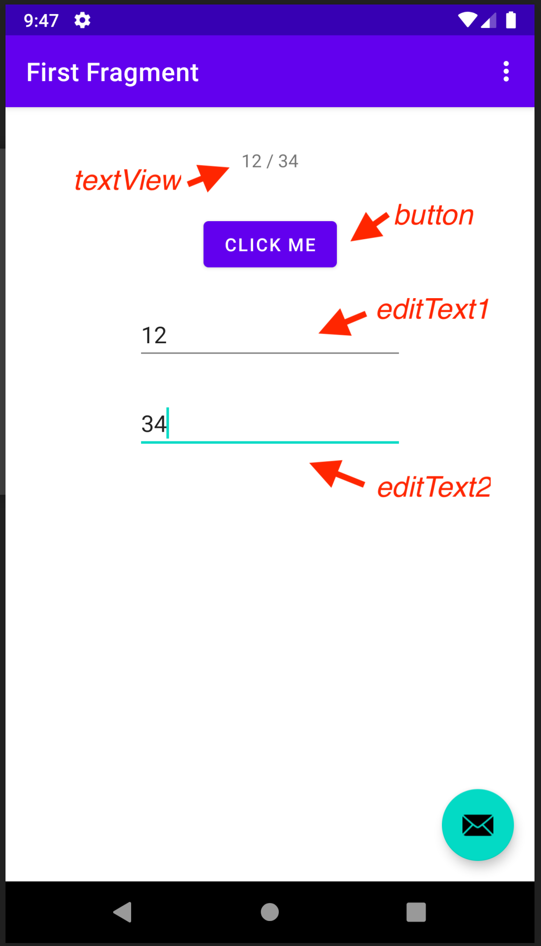
RECAP

```
binding.button.setOnClickListener {
 val t1 = binding.editText1.text.toString()
 val t2 = binding.editText2.text.toString()

 binding.textView.text = t1 + t2.toInt().toString()
}
```

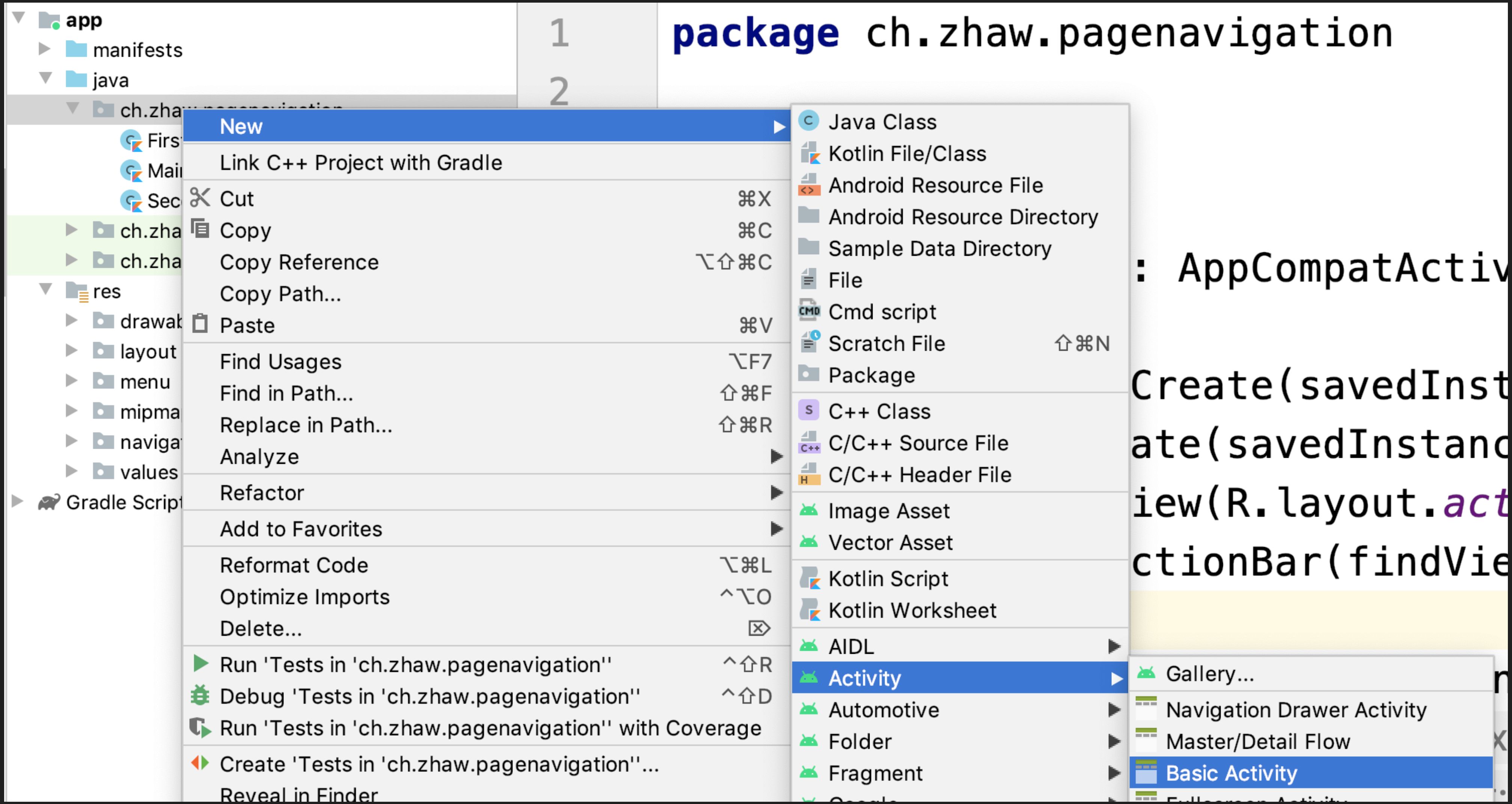
What's the output of this program if the user enters:

- editText1: 1, editText2: 2
- editText1: A, editText2: 2
- editText1: A, editText2: B



SWITCHING BETWEEN ACTIVITIES I

In order to change to a new activity, create one first:



```
package ch.zhaw.pagenavigation

: AppCompatActiv
Create(savedInst
ate(savedInstanc
iew(R.layout.act
ctionBar(findVie
```

SWITCHING BETWEEN ACTIVITIES II

```kotlin
//an Intent is the general way of switching to other activities
//or apps. Intents will be covered in depth later
//for now, let's create the intent to our detail activity
val intent = Intent(this, DetailActivity::class.java)
//and start it
startActivity(intent)
```

SWITCHING BETWEEN ACTIVITIES III

You can pass parameters using an intent by adding them as "extra":

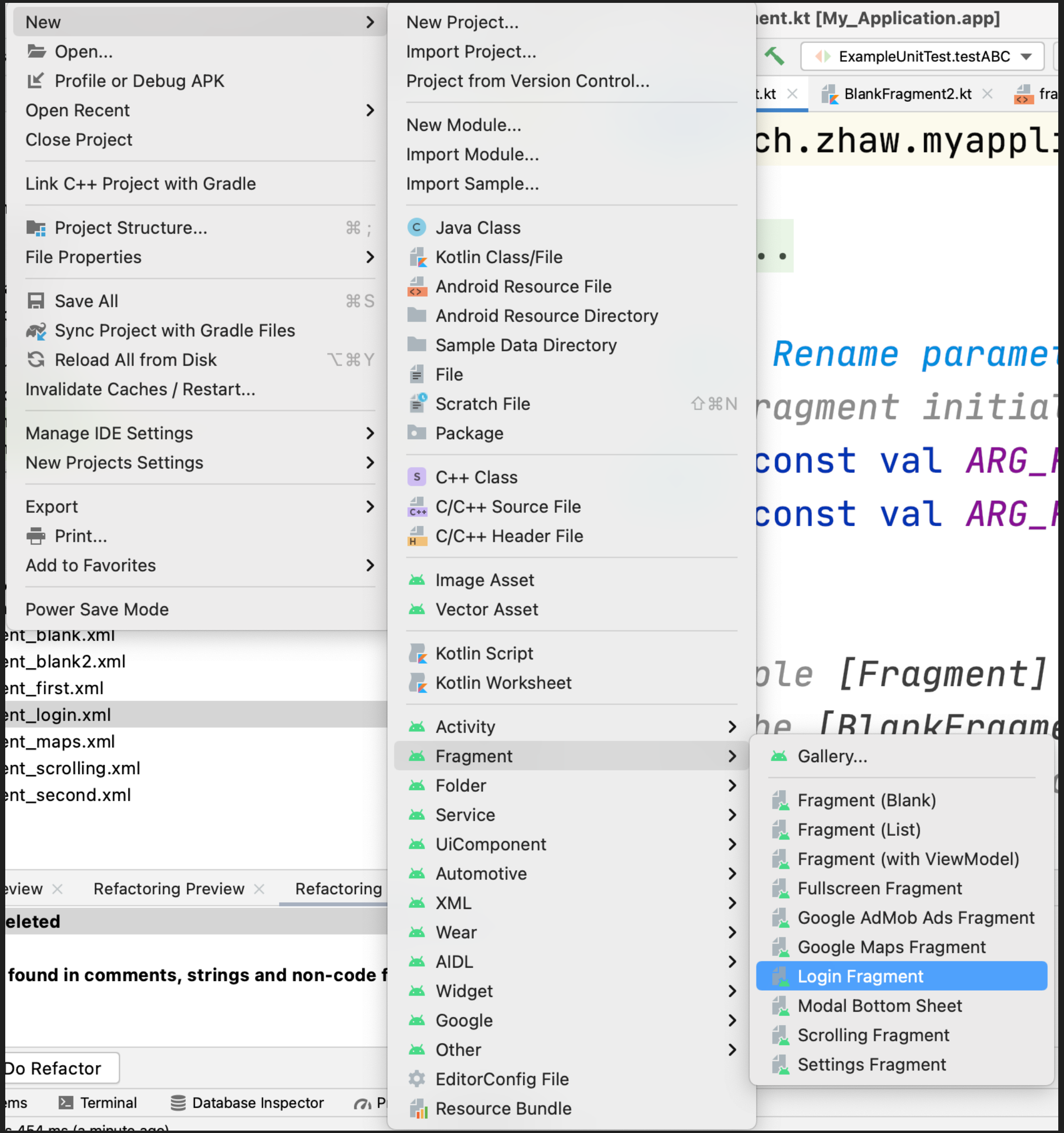
```
//create the intent to our detail activity
val intent = Intent(this, DetailActivity::class.java)
//add some data
intent.putExtra(USERNAME, "peter.muster");

//and start it
startActivity(intent)
```

