



VORBEMERKUNG

Die folgenden Seiten sind in der Projektphase entstanden.

Schülerinnen und Schüler haben Fragen und Schwierigkeiten beim Programmieren ihres Projektes vorgestellt. Dazu habe ich die folgenden Anleitungen speziell für deren Projekt geschrieben, aber gleichzeitig auch als Anregung für die anderen Teams.

Der MIT App Inventor (<http://appinventor.mit.edu>) wurde ursprünglich von einem Entwicklerteam um Mark Friedman und Hal Abelson bei Google entwickelt und 2012 an das MIT übergeben.

Der MIT App Inventor wird unter der Creative Commons Attribution-ShareAlike 3.0 Unported License veröffentlicht: <https://creativecommons.org/licenses/by-sa/3.0>

Scratch wird von der Lifelong-Kindergarten-Group am MIT-Media-Lab entwickelt. Siehe <http://scratch.mit.edu>.
Scratch ist lizenziert unter **CC BY-SA 2.0** (<https://creativecommons.org/licenses/by-sa/2.0/deed.en>).



HILFEKARTEN

Spiel geschafft? - Übergabe zwischen Screens

Benötigte Komponenten:

1. Screen 1: ListView „lvSpielauswahl“
2. Screen 2 („Spiel_1“): Button „btGeschafft“

Auf dem ersten Screen ist eine Auswahlliste für verschiedene Aufträge (bzw. Spiele) der App. Wenn man einen Auftrag erfolgreich erledigt hat, soll das in der Liste sichtbar sein.

Für jeden Auftrag wird ein neuer Screen gestartet, der dann den Wert „true“ (wahr) zurückgeben soll, wenn der Nutzer / die Nutzerin erfolgreich war.

Hier ist nur die Programmierung für einen Auftrag enthalten. Ihr schafft es bestimmt, weitere Aufträge mit einzubinden. Wenn ihr Fragen dazu habt, meldet euch!

Blöcke Screen 1:

Zunächst erstellen wir eine neue Liste, in der die einzelnen Aufträge aufgelistet sind.

```
initialize global listviewelement_liste to [
    make a list [
        " Spiel 1 "
        " Spiel 2 "
        " Spiel 3 "
    ]
]
```

Diese wird bei der Initialisierung des Screens der ListView übergeben.

```
when Screen1 .Initialize
do set lvSpielauswahl . Elements to get global listviewelement_liste
```

Wird ein Eintrag ausgewählt, soll der entsprechende neue Screen geöffnet werden (hier muss für die weiteren Aufträge ergänzt werden: „else if...“):

```
when lvSpielauswahl .AfterPicking
do if lvSpielauswahl . SelectionIndex = 1
then open another screen screenName " Spiel_1 "
```

Jetzt kümmern wir uns zunächst um den weiteren Screen „Spiel_1“.



Blöcke Screen 2:

Wenn der Auftrag erledigt ist, wird der Screen geschlossen und der Wert „true“ übergeben. Hier im Beispiel erfolgt das, wenn auf den Button btGeschafft geklickt wird. Je nach Auftrag kann der Block auch in einem anderen Ereignis eingebunden werden.

```
when btGeschafft .Click
do close screen with value result true
```

Der Screen wird geschlossen und der Nutzer / die Nutzerin sieht wieder den ursprünglichen Screen mit der Auswahlliste.

