von 0-15 geben. Die Verteilung von den Zahlen 0-15 machen wir über das **binäre Zahlensystem**.

Das **binäre Zahlensystem**, auch bekannt als **Dualsystem**, ist ein Zahlensystem, das auf der Verwendung von nur 0 und 1 besteht.

Im **dezimalen Zahlensystem** verwenden wir zehn Ziffern (0 bis 9) und stellen Zahlen durch die Kombination dieser Ziffern dar. Zum Beispiel ist die Zahl "123" im **dezimalen System** die Summe von $1 * 10^2 + 2 * 10^1 + 3 * 10^0$.

Im binären Zahlensystem verwenden wir nur zwei Ziffern: 0 und 1. Jede Stelle einer **Binärzahl** repräsentiert eine **Potenz** von 2. Die äußerste rechts liegende Stelle hat den Wert 2^0 , die nächste Stelle den Wert 2^1 , dann 2^2 und so weiter.

Hier ist ein Beispiel, um dies zu verdeutlichen:

- Die binäre Zahl "10101" im dezimalen System entspricht:

$$1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 16 + 0 + 4 + 0 + 1 = 21.$$

Die Kombinationen von 0 und 1 werden als "**Bits**" (**Binary Digits**) bezeichnet, und 8 **Bits** ergeben ein "**Byte**".

So wollen wir das **binäre Zahlensystem** nun nutzen:

```
1 if (!_isNumberTrueFalse && !_isLetterBigTrueFalse &&
2     !_isLetterSmallTrueFalse && !_isSpecialCharacterTrueFalse
3 ) //No/Yes/Yes/No = 0110 = 6
4 {
5     _switch = 6;
6 }
```

Wir schauen unsere Bedingungen an und prüfen, welche **Checkboxen** angekreuzt sind und welche nicht. Ist eine Checkbox angekreuzt, entspricht dies dem Wert 1 und wenn nicht dem Wert 0.

Im oberen Beispiel ist die **Checkbox** 2 und 3 angekreuzt, 1 und 4 nicht. Somit haben wir 4 **Bits** die dann 0110 ergeben. Das resultiert zur **Dezimalzahl** 6. So gehen wir auch bei den restlichen Möglichkeiten vor.

Am Ende der Schleife schreiben wir auch noch 2 Zeilen in welchen wir zum einen unsere "_generatePassword" Methode aufrufen und zum anderen wird die Textbox "txtGeneratedPassword" mit unseren zufälligen Zeichen gefüllt.

```
1     GeneratePassword();
2     txtGeneratedPassword.Text = txtGeneratedPassword.Text + _character;
3 }
```

Da wir unsere Werte für unsere **cases** haben, können wir uns damit beschäftigen, was in diesen **cases** umgesetzt werden soll.

In jedem **case** müssen wir nun eine oder mehrere Zahlen zufällig generieren, welche wir später in den if/else der GeneratingPasswordCharacterASCII Methode verwenden. Um dies umzusetzen nutzen wir die **Random**-Klasse.

Die **Random**-Klasse wird verwendet, um Zufallszahlen zu generieren.

Um die **Random**-Klasse nutzen zu können, muss erst mal eine **Instanz** von dieser erstellt werden.

```
1 Random random = new Random();
```

Nachdem du dein **Random Objekt** instanziiert hast, bestehen für dich mehrere Wege, wie du eine Zahl generieren kannst.

```
1 int number = random.Next();
2 // Generiert eine postive Ganzzahl
3
4 int number = random.Next(minimalWert, maximalWert);
5 // Generiert eine Ganzzahl zwischen dem minimalWert (einschließlich) und dem maximal Wert
  (ausschließlich), also +1 vom gewollten maximal Wert
6
7 int number = random.NextDouble();
8 // Generiert eine Zufallszahl zwischen 0 und 1 als Gleitkommazahl
```

Wir nutzen die **Next** Methode mit minimal- und maximal Wert.

Nun müssen wir unsere "_number" Variable nutzen. Dazu müssen wir auch ein **public Random** instanziiieren, welches wir "_randomNumber" nennen.

Ab jetzt kommen die abgesteckten Bereiche der **ASCII Tabelle** ins Spiel. **(33-47, 58-64, 91-96, 123-126)**

Nehmen wir mal an, dass der **case** für den Wert 5 ausgeführt werden soll. Das heißt, dass die 5 in **binär** 0101 ist. Dies wiederum bedeutet, dass nur unsere erste und dritte **Checkbox** angekreuzt ist.