SWITCHING BETWEEN ACTIVITIES IV

Receive the data in the onCreate method of DetailActivity:

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
 val intent = intent
 //and retrieve the data
 val username = intent.getStringExtra("USERNAME")
 textView.text = username
```

FRAGMENT CREATION I



Unfortunately, Android Studio does not yet support the creation of a new Fragment containing a ViewBinding from a wizard. Workaround: Create a LoginFragment or BlankFragment and choose a name (ie NewFragment).

FRAGMENT CREATION II

```
//delete the autogenerated code and copy this code (adapt the
//name from NewFragment to your name accordingly)
class NewFragment : Fragment() { //inherit from Fragment
 private var _binding: FragmentNewBinding? = null
 private val binding get() = _binding!!

 override fun onCreateView(
  inflater: LayoutInflater, container: ViewGroup?,
  savedInstanceState: Bundle?): View? {
  //adapt also this name (the class will be autogenerated)
   _binding = FragmentNewBinding.inflate(inflater, container, false)
   return binding.root
  }
 override fun onViewCreated(view: View,savedInstanceState: Bundle?){
      super.onViewCreated(view, savedInstanceState)
 }}
```

FRAGMENT CREATION III

```
<?xml version="1.0" encoding="utf-8"?> <!-- Copy this layout -->
<androidx.constraintlayout.widget.ConstraintLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".SecondFragment">
    <TextView
    android:id="@+id/a_textview"
    android:text="XXX"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toStartOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

HOST FRAGMENT I

- NavHostFragment provides an area where different fragments could be presented including a navigation between them.
- As the name suggests, NavHostFragment is itself a fragment. Therefore, it can be included in a layout as follows:

```
<fragment
   android:name="androidx.navigation.fragment.NavHostFragment"
   app:defaultNavHost="true"
   app:navGraph="@navigation/nav_graph"
   ... />
```

HOST FRAGMENT II

- android:name: Provide a class name for a fragment to create, in this case the NavHostFragment.
- app:defaultNavHost: Has to be set for back button support.
- app:navGraph: references the nav_graph xml file that stores all fragments to be displayed within in this host.

HOST FRAGMENT III

```
<!-- host fragment within an activity layout-->
 <fragment
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:navGraph="@navigation/nav_graph" />
```

HOST FRAGMENT IV

```
<!-- create a nav_graph that defines the fragments of this host -->
<navigation ...
    android:id="@+id/nav_graph"
    app:startDestination="@id/FirstFragment"> <!-- default -->
    <fragment
        android:id="@+id/FirstFragment"
        android:name="ch.zhaw.fragmentexample.FirstFragment"
        android:label="@string/first_fragment_label">
        <action
            android:id="@+id/action_FirstFragment_to_SecondFragment"
            app:destination="@id/SecondFragment" />
    </fragment>
    <fragment
        android:id="@+id/SecondFragment" ...
    </fragment>
</navigation>
```

