ANDROID - Lab 2

Lesson goals:

- Introduction to fragments
- Create and customize more complex views (ConstraintLayout, shape drawables, icons, data binding)
- Create a list with a custom adapter
- Introduction to notifications

Fragments

A **Fragment** represents a reusable portion of your app's UI. A fragment defines and manages its own layout, has its own lifecycle, and can handle its own input events. Fragments cannot live on their own, they must be *hosted* by an activity or another fragment. The fragment's view hierarchy becomes part of, or *attaches to*, the host's view hierarchy.

For example, when the activity is paused the fragments are also in pause state, and when activity is destroyed all fragments are destroyed. While an activity is running, we can manipulate each fragment independently, such as add or remove them. Also when we perform fragment transactions we can also add it to a back stack that will be managed by the activity.

Fragments introduce modularity and reusability into your activity's UI by allowing you to divide the UI into discrete chunks. Activities are an ideal place to put global elements around your app's user interface, such as a navigation drawer.

getContext() from the fragment

A fragment does not have a context of its own. The getContext() method called from a fragment returns the Context which is linked to the Activity from which it is called.

Create a fragment

Fragment requires a dependency on the AndroidX Fragment library (the new and improved android support library).

To include the AndroidX Fragment library to your project, add the following dependencies in your app's *build.gradle(module)* file, then click **sync now** and wait a little while for the library to be imported.

```
dependencies {
        implementation 'androidx.fragment:fragment:1.3.0'
}
```

To create a fragment class you need to extend the AndroidX Fragment class. Then in the constructor pass the layout.

```
public class FirstFragment extends Fragment {
  public FirstFragment() {
    super(R.layout.fragment_first);
  }
}
```

In order to add your fragment to your activity you have to add to the activity's layout a **FragmentContainerView** that defines the location where your fragment should be placed.

```
<androidx.fragment.app.FragmentContainerView
  android:id="@+id/fragment_container"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:name="ro.example.myapplication.FirstFragment"/>
```

The **name** attribute specifies the class name of the Fragment to instantiate.

This is one way to instantiate the fragment, the other way is by omitting the name attribute and specifying the fragment programmatically from the activity.

Adding, removing or replacing a fragment is called a **transaction**. Responsible for these transactions is the **FragmentManager** class.

You might never interact with FragmentManager directly if you're using the Jetpack Navigation library, as it works with the FragmentManager on your behalf, but this topic will not be covered in this course.

So, in your activity you can get an instance of the FragmentManager, which will be used to create a FragmentTransaction.

```
if (savedInstanceState == null) {
  getSupportFragmentManager().beginTransaction()
        .setReorderingAllowed(true)
        .add(R.id.fragment_container, FirstFragment.class, null)
        .commit();
}
```

The setReorderingAllowed should be true to allow FragmentManager to properly execute transactions, particularly when it operates on the back stack and runs animations and transitions.

The third argument of the add() method is the arguments you can pass on to the fragment. The argument must be a bundle.

Now, add a TextView to your fragment's layout, then press run. You have your first fragment created.