Das würde bedeuten, dass das Passwort, welches generiert werden soll, aus Großbuchstaben und Sonderzeichen bestehen soll. Das wären dann die Zahlenbereiche:

- **33 - 47/ +1 = 48**
- **58 - 64/ +1 = 65**
- **65 - 90/ +1 = 91**
- **91 - 96/ +1 = 97**
- **123 - 126/ +1 = 127**

Wenn wir jetzt die aufeinander folgenden Zahlenbereiche in einen Zahlenbereich zusammenführen, sieht es nun so aus:

- **33 - 47/ +1 = 48**
- **58 - 96/ +1 = 97**
- **123 - 126/ +1 = 127**

Nun haben wir drei Bereiche. Das bedeutet, dass wir erst mal eine Zahl von 1-3 generieren müssen:

```
1 case 5:
2     _randomNumber = new Random();
3     _number = _randomNumber.Next(1,4);
4     break;
```

Jetzt müssen wir einen weiteren **if/else** oder einen **switch-case** erstellen. In diesem nutzen wir die zufällig generierten Zahlen und generieren dann in dem auszuführenden Code eine Zahl aus den bevor erwähnten Bereichen.

Nachdem eine Zahl für die "_number" Variable generiert wurde, rufen wir die "GeneratingPasswordCharacterASCII" Methode auf und geben die "_number" Variable als **Übergabewert** an:

```
1 case 5:
2     _randomNumber = new Random();
3     _number = _randomNumber.Next(1,4);
4     if (_number == 1)
5     {
6         _randomNumber = new Random();
7         _number = _randomNumber.Next(33, 48);
8         GeneratingPasswordCharacterASCII(_number);
9     }
10    else if (_number == 2)
11    {
12        _randomNumber = new Random();
13        _number = _randomNumber.Next(91, 97);
14        GeneratingPasswordCharacterASCII(_number);
15    }
16
17    else if (_number == 3)
18    {
19        _randomNumber = new Random();
20        _number = _randomNumber.Next(123, 127);
21        GeneratingPasswordCharacterASCII(_number);
22    }
23    break;
```

So können wir nun mit unseren restlichen **cases** vorgehen. Es gibt jedoch eine Ausnahme und zwar unseren **case 0**, da in diesem keine der **Checkboxen** angekreuzt sind. Darauf gehen wir später ein.



4.3:

Nun gegen Ende wollen wir uns noch mit Kleinigkeiten für unseren Passwort Generator beschäftigen. Zuerst nutzen wir unseren zweiten Button und geben ihm ein Ereignis. Bei diesem Button wollen wir auch eine **Click** Methode erzeugen.

In dieser Methode wollen wir unsere **TextBoxen, Labels** und **Checkboxes**(Ausnahme "lbl_Anzahl") auf den Ursprungswert zurücksetzen.

```
1 private void btnResetClick(object sender, RoutedEventArgs e)
2 {
3     _numberTrueFalse = false; //...
4     cbCaseOne.IsChecked = false; //...
5     txtGeneratedPassword.Text = ""; //...
6 }
```

Als Nächstes wollen wir noch unsere Fehlermeldungen implementieren. Hierbei wollen wir, dass, wenn diese auftreten, dasselbe passiert, als würden wir den "Reset" Button drücken.

Hierfür legen wir eine neue Methode an. Dieses Mal mit dem Rückgabewert bool. Wir nennen sie **_errorMessageCheck**.

Genau wie bei einer **string** Methode müssen wir auch bei einer **bool** Methode am Ende einen **Rückgabewert** angeben. Dazu deklarieren wir noch eine neue **bool** Variable, welche wir "_error" nennen und diese nutzen wir direkt auch als **Rückgabewert**. In dieser Methode erstellen wir jetzt ein **try** und **catch**.

Try und **catch** wird dafür benutzt, um Fehler und Ausnahmen aufzufangen und darauf reagieren zu können, ohne dass das Programm abbricht.

Der **try**-Block enthält den Code, in dem möglicherweise Ausnahmen auftreten könnten. Wenn eine Ausnahme innerhalb des **try**-Blocks auftritt, wird der normale Programmfluss unterbrochen, und die Ausführung wird zum entsprechenden **catch**-Block oder zu einer äußeren Umgebung geleitet, welche einen passenden **catch**-Block hat.

```
1 try
2 {
3     // Code, in dem eine Ausnahme auftreten könnte
4 }
5 catch (ExceptionTyp ex)
```

```
6 {  
7     // Code zur Behandlung der aufgetretenen Ausnahme  
8 }
```

Der **catch**-Block enthält den Code, der ausgeführt wird, wenn eine Ausnahme des angegebenen Typs auftritt. Der angegebene Typ (**ExceptionType**) kann eine spezifische Ausnahme oder eine allgemeine **Exception** sein, die die Basis für alle Ausnahmen darstellt.

```
1 try  
2 {  
3     // Code, in dem eine Ausnahme auftreten könnte  
4 }  
5 catch (number = 4)  
6 {  
7     // Code zur Behandlung beim Auftreten von number = 4  
8 }  
9 catch  
10 {  
11     // Code zur Behandlung anderer Ausnahmen  
12 }
```

Wir möchten **try** und **catch** verwenden, um eine Situation abzufangen, in der keine positive Ganzzahl in der "txtAmount" **TextBox** eingegeben wurde.