HOST FRAGMENT NAVIGATION I

Navigate between Fragments by defining the action in the nav_graph:

```
<action
    android:id="@+id/action_FirstFragment_to_SecondFragment"
    app:destination="@id/SecondFragment" />
```

and using the action within the fragment class:

```
findNavController()
      .navigate(R.id.action_FirstFragment_to_SecondFragment)
```

HOST FRAGMENT NAVIGATION I

Send data to the other fragment by adding them to a bundle:

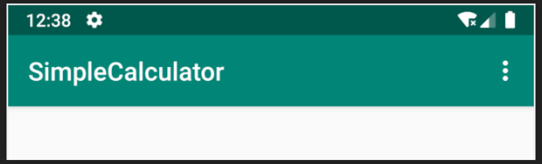
```
val bundle = bundleOf("USERNAME" to "peter.muster")
findNavController().navigate(R.id.action, bundle)
```

Receive the data in the called fragment:

```
override fun onViewCreated(view: View,
  savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)

  val username = arguments?.getString("USERNAME")
```

APP BAR

APP BAR



Typically, an application shows an App Bar on top (directly under the Status Bar). The App Bar:

- has been known as "ActionBar" previously.
- is a special Toolbar component.
- shows the name of the app.
- shows the navigation element.
- might contain further actions like search.

APP BAR SOURCE

Add this to your activity_layout.xml:

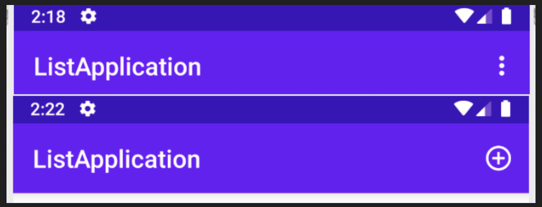
```xml
<!-- The AppBarLayout expects a Toolbar widget -->

<com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay" />

</com.google.android.material.appbar.AppBarLayout>
```

APP BAR REGISTRATION

Finally, register the Toolbar as your App Bar in your onCreate method:

```
setSupportActionBar(binding.toolbar)
```

APP BAR BUTTONS



- The App Bar can show buttons.
- All buttons are defined in menu.xml. The attribute app:showAsAction defines if the button is shown in the collapse menu (three dots, showAsAction="never") or in the bar directly (showAsAction="ifRoom").

DENSITY-INDEPENDENT PIXEL

In order to place components on the screen, Android provides different layout options. When defining the dimension of a component, be sure to use the virtual pixel unit density-independent pixel (dp):

- A dp pixel is equivalent to one physical pixel on a 160 dpi screen.
- A dp pixel is automatically scaled at runtime. A dp pixel can be converted to screen pixels:
  - medium screens: px = dp * (dpi / 160)
  - high density screens: px = dp * (dpi / 240)

RESOURCE FILE STRUCTURE

- All resources are placed in the /res folder.
- Different resource types of the same kind (images, layouts) are placed in different subdirectories with name schema - (drawable-hdpi, mipmap-xhdpi).
- You can check the official documentation for all possible values.

CONTEXT

Accessing the resources requires a reference to the context. The base class can be found in the package android.context.Context

Retrieve a reference to the context in a fragment within requireContext().

In an activity, you can either use "this" or getApplicationContext().

IMAGE FILE RESOLUTIONS I

Whenever you add an image, be sure to add the file in different resolutions (or use vector graphics!):

- LDPI (~120dpi): Low resolution (probably, you might not need this anymore as only 3.3% (Sep. 2015) of all devices use LDPI)
- MDPI (~160dpi): Medium resolution, 48x48 pixels for icons.
- HDPI (~240dpi): High resultion (MDPI*1.5), 72x72 pixels for icons.

IMAGE FILE RESOLUTIONS II

- XHDPI (~320dpi): Extra high resolution (MDPI*2), 96x96 pixels for icons.
- XXHDPI (~480dpi): Extra extra high resolution (MDPI*3), 144x144 pixels for icons.
- XXXHDPI (~640dpi): Extra^3 high resolution (MDPI*4), 192x192 pixels for icons.