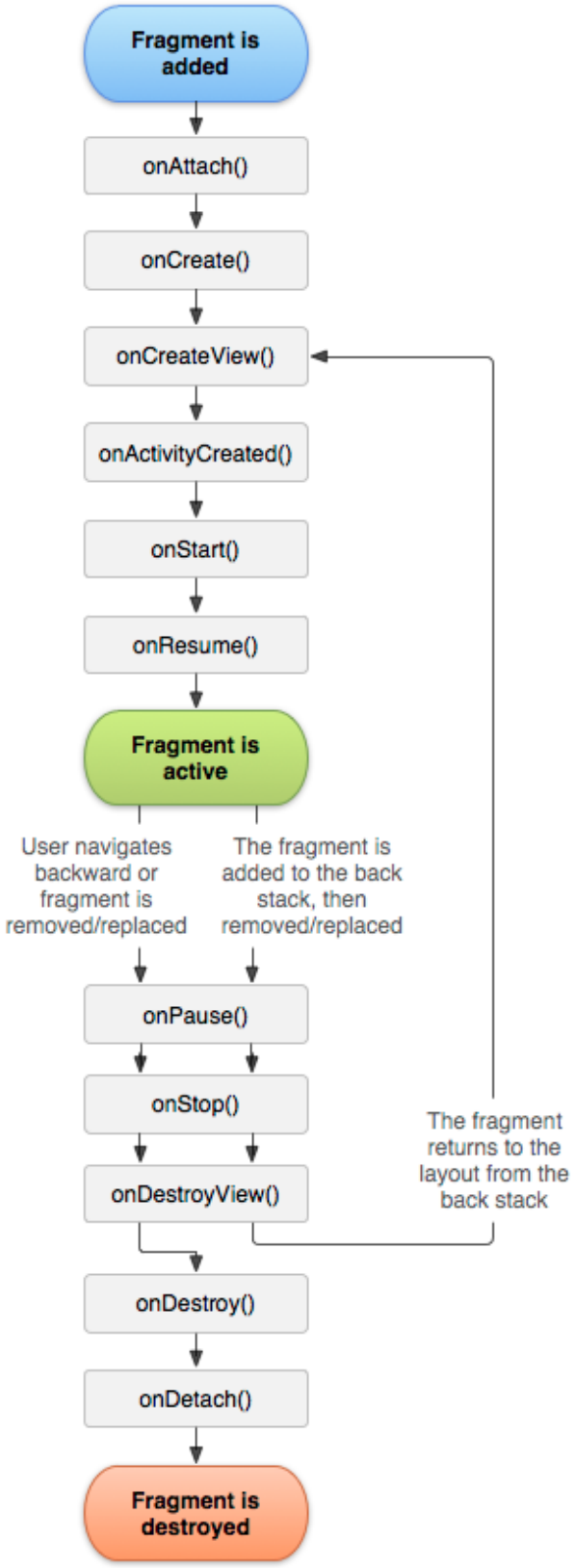
Fragment Lifecycle

Each fragment has its own lifecycle.

It contains callback methods similar to an activity, such as onCreate(), onStart(), onPause(), and onStop().

onAttach() - Called when the fragment has been associated with the activity.

onCreateView() - The system calls this when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a View from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.



RecyclerView

RecyclerView makes it easy to efficiently display large sets of data. You supply the data and define how each item looks, and the RecyclerView library dynamically creates the elements when they're needed.

As the name implies, RecyclerView *recycles* those individual elements. When an item scrolls off the screen, RecyclerView doesn't destroy its view. Instead, it reuses the view for new items that have scrolled on screen. This reuse vastly improves performance, improving your app's responsiveness and reducing power consumption.

Several different classes work together to build your dynamic list:

- **`RecyclerView`** is the `ViewGroup` that contains the views corresponding to your data. It's a view itself, so you add `RecyclerView` into your layout the way you would add any other UI element.

```
<androidx.recyclerview.widget.RecyclerView
  android:id="@+id/recycler_view"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"/>
```

- Each individual element in the list is defined by a *view holder* object. When the view holder is created, it doesn't have any data associated with it. After the view holder is created, the `RecyclerView` *binds* it to its data. You define the view holder by extending `RecyclerView.ViewHolder`.

- The `RecyclerView` requests those views, and binds the views to their data, by calling methods in the *adapter.* You define the adapter by extending `RecyclerView.Adapter`.

- The *layout manager* arranges the individual elements in your list. There are three layout managers provided by the library

    - **LinearLayoutManager** arranges the items in a one-dimensional list.
    - **GridLayoutManager** arranges all items in a two-dimensional grid:
    - **StaggeredGridLayoutManager** is similar to GridLayoutManager, but it does not require that items in a row have the same height (for vertical grids) or items in the same column have the same width (for horizontal grids)

In order to use RecyclerView in your project you need to add its dependencies

```
implementation "androidx.recyclerview:recyclerview:1.1.0"
```

When you define your adapter, you need to override three key methods:

- onCreateViewHolder(): RecyclerView calls this method whenever it needs to create a new ViewHolder. The method creates and initializes the ViewHolder and its associated View, but does *not* fill in the view's contents—the ViewHolder has not yet been bound to specific data.
- onBindViewHolder(): RecyclerView calls this method to associate a ViewHolder with data. The method fetches the appropriate data and uses the data to fill in the view holder's layout. For example, if the RecyclerView displays a list of names, the method might find the appropriate name in the list and fill in the view holder's TextView widget.
- getItemCount(): returns the size of the data set. RecyclerView uses this to determine when there are no more items that can be displayed.

```java
public class CustomAdapter extends RecyclerView.Adapter<CustomAdapter.MovieViewHolder> {

    private String[] localDataSet;

    public CustomAdapter(String[] dataSet) {
        localDataSet = dataSet;
    }

    // Create new views (invoked by the layout manager)
    @NonNull
    @Override
    public MovieViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
        // Create a new view, which defines the UI of the list item
        View view = LayoutInflater.from(viewGroup.getContext())
                .inflate(R.layout.item_movie, viewGroup, attachToRoot: false);
        return new MovieViewHolder(view);
    }

    // Replace the contents of a view (invoked by the layout manager)
    @Override
    public void onBindViewHolder(MovieViewHolder viewHolder, final int position) {
        // Get element from your dataset at this position and replace the
        // contents of the view with that element
        viewHolder.bind(localDataSet[position]);
    }

    // Return the size of your dataset (invoked by the layout manager)
    @Override
    public int getItemCount() {
        return localDataSet.length;
    }

    public static class MovieViewHolder extends RecyclerView.ViewHolder {
        private final TextView textView;

        public MovieViewHolder(View view) {
            super(view);
            textView = (TextView) view.findViewById(R.id.textView);
        }

        public void bind(String item){
            textView.setText(item);
        }
    }
}
```

ConstraintLayout

Next you will create a layout for the movie item. It will consist of an image, a title and duration.

This time the root ViewGroup will be ConstraintLayout. This is a ViewGroup which allows you to position and size widgets in a flexible way.

Here you can see a great visual representation of the constraint types:
https://developer.android.com/reference/androidx/constraintlayout/widget/ConstraintLayout
Or in here
https://developer.android.com/training/constraint-layout

The first thing to keep in mind is that ConstraintLayout's children must have at least one constraint on the vertical axis (top, bottom) and one on the horizontal(left, right).

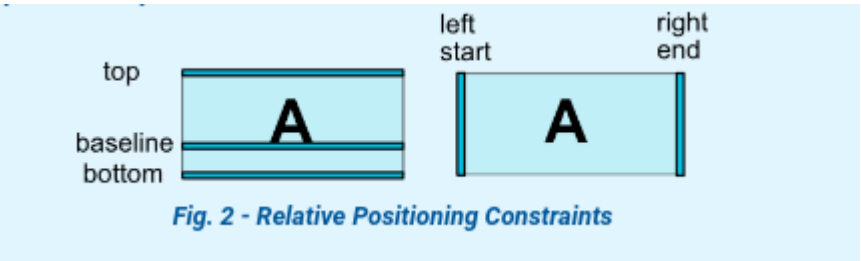


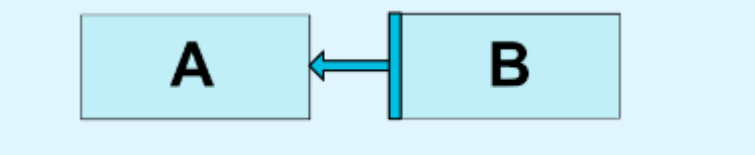
Fig. 2 - Relative Positioning Constraints

A constraint of a view can look like:

```
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent"
```

This means that the View has the start point as the start point of its parent. Instead of the parent keyword you can put an id of a sibling view.

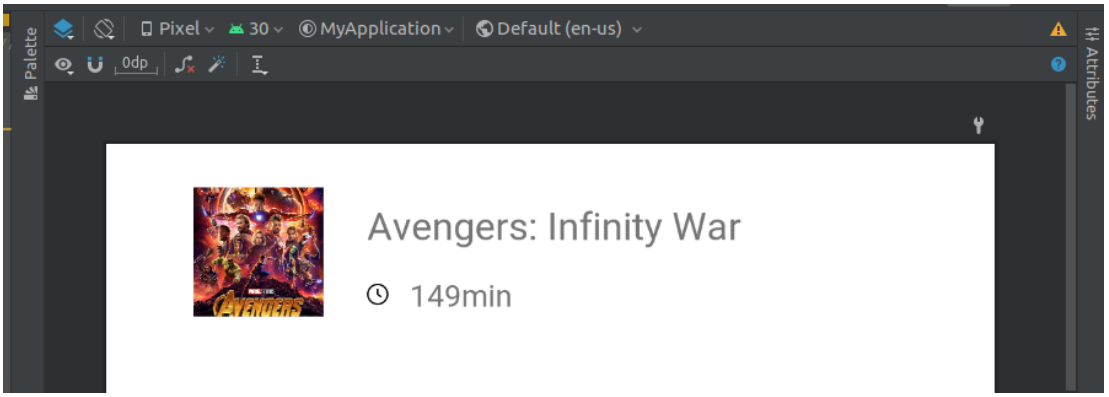
Let's say you have two button A and B and you want B to follow A