Wenn dies geschieht, möchten wir den Wert in einer neuen Variablen, mit dem Datentyp **int** und dem Namen "_error" speichern.

Dadurch können wir später die Fehlermeldung erzeugen, dass eine positive Ganzzahl eingetragen werden muss.

Zuerst müssen wir die Anzahl aus der **TextBox** auslesen und in der "_anzahlText" Variable abspeichern. Dann erstellen wir eine neue lokale Variable aber dieses mal einen neuen Typ.

Wir deklarieren eine **uint** Variable.

Der Datentyp **uint** steht für **unsigned integer** (nicht vorzeichenbehafteter ganzer Wert). **UInt** wird benutzt, um positive Ganzzahlen ohne Vorzeichen darzustellen. Das bedeutet, dass der **uint**-Datentyp keine negativen Werte speichern kann.

Ein **uint** kann Werte von 0 bis 4.294.967.295. Das sind 32 Bit/4 Bytes.

```

1 uint richtig = 02834074; <--- Funktioniert
2
3 uint falsch = -084082; <--- Funktioniert nicht

```

Unsere lokale **uint** Variable nennen wir "try". Wir konvertieren den Wert der "_anzahlText" Variable in unsere "try" Variable.

```

1 try
2 {
3     _anzahlText = txtAmount.Text;
4     uint try = Convert.ToInt(_anzahlText);
5 }

```

Wenn keine gerade positive Zahl angegeben wurde, wird unser **catch** ausgeführt. In diesem geben wir unserer "_error" Variablen den Wert **true**.

```

1 public bool ErrorMessageCheck()
2 {
3     try
4     {
5         _anzahlText = txtAmount.Text;
6         uint try2 = Convert.ToInt32(_anzahlText);
7     }
8     catch
9     {
10        _error = true;
11    }
12    return _error;
13 }

```

Wenn wir diese Methode fertig geschrieben haben, rufen wir sie am Anfang der "btnGenerate_Click" auf. In dem darunter geschriebenen **if/else**, ersetzen wir die vorherige Platzhalterbedingung mit "_error" == **true**.

```

1 private void btnGenerate_Click(object sender, RoutedEventArgs e)
2 {
3     _errorMessageCheck();
4     if (_error == true)
5     {
6
7     }
8     else
9     {
10    //...

```


In den Codeblock des **if-Statements** schreiben wir, dass unser Label eine Fehlermeldung anzeigt wie z.B. "Bitte geben sie eine positive gerade Zahl ein". Ebenfalls wollen wir, dass unsere TextBoxen geleert werden. Hierfür sollen unsere **Checkboxen IsChecked** und die **bool** Werte der **Checkboxen** auf **false** gesetzt werden.

```
1 private void btnGenerate_Click(object sender, RoutedEventArgs e)
2 {
3     _errorMessageCheck();
4     if (_error == true)
5     {
6         lbl_ErrorMessage.Content = "Bitte geben sie eine positive gerade Zahl an";
7         txtGeneratedPassword.Text = ""; //...
8         _numberTrueFalse = false; //...
9         cbCaseOne.IsChecked = false; //...
10    }
11    else
12    {
13        //...
```

Für unsere zweite Fehlermeldung gehen wir jetzt in unseren **case 0**. In diesem implementieren wir die gleichen Zeilen Code, wie bei der ersten Fehlermeldung. Hier ändern wir die Fehlermeldung allerdings selbst ab. Du kannst z.B. folgenden Text hierfür nutzen: "Bitte kreuzen sie mindestens eine der Möglichkeiten an".

```
1 case 0:
2     lbl_ErrorMessage.Content = "Bitte wählen sie eine der Möglichkeiten";
3     txtGeneratedPassword.Text = ""; //...
4     _numberTrueFalse = false; //...
5     cbCaseOne.IsChecked = false; //...
6     break;
```

Die Entwicklung des Keppler.Password.Generators ist abgeschlossen. Sollte das Programm jetzt nicht optimal funktionieren, dann verfolge deine Schritte der Entwicklung und suche nach den Problemen. Solltest du bestehende Probleme eigenständig nicht lösen können, darfst du uns selbstverständlich fragen.

Wenn du jetzt mit allen Aufgaben fertig bist, alles fehlerlos funktioniert und du noch Zeit hast, kannst du mit der Zusatzaufgabe 5 weitermachen.

Zusatzaufgabe 5:



5.1:



5.2:



Zusatzaufgabe 6:



6.1:



6.2:



6.3:



Ideen zur Weiterentwicklung?



Hilfe benötigt?