You can find a table containing the dpis for different older devices here: http://qualitytestingtips.blogspot.ch/2013/08/ldpi-mdpi-hdpi-xhdpi-xxhdpi.html

ICONS

Icons are added to the res/drawable- folder. Use them in your xml:

```
<android.support.design.widget.FloatingActionButton
  ...
    app:srcCompat="@drawable/ic_time_to_leave_black_24dp"
/>
```

LOCALIZATIONS

- Language dependent strings can be moved to values/strings.xml:

```
<string name="authenticate_button_text">Anmelden</string>
```

Use these strings in your xml layout:

```
<button android:text="@string/authenticate_button_text" ...>
```

SCREEN ORIENTATION

If you want to have a different layout for landscape mode, create a new layout directory "layout-land". Put your layout.xml file there. It will be used automatically in landscape mode.

- standard layout in /res/layout/
- landscape layout in /res/layout-land/

Problem: Whenever you make changes, you need to check two files.

EXERCISE

Create a new "Blank Activity" project and add a landscape layout. Copy the content from portrait to landscape layout. Change the toolbar color in landscape mode. Test your app in both modes.

ANDROID LAYOUT

ANDROID LAYOUT PARAMETERS

layout_width and layout_height specify the width and height of a view. There are three different ways to do this:

- An exact number: A static size, ie. 200dp.
- WRAP_CONTENT: The view should be just big enough to enclose its content
- MATCH_PARENT: The view should be as big as its parent.
- MATCH_CONSTRAINT (0dp): The view should be sized in accordance to the constraints.

In constraint layout, you typically only use WRAP_CONTENT and MATCH_CONSTRAINT.

CONSTRAINT LAYOUT

Constraint layout can be used to define constraints between view components. It is now the default layout.

Each view element needs to have at least two constraints, one for the x and another for the y position.

RELATIVE POSITIONING I

- Basic idea: Position one view relative to one another:

```
<Button android:id="@+id/right_button" ...
  app:layout_constraintLeft_toRightOf="@+id/left_button" />
```

RELATIVE POSITIONING II

All other relative positioning tags:

```
    layout_constraintLeft_toLeftOf
    layout_constraintLeft_toRightOf
    layout_constraintRight_toLeftOf
    layout_constraintRight_toRightOf
    layout_constraintTop_toTopOf
    layout_constraintTop_toBottomOf
    layout_constraintBottom_toTopOf
    layout_constraintBottom_toBottomOf
    layout_constraintBaseline_toBaselineOf
    layout_constraintStart_toEndOf
    layout_constraintStart_toStartOf
    layout_constraintEnd_toStartOf
    layout_constraintEnd_toEndOf
```

RELATIVE POSITIONING III

You can also set a margin:

```
    android:layout_marginStart <!--Start and End are the same -->
    android:layout_marginEnd   <!--As left/right but will be -->
    android:layout_marginLeft  <!--Switched on right to left -->
    android:layout_marginRight <!--GUIs-->
    android:layout_marginTop
    android:layout_marginBottom
```

CONSTRAINT CENTERING I

If you define two constraints that cannot be satisfied simultaneously, Android will center the component for you:

```
<!--This button will be centered vertically within its parent-->
<Button android:id="@+id/button" ...
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent" />
```

CONSTRAINT CENTERING II

Set a bias to force Android to change centering for favoring one side over another:

```
<!--This button will be placed with the left side shorter (1/3)
 than the right side (2/3).-->
<Button android:id="@+id/button" ...
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintHorizontal_bias="0.3" />
```

VIEW CHAINS I

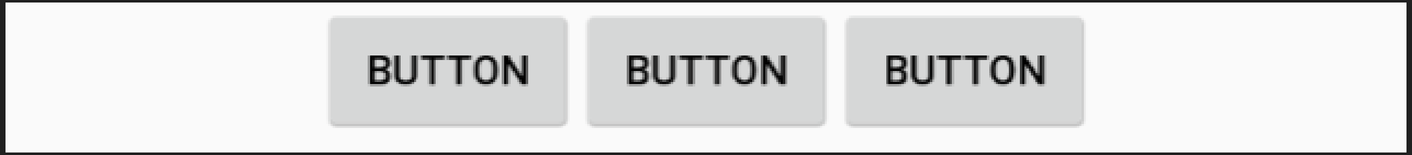
A chain is a series of views that are linked vie double directional connections (double linked list). A chain is either vertically or horizontally.

```
<!-- A chain with two buttons -->
<Button android:id="@+id/button1" ...
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="@id/button2" />

<Button android:id="@+id/button2" ...
    app:layout_constraintLeft_toLeftOf="@id/button1"
    app:layout_constraintRight_toRightOf="parent" />
```

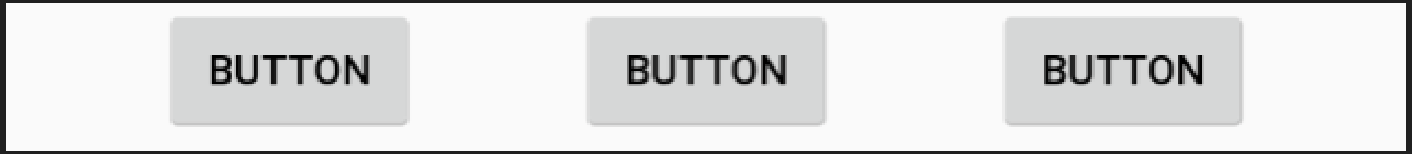
When using a chain, all settings are done on the first (top or left) view.