Blöcke Screen 1:

Das Ereignis, das ausgelöst wurde heißt „when ... OtherScreenClosed“. Es bekommt sowohl den Name des anderen Screens (otherScreenName) als auch den Übergabewert (result) mit übergeben.

```
when Screen1 .OtherScreenClosed
otherScreenName result
do
```

In einer Entscheidung fragen wir ab, welcher Screen geschlossen wurde (also von welchem Auftrag der Nutzer / die Nutzerin zurückkehrt) und ob es erfolgreich war.

```
if
  get otherScreenName = "Spiel_1" and get result = true
then
```

Wenn ja, wird der entsprechende Eintrag in der Auswahlliste ersetzt von z.B. „Spiel 1“ zu „Spiel 1 geschafft!“.

```
replace list item list
index 1
replacement "Spiel 1 geschafft! :) "
```

Natürlich muss am Ende die veränderte Liste wieder an die ListView zur Anzeige übergeben werden. Insgesamt sehen die Blöcke so aus:

```
when Screen1 .OtherScreenClosed
  otherScreenName result
do
  if
    get otherScreenName = "Spiel_1" and get result = true
  then
    replace list item list
      get global listviewelement_liste
      index 1
      replacement "Spiel 1 geschafft! :) "
    set lvSpielauswahl .Elements to get global listviewelement_liste
```

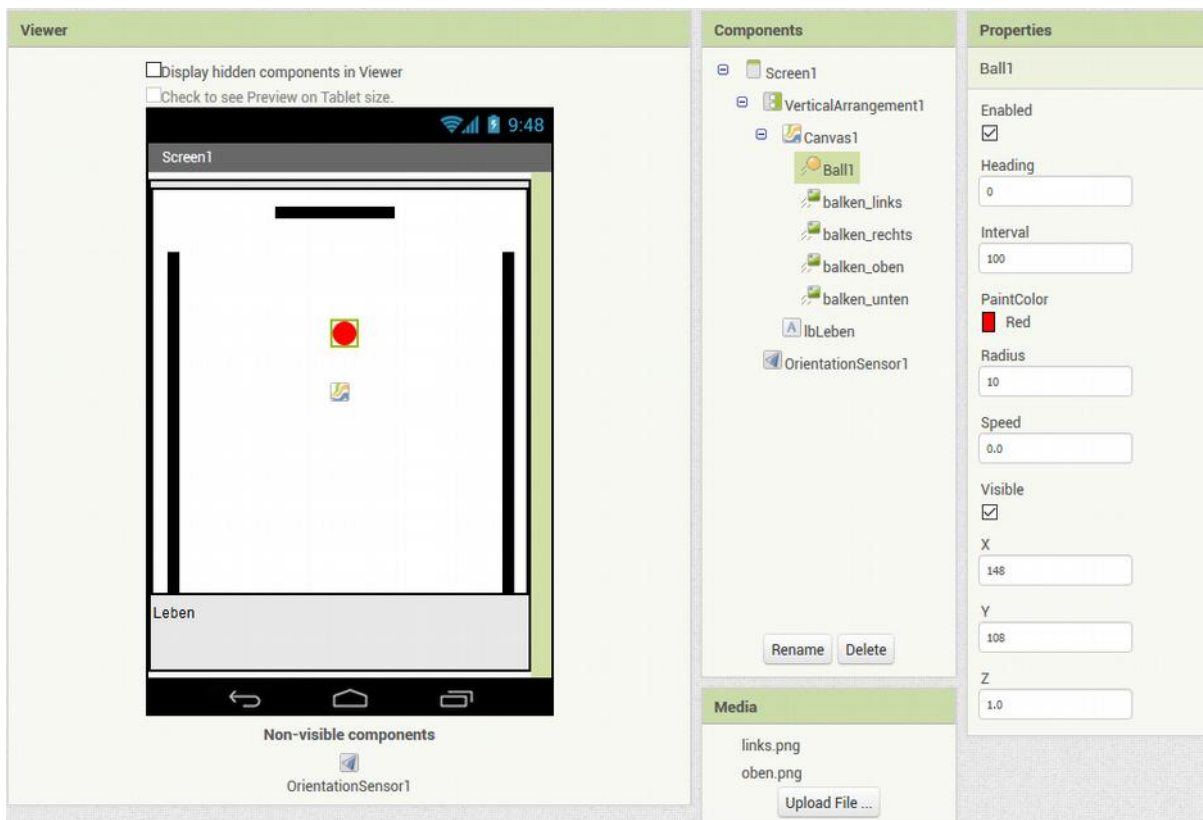


HILFEKARTEN

Nutzen des Orientation Sensors

Benötigte Komponenten:

1. Als Layout ein „VerticalArrangement“.
2. Eine Leinwand (Canvas) mit einem Ball und vier ImageSprites (Balken).
3. Ein Label, um die Anzahl der Leben auszugeben.
4. Einen Orientation Sensor.