To get this you need a constraint on the B button like

```
app:layout_constraintStart_tEndOf="@id/button_a"
```

ConstraintLayout also has many helpful components like Guideline, Barrier and Group, that will not be included in this lab but you can go to the links attached and read more about them.



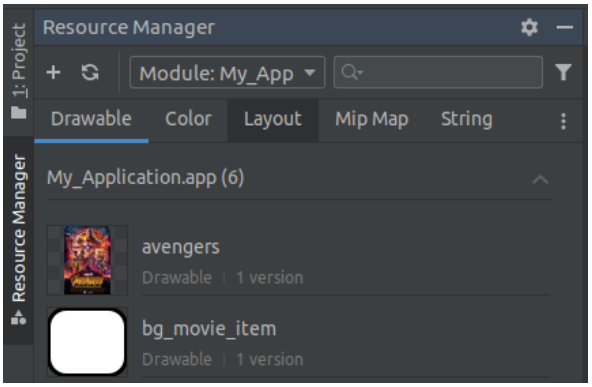
This is how your layout will look like.

First add the ImageView child in your layout and center it vertically with the help of the constraints.

```
<ImageView
    android:id="@+id/iv_picture"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:layout_marginStart="20dp"
    android:src="@drawable/avengers"
    android:scaleType="centerCrop"

    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    />
```

The src attribute holds the reference to the image in your drawable folder.

To add the photo click right on the drawable folder and open **Show in files** and drag your photo to it or Click on the **ResourceManager > + (plus) > ImportDrawables**

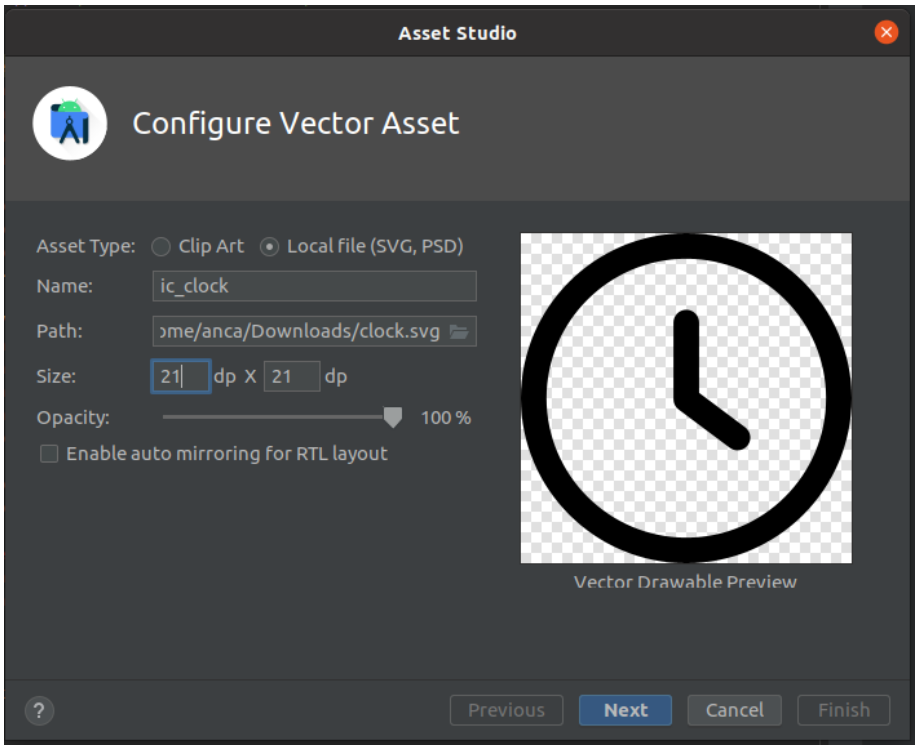


Next add a TextView to hold the movie title and one to hold the duration, next to the image view.