VIEW CHAINS II

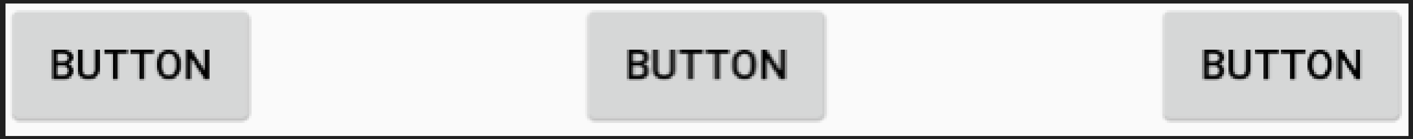
Chains support three different modes:



Packed



Spread



Spread Inside

```
<!--Set this attribute on your first view -->
app:layout_constraintHorizontal_chainStyle="packed"
```

ANDROID LISTS

LISTVIEW

Android provides a ListView component:

- All elements usually have the same layout.
- A user can interact with the list elements.
- The connection between the data and view is done using an adapter.
- The list elements can be layouted in xml (but there are adapters that already offer you a layout).

LISTVIEW ADAPTER

- The adapter connects the data model with the entries in the list.
- Android provides a BaseAdapter to inherit from but also Adapters for other data models like ArrayAdapter or CursorAdapter.

LISTVIEW EXAMPLE

```xml
<!--Add a listview to your layout-->
<ListView
    android:id="@+id/user_list"
    android:layout_width="0dp"
    android:layout_height="0dp">
</ListView>
```

```kotlin
val data = mutableListOf("1. Element", "2. Element", "3. Element",
    "4. Element")

val adapter : ArrayAdapter<String> = ArrayAdapter<String>(
    baseContext,
    android.R.layout.simple_list_item_1, android.R.id.text1, data);

user_list.adapter = adapter
```

LISTVIEW MODEL

```
//you can use a class to store your model
class Person(val name: String, val street : String) {}

class NewFragment : Fragment() {
  val data = mutableListOf(
    Person(name="Peter Muster", street="Musterstrasse 1"),
    Person(name="Paul Frey", street="Bahnhofstrasse 12")
    ...)
```

LISTVIEW ADAPTER I

```
class PersonAdapter(var persons: MutableList<Person>,
  val context : Context) : BaseAdapter() {
    var layoutInflater : LayoutInflater
    private var _binding: PersonCellBinding? = null
    private val binding get() = _binding!!
    private var bindings = mutableMapOf<View,CellLayoutBinding>()
    init {
        layoutInflater = LayoutInflater.from(context)
    }
    override fun getCount(): Int { //number of elements to display
        return persons.size}
    override fun getItem(index: Int): Person { //item at index
        return persons.get(index)}
     override fun getItemId(index: Int): Long { //itemId for index
        return index.toLong()
    }
```

LISTVIEW ADAPTER II

```
override fun getView(index: Int, oldView: View?,
  viewGroup: ViewGroup?): View {
    var view : View
    if (oldView == null) { //check if we get a view to recycle
        _binding = PersonCellBinding.inflate(layoutInflater,
          viewGroup, false)
        view = binding.root;bindings[binding.root] = binding
    }
    else {  //if yes, use the oldview
        view = oldView
        _binding = bindings[view]
    }
    val person = getItem(index) //get the data for this index
    binding.name.text = person.name
    binding.street.text = person.street
    return view}
```

LISTVIEW ADAPTER III

```
<!--you can create your own cell layout (person_cell.xml) -->
<androidx.constraintlayout.widget.ConstraintLayout ...>

    <TextView
        android:id="@+id/name"
        ...></TextView>

    <TextView
        android:id="@+id/street"
        ...></TextView>
</androidx.constraintlayout.widget.ConstraintLayout>
```