Bei der Initialisierung des Screens wird der Ball in die Mitte der Leinwand gesetzt und dem Label die Anzahl der Leben mit übergeben.

```

initialize global leben to 10

when Screen1 Initialize
do
  call Ball1 MoveTo
  x Canvas1 Width / 2
  y Canvas1 Height / 2
  set lbLeben Text to join "Leben: "
  get global leben
    
```



Bewegt sich das Smartphone, wird das Ereignis „when ... OrientationChanged“ ausgelöst. Dabei bekommt man drei Werte übergeben. Probiere aus, was passiert, wenn du folgende Blöcke programmierst:

```

when OrientationSensor1 .OrientationChanged
  azimuth pitch roll
do
  call Ball1 .MoveTo
    x Ball1 . X + get roll
    y Ball1 . Y + get pitch
  
```

Möchte man noch die Kollision des Balles mit den Balken am Rand programmieren, nutzt man das Ereignis „when ... CollidedWith“ des Balles.

Überlege dir zunächst, was in den folgenden Blöcken programmiert wurde und teste es dann aus.

```

when Ball1 .CollidedWith
  other
do
  if get global leben > 0
  then
    set global leben to get global leben - 1
    call Ball1 .MoveTo
      x Canvas1 . Width / 2
      y Canvas1 . Height / 2
  else
    set OrientationSensor1 . Enabled to false
  set lblLeben . Text to join ( "Leben: "
    get global leben
  