SVG import

In order to import an icon to your project, get the icon from a website (like https://www.flaticon.com/) in SVG format. Then right click on the drawable folder **New > Vector Asset.**

To configure the vector asset, choose **Local file** and choose the path to the icon by clicking the folder icon. Make sure the size of the icon is not too big (21dp is good enough)



The clock icon will be held in an ImageView.

```
<TextView
    android:id="@+id/movie_title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="20dp"
    tools:text="Avengers: Infinity War"
    android:textSize="18sp"

    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toEndOf="@id/iv_picture"
    app:layout_constraintBottom_toTopOf="@id/iv_clock"
    />
```

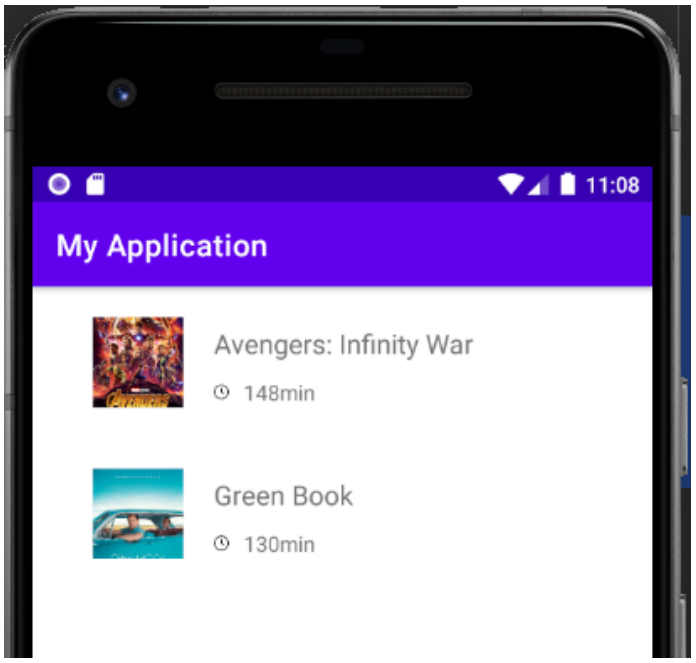
```
<ImageView
    android:id="@+id/iv_clock"
    android:layout_width="10dp"
    android:layout_height="10dp"
    android:layout_marginTop="10dp"
    android:src="@drawable/ic_clock"

    app:layout_constraintTop_toBottomOf="@id/movie_title"
    app:layout_constraintStart_toStartOf="@id/movie_title"
    app:layout_constraintBottom_toBottomOf="parent" />

<TextView
    android:id="@+id/tv_duration"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="10dp"
    tools:text="149min"

    app:layout_constraintStart_toEndOf="@id/iv_clock"
    app:layout_constraintTop_toTopOf="@id/iv_clock"
    app:layout_constraintBottom_toBottomOf="@id/iv_clock"/>
```

Add the layout to your adapter. Then create a model class for your movie with title, duration and imageId fields. Why the imageId because in the **bind** method you will add your image to the ImageView as a Drawable and you will hold in the imageId the id of your drawable.

```
movieImage =  view.findViewById(R.id.iv_picture);
movieImage.setImageDrawable(ContextCompat.getDrawable(
                movieImage.getContext(),
                item.getImageId())
                );
```

The only thing left to do is create an CustomAdapter object and pass it to the RecyclerView, in the Fragment's **onViewCreated** method.

```
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle
savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    CustomAdapter adapter = new CustomAdapter(movieList);
    RecyclerView rv = view.findViewById(R.id.recycler_view);
    rv.setAdapter(adapter);
}
```



ShapeDrawables

Next you will make each list item to look like a card with a stroke.