LISTVIEW DATA UPDATES

Update the model and propagate a data changed event to the adapter:

```
//change the model
data.add(Person(name="Paul Muster", street="Musterstrasse 16"))

//propate the data change
adapter?.notifyDataSetChanged()
```

EXCERCISE: SBB STATIONBOARD I

Write an SBB Stationboard app in Android. We start with a static list. Create a ListView that displays the first three entries from this mutable list:

```
class StationboardEntry(val name: String, val to : String) {};

val stationboard = mutableListOf<StationboardEntry>(
    StationboardEntry("IC 817", "Romanshorn"),
    StationboardEntry("IC 820", "Brig"),
    StationboardEntry("S7 18753", "Rapperswil"))
```

JSON PARSING

You can parse a JSON structure by:

- an integrated stream parser.
- an external library like Klaxon or GSON that converts the JSON to a set of objects.

KLAXON: PROCESS

1. Add Klaxon to your dependencies:

   ```
   implementation 'com.beust:klaxon:5.5'
   ```

2. Create a backing object (tree) that matches your JSON structure.
3. Read JSON from file or server.
4. Convert JSON to your backing object.

KLAXON: JSON EXAMPLE

File persons.json:

```
{
  "persons":
  [{"name": "Peter Muster", "street": "Musterstrasse 1"},
   {"name": "Paul Frey", "street": "Bahnhofstrasse 12"},
   {"name": "Erwin Dobler", "street": "Brückstrasse 16"},
   {"name": "Walter Loder", "street": "Bachweg 17"},
   {"name": "Maria Weibel", "street": "Aareweg 5"}]
}
```

KLAXON: BACKING OBJECT

```
class Person(val name: String, val street : String) {}

class Persons(val persons : MutableList<Person>) {}
```

Note that you only need to define the properties you are interested in!

KLAXON: PARSING

```
//read JSON from file
//the file "persons.json" is stored in res/raw/
val inputStream =
  requireContext().resources.openRawResource(R.raw.persons)

//convert
val persons = Klaxon().parse<Persons>(inputStream)

//use persons...
```

EXERCISE: SBB STATIONBOARD II

Now we will read a static file. Download the JSON file "stationboard.json" from Moodle and present it in a ListView:

- You can store the JSON file in your resources (res/raw/stationboard.json).
- You can receive an InputStream of this file with this line:

```
val inputStream =
   baseContext.resources.openRawResource(R.raw.stationboard)
```

SERVER CALLS I

In order to retrieve data from a server, you need to have Internet permission. This is added in your manifest file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="zhaw.ch.gsontest">

    <uses-permission android:name="android.permission.INTERNET" />

    <application ...
</manifest>
```

SERVER CALLS II

There are different ways to retrieve data from a server:

1. Your own implementation that runs in its own thread.
2. A library like Volley that does the work for you.

SERVER CALLS: VOLLEY

1. Add volley to your dependencies:

```
implementation 'com.android.volley:volley:1.2.1'
```

2. Call the server.
3. Retrieve either the answer from the server or an error.

SERVER CALLS: VOLLEY SERVER CALL

```
//create a request queue
val requestQueue = Volley.newRequestQueue(requireContext())

//define a request.
val request = StringRequest(
  Request.Method.GET, ENDPOINT,
  Response.Listener<String> { response ->
    //find the response string in "response"
  },
  Response.ErrorListener {
    //use the porvided VolleyError to display
    //an error message
  })

//add the call to the request queue
requestQueue.add(request)
```

EXERCISE: SBB STATIONBOARD III

Next, we will add an internet server call. The following endpoint:

https://transport.opendata.ch/v1/stationboard?station=Winterthur

shows the next departing trains from Winterthur Hbf. Call this endpoint to retrieve the JSON. Hint: Remember to add a to your manifest file.

RECYCLERVIEW

- The RecyclerView works in a similar way to ListView.
- It has been introduced with Material Design and provides some optimizations (better layouting the cells, animations, optimized performance, ...).
- Using a RecyclerView is a bit more complex as you need to define a holder class.

RECYCLERVIEW EXAMPLE I

```xml
<androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="0dp"
        android:layout_height="0dp" />
```

```kotlin
//create the data to display
class Person(val name: String, val street : String) {}
val data = mutableListOf(
    Person(name="Peter Muster", street="Musterstrasse 1"),
    ...)