```

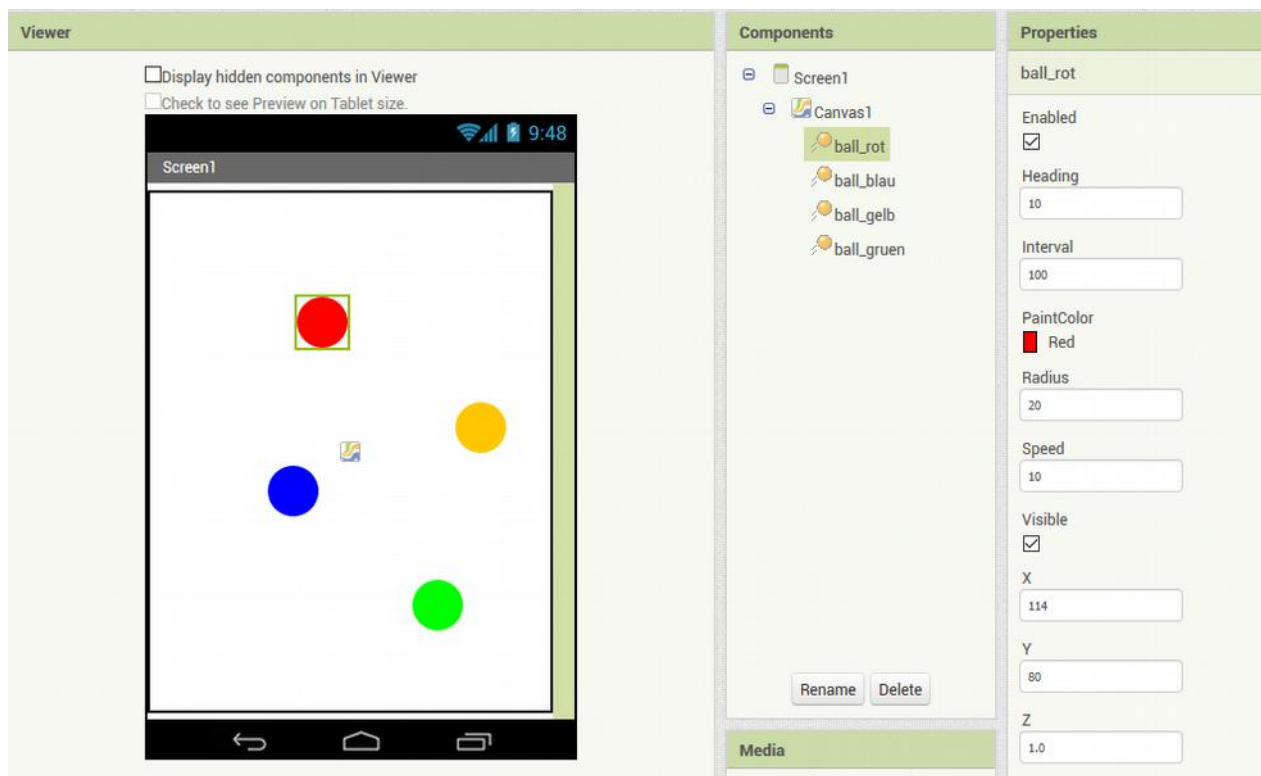


HILFEKARTEN

Kollision von Objekten

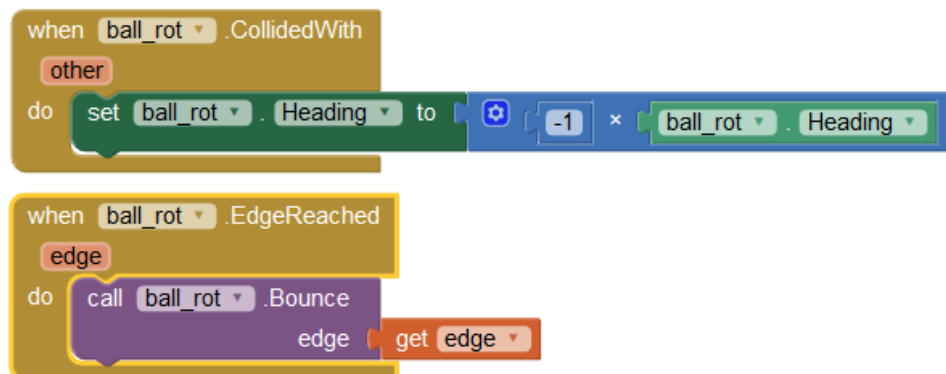
Benötigte Komponenten:

1. Eine Leinwand (Canvas).
2. Vier Bälle mit unterschiedlichen Farben, einer Geschwindigkeit (Speed) von 10 und unterschiedlichen Richtungswinkeln (Heading).



Die Bälle bewegen sich auf dem Screen und sollen voneinander abprallen, d.h. sich beim Zusammenstoß in die Gegenrichtung weiterbewegen. Außerdem sollen sie vom Rand abprallen.

Folgende Blöcke brauchst du für alle vier Bälle:



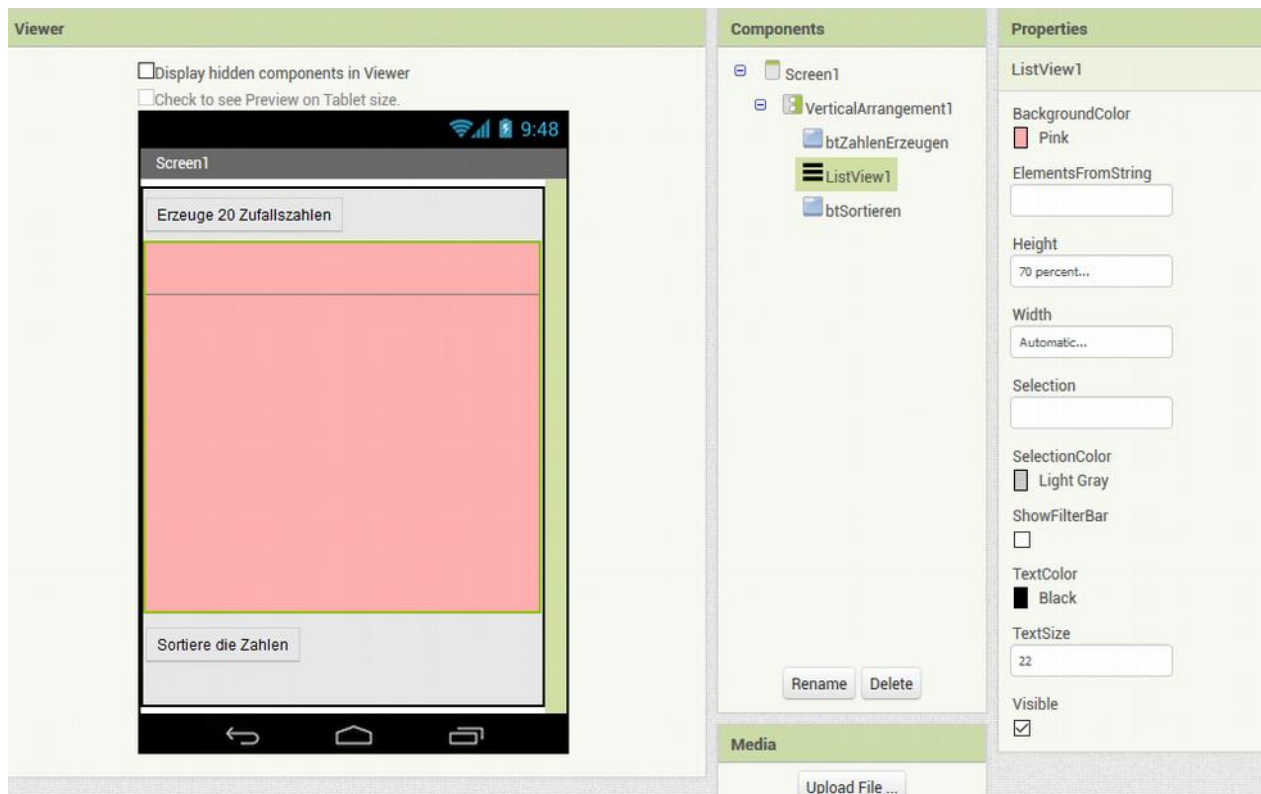


HILFEKARTEN

Sortieren einer Zahlenliste

Benötigte Komponenten:

1. Für das Layout nutzen wir ein VerticalArrangement.
2. Einen Button zum Erzeugen von Zufallszahlen.
3. Eine ListView, in der die Zahlen angezeigt werden.
4. Einen Button zum Sortieren der Zahlen.



Um eine Liste von Zahlen zu sortieren, kann man viele unterschiedliche Ansätze nutzen. Bei einer kleinen Menge kommt es nicht auf das Verfahren an, da selbst zeitaufwendige Verfahren schnell erledigt sind. Deshalb nutzen wir hier jetzt unsere Idee der Maximumsuche.

Wir durchlaufen im ersten Schritt unsere gesamte Liste und suchen nach dem Maximum. Dieses Maximum setzen wir in einer neuen Liste an die erste Stelle und entfernen es aus der ersten Liste. Diese ist nun einen Eintrag kürzer.

Wir suchen erneut nach dem nächsten Maximum, hängen es an die sortierte Liste (die gerade nur ein Element, nämlich das absolute Maximum enthält) an und entfernen es wieder aus der ursprünglichen Liste. Das machen wir so lange, bis die Liste keine Einträge mehr enthält.