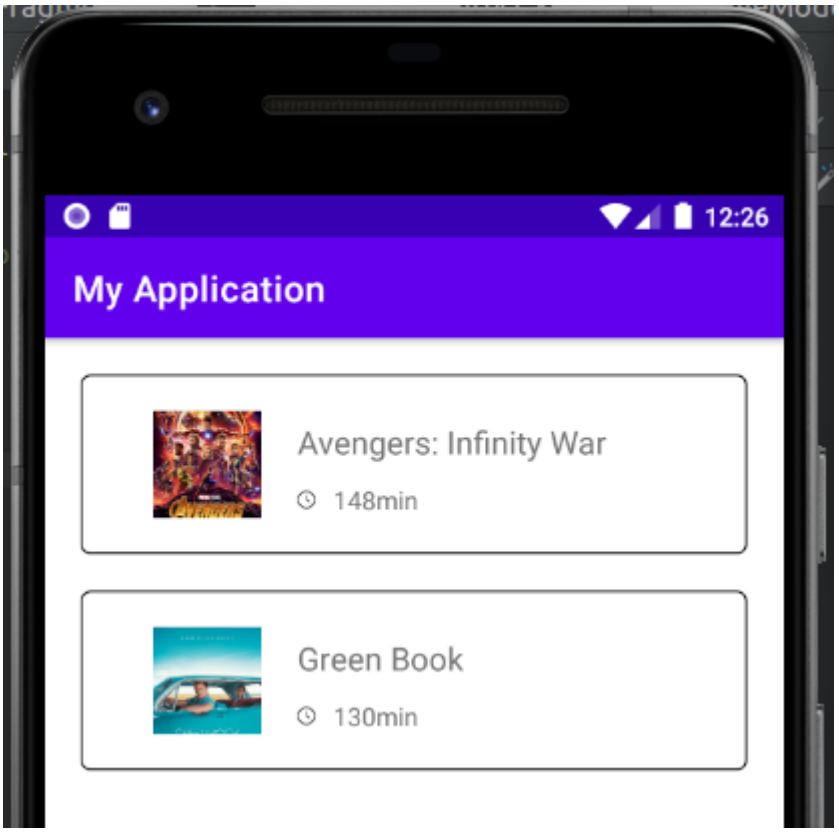
A ShapeDrawable takes a Shape object and manages its presence on the screen. Create one by right clicking drawable folder > New > Drawable resource

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">

<!--    <gradient-->
<!--        android:startColor="@color/purple_200"-->
<!--        android:endColor="@color/teal_200"-->
<!--        android:angle="45"-->
<!--        />-->

    <solid
        android:color="@color/white"/>

    <corners
        android:radius="5dp"/>

    <stroke
        android:width="1dp"
        android:color="@color/black" />

<!--    <padding-->
<!--        android:left="10dp"/>-->

</shape>
```

The shapeDrawable is added to your view on the **background** attribute of the view, in your case the ConstraintLayout of the movie item layout.

```
android:background="@drawable/bg_movie_item"
```

Add some margins to the movie item layout and this is how your app looks now:



This approach won't work on the ImageView, if you want rounded corners for an image you need to work with ShapeableImageView or some library that helps you with this.

Callbacks

Let's say you want to click on a card and open a new screen with the movie details. To do that you need to let the fragment know when an item from the adapter was clicked.

RecyclerView does not have *setOnItemClickListener,* so you'll have to create your own way to do it.

Sending data to an activity or fragment with a bundle is a one way communication, but how do you send data asynchronously? With Callbacks, a request to someone to return to a place. You have already met them in your activity methods(onCreate, onResume etc) and in the interaction with a button, because the most used callback is the View.OnClickListener.

The concept of callbacks is to inform a class if some work in another class is done.

First you'll need an interface that specifies the listener's behaviour.

```
public interface OnItemClickListener {
  void onItemClick(MovieModel item);
}
```

Next, in your adapter you need to define a variable of this interface and add it to the adapter constructor.

```
public static OnItemClickListener itemClickListener;

public CustomAdapter(
      List<MovieModel> dataSet,
      OnItemClickListener listener
) {
  itemClickListener = listener;
  localDataSet = dataSet;
}
```

You will set the click listener to the whole layout, this means you need to access the root element. Add an id to your ConstraintLayout item and define it in your ViewHolder. Inside the onClick method you call the interface method with your item in it.

```
layout =  view.findViewById(R.id.container);
layout.setOnClickListener(new View.OnClickListener(){
  @Override
  public void onClick(View v) {
    itemClickListener.onItemClick(item);
  }
});
```

Back to your fragment, where the recycler view is. Implement the interface and override its method. In this