//add the adapter
recyclerView.adapter = PersonAdapter(data)

//and a layout manager (needed!). This layout defines the
//positioning of the cells
recyclerView.layoutManager = LinearLayoutManager(this)
```

RECYCLERVIEW EXAMPLE II

The RecyclerView needs a Holder class when creating the adapter. The Holder class should store a reference to the components for performance issues:

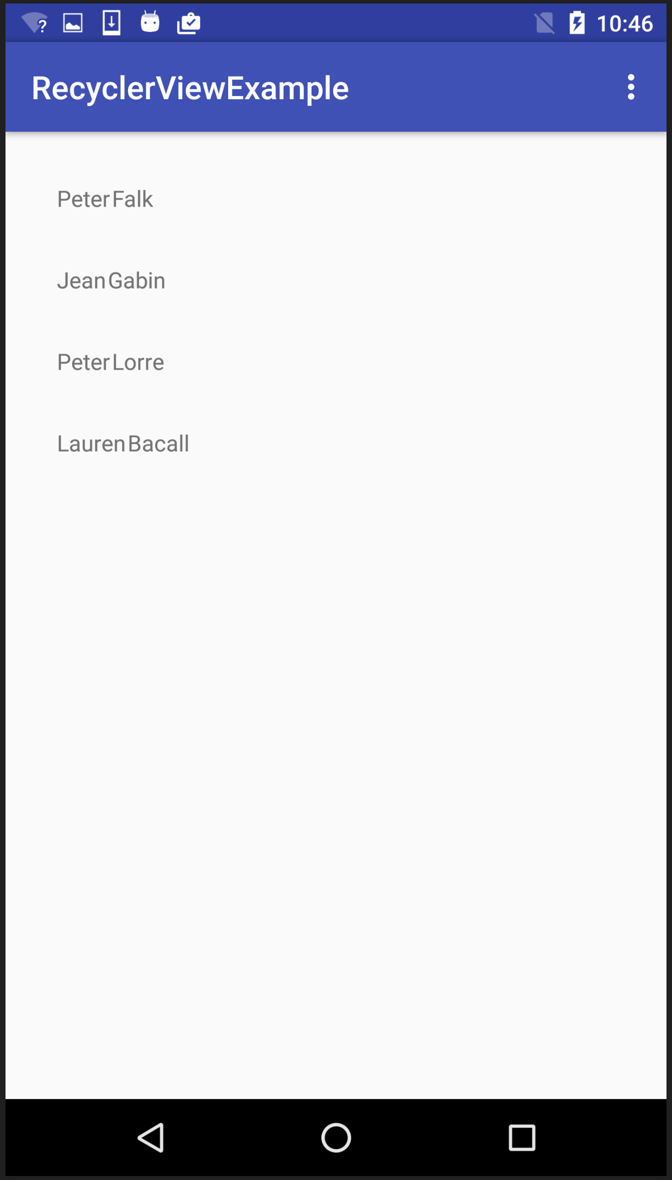
```
class PersonViewHolder(inflater: LayoutInflater, parent: ViewGroup):
  RecyclerView.ViewHolder(inflater.inflate(R.layout.person_cell,
    parent, false)) {
        private var nameView: TextView? = null
        private var streetView: TextView? = null

        init {
            nameView = itemView.findViewById(R.id.name)
            streetView = itemView.findViewById(R.id.street)
        }
        fun bind(person: Person) {
            nameView?.text = person.name
            streetView?.text = person.street
        }
    }
```

RECYCLERVIEW EXAMPLE III

```
//now the adapter
class PersonAdapter(private val list : List<Person>)
  : RecyclerView.Adapter<PersonViewHolder>() {

  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
    : PersonViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        return PersonViewHolder(inflater, parent)
  }
  override fun onBindViewHolder(holder: PersonViewHolder,
    position: Int) {
        val movie: Person = list[position]
        holder.bind(movie)
  }

  override fun getItemCount(): Int = list.size}
```

RECYCLERVIEW EXAMPLE V

```xml
<!-- Finally, the cell layout -->
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"

  android:layout_width="wrap_content"
  android:layout_height="wrap_content">

  <TextView
 android:id="@+id/name"
 ...></TextView>

  <TextView
 android:id="@+id/street"
 ...></TextView>
```



Peter Falk

Jean Gabin

Peter Lorre

Lauren Bacall

EXERCISE: SBB STATIONBOARD IV

Finally, we can extend the class to view the departing time. The following method can be used to convert a unix epoch to an HH:mm string:

```
private fun getDateTime(l: Long): String? {
    try {
        val sdf = SimpleDateFormat("HH:mm")
        sdf.timeZone = TimeZone.getDefault()
        val netDate = Date(l*1000)
        return sdf.format(netDate)
    } catch (e: Exception) {
        return e.toString()
    }
}
```