Initialisieren der beiden Listen:

```

initialize global zahlenliste to create empty list
initialize global sortierte_liste to create empty list
    
```

Erzeugen der Zufallszahlen und Eintragen dieser Zahlen in die „zahlenliste“:

```

when btZahlenErzeugen .Click
do
  set global zahlenliste to create empty list
  set global sortierte_liste to create empty list
  for each number from 1 to 20 by 1
  do
    add items to list list get global zahlenliste
    item random integer from 1 to 100
  set ListView1 . Elements to get global zahlenliste
    
```

Überlege dir, warum hier die beiden Listen geleert werden. (Müsste beim ersten Mal nicht gemacht werden. Warum später?)

Maximumsuche einer Liste – hier als Prozedur „maximum“:

```

to maximum list
do
  initialize local max_aktuell to 0
  initialize local pos_aktuell to 1
  in
    for each number from 1 to length of list list get list by 1
    do
      if select list item list get list > get max_aktuell index get number
      then
        set max_aktuell to select list item list get list index get number
        set pos_aktuell to get number
      add items to list list get global sortierte_liste
      item get max_aktuell
      remove list item list get global zahlenliste index get pos_aktuell
    
```

Beschreibe, was in der Prozedur passiert. Nutze dazu eine Beispielliste: 3, 6, 17, 2, 21, 8.



Beim Klick auf den Button „btSortieren“ soll nun die Zahlenliste sortiert werden wie oben beschrieben.

Dazu nutzen wir im Click-Ereignis eine lokale Variable „laenge“, der zunächst die Länge der aktuellen Zahlenliste übergeben wird.

Die Schleife wird „laenge“ Mal durchlaufen und in jedem Durchlauf wird die Prozedur „maximum“ mit der immer weiter verkürzten Zahlenliste aufgerufen.

Am Ende werden die Elemente der sortierten Liste an die ListView zur Anzeige übergeben.

```

when btSortieren .Click
do
  initialize local laenge to length of list list get global zahlenliste
  in
    for each startposition from 1
      to get laenge
      by 1
    do
      call maximum
      list get global zahlenliste
  set ListView1 . Elements to get global sortierte_liste
  
```



HILFEKARTEN

Chat-App mit Nutzen von firebase

Benötigte Komponenten:

1. Horizontales Layout mit einem Label (Text: „Dein Name:“) und einer Textbox.
2. Eine größere Textbox, bei der die Eigenschaft MultiLine (mehrere Zeilen) ausgewählt ist.
3. Horizontales Layout mit einem Button (btSend), einer Textbox (für die Nachricht) und einem weiteren Button, um das mittlere Textfeld löschen zu können.

Außerdem brauchen wir im Internet eine Möglichkeit, die aktuelle Nachricht und den aktuell Schreibenden speichern zu können. Dazu nutzen wir eine so genannte Datenbank. Hier von firebase.

Um auf diese Datenbank zugreifen zu können, brauchen wir auf unserem Screen noch eine unsichtbare Komponente aus der Palette „Experimental“: FirebaseDatabase

The screenshot displays the Android Studio interface with three main panels: Viewer, Components, and Properties.

- Viewer:** Shows a mobile app preview with a form containing a label "Dein Name:", a text input field, a large multi-line text area, a "Send" button, and a "Clear History" button. Below the preview, the "Non-visible components" section shows the "FirebaseDB1" component highlighted.
- Components:** A tree view showing the hierarchy of components: Screen1, HorizontalArrangement1 (containing Label1 and tbName), tbChatHistory, HorizontalArrangement2 (containing btSend, tbChatMsg, and btClear), and the highlighted FirebaseDatabase1 component.
- Properties:** The configuration for the FirebaseDatabase1 component, including:
 - FirestoreToken: eyJhbGciOiJIUzI1NiIsInR5cGU6Ij...
 - FirestoreURL: https://chatapp-ccec0.firebaseio.com
 - Use Default:
 - Persist:
 - ProjectBucket: ChatApp

Was bei den Eigenschaften der FirebaseDatabase-Komponente eingetragen werden muss, erfährst du bei deiner Lehrerin (bzw. an der Tafel).



Blöcke:

Zunächst deklarieren und initialisieren wir zwei globale Variable, die wir für den Nutzer und die Nachricht in der Datenbank brauchen:

```
initialize global KEY_USER to "KEY_USER"
```

```
initialize global KEY_CHAT to "KEY_CHAT"
```

Das einfachste ist die Programmierung des Buttons btClear. Hier müssen wir nur den Text in der mittleren Textbox löschen.

```
when btClear .Click
do set tbChatHistory .Text to ""
```

Um das Programm übersichtlicher zu gestalten, schreiben wir uns zwei Prozeduren. Eine, die beschreibt, was passiert, wenn man den Chat betritt. Diese ruft eine weitere Prozedur auf, die sich um die Speicherung des Namens des Chattenden kümmert.

```
to isChatEntered
do
  if not is empty trim tbChatMsg .Text
  then call ValidateNameAndStoreName
  call FirebaseDB1 .StoreValue
  tag get global KEY_CHAT
  valueToStore join trim tbName .Text
  ""
  trim tbChatMsg .Text
```

```
to ValidateNameAndStoreName
do
  if not is empty trim tbName .Text
  then call FirebaseDB1 .StoreValue
  tag get global KEY_USER
  valueToStore tbName .Text
```

Jetzt müssen wir den Senden-Button programmieren. Hier wird im mittleren Textfeld die aktuelle Nachricht eine Zeile nach unten geschrieben, dann die erste Prozedur von oben aufgerufen und anschließend das Eingabefeld für die Nachricht wieder geleert.



```

when btSend .Click
do
  set tbChatHistory . Text to join ( "\n "
  tbChatHistory . Text
  call isChatEntered
  set tbChatMsg . Text to " "
  
```

Es fehlt noch die Programmierung der Datenbank. Wenn die Daten verändert wurden, sollen Sie aus der Datenbank ausgelesen und in das mittlere Textfeld geschrieben werden.

Dazu wird der ausgelesene Wert (value) vor den bisherigen Text geschrieben. Eine neue Zeile erreicht man durch „\n“.

```

when FirebaseDB1 .DataChanged
  tag value
do
  if (get tag = get global KEY_CHAT)
  then
    set tbChatHistory . Text to join ( get value
    "\n "
    tbChatHistory . Text
  
```

Testet die App, indem ihr euch gegenseitig Nachrichten schreibt. Viel Spaß dabei! Und bleibt höflich!



HILFEKARTEN

Steuern eines Lego-Roboters (EV3)

Benötigte Komponenten:

1. Einen BluetoothClient (Palette Connectivity).
2. Einen ListPicker zum Verbinden und einen Button zum Trennen der Verbindung.
3. Je nachdem, was genutzt werden soll, braucht man noch Komponenten für die Motoren oder die Sensoren. Alles dazu findet man in der Palette **Lego Mindstorms**. Für diese Komponenten muss die Eigenschaft BluetoothClient auf den Namen des eingesetzten BluetoothClients geändert werden

Bei den Motoren und Sensoren müssen die Ports auch vorab in den Eigenschaften eingestellt werden.

Blöcke:

Um sich mit dem EV3 zu verbinden, brauchen wir zwei Ereignisse des ListPickers.

In die Auswahlliste werden zunächst alle Adressen der Geräte aufgenommen, die in Reichweite sind und Bluetooth angeschaltet haben.

```

when lpVerbindungen .BeforePicking
do set lpVerbindungen .Elements to BluetoothClient1 .AddressesAndNames
    
```

Nachdem ein Gerät ausgewählt worden ist, ruft man eine Methode auf, die die beiden Geräte über Bluetooth verbindet und stellt gleichzeitig sicher, dass das geklappt hat. Die Connect-Methode liefert dann true zurück und die Anweisung(en) in der Verzweigung werden ausgeführt.

```

when lpVerbindungen .AfterPicking
do if call BluetoothClient1 .Connect address lpVerbindungen .Selection
then set btSoundTest .Enabled to true
    
```

Es ist sinnvoll, hier die Buttons erst zu aktivieren, die eine Bewegung, einen Ton, o.ä. beim Roboter auslösen sollen, um einen Absturz zu vermeiden.

Im folgenden Beispiel wird beim Klick auf den Button btSoundTest der Ton a' (440 Hz) eine Sekunde lang mit Lautstärke 50 gespielt.



```

when btSoundTest .Click
do
  call Ev3Sound1 .PlayTone
    volume 50
    frequency 440
    milliseconds 1000
  
```

Die erste Zeile der Melodie von „Alle meine Entchen...“ liefert:

```

when btEntchen .Click
do
  set global tonzaehler to 1
  set global tonlaenge to 500
  set Clock1 . TimerInterval to 500
  set Clock1 . TimerEnabled to true
  
```

```
initialize global tonlaenge to 500
```

```
initialize global tonzaehler to 1
```

```

initialize global entchentoene to
  make a list
    261.63
    293.66
    329.63
    349.23
    392
    392
  
```



```

when Clock1 .Timer
do
  call Ev3Sound1 .PlayTone
    volume 50
    frequency select list item list
      get global entchentoene
      index get global tonzaehler
    milliseconds get global tonlaenge
  set global tonzaehler to get global tonzaehler + 1
  if get global tonzaehler = 5
  then set global tonlaenge to 1000
  else if get global tonzaehler = 6
  then set Clock1 .TimerInterval to 1000
    set global tonlaenge to 1000
  else if get global tonzaehler = 7
  then set Clock1 .TimerEnabled to false
  
```

Kannst du den Code erklären?

Versuche, das Lied weiterspielen zu lassen.

Ganz am Ende ist es wichtig, dass die Verbindung wieder getrennt wird. Dabei müssen auch Motoren gestoppt und der Sound ausgeschaltet werden. Wer mag, kann eine Nachricht dazu sprechen lassen.

Dazu braucht ihr auf dem Screen eine TextToSpeech-Komponente (Palette Media).

```

when btEnde .Click
do
  call Ev3Sound1 .StopSound
  call Ev3Motors1 .Stop
    useBrake true
  call BluetoothClient1 .Disconnect
  call TextToSpeech1 .Speak
    message "Verbindung getrennt"
  
```

Um den Roboter fahren zu lassen, müssen wir die Motorkomponente programmieren. Im Designer werden die Motoren bei MotorPorts eingegeben.



Schau dir den Code an. Teste ihn aus und ändere die Werte bei power und turnRatio. Kannst du erklären, für was die beiden Werte stehen?

```
when btVorwaerts .Click
do call Ev3Motors1 .RotateIndefinitely
   power 40

when btZurueck .Click
do call Ev3Motors1 .RotateIndefinitely
   power -40

when btMotorStop .Click
do call Ev3Motors1 .Stop
   useBrake true
```

```
when btRechts .Click
do call Ev3Motors1 .RotateSyncIndefinitely
   power 40
   turnRatio 45

when btLinks .Click
do call Ev3Motors1 .RotateSyncIndefinitely
   power 40
   turnRatio -45
```

Der MIT App Inventor (<http://appinventor.mit.edu>) wurde ursprünglich von einem Entwicklerteam um Mark Friedman und Hal Abelson bei Google entwickelt und 2012 an das MIT übergeben.
Der MIT App Inventor wird unter der Creative Commons Attribution-ShareAlike 3.0 Unported License veröffentlicht: <https://creativecommons.org/licenses/by-sa/3.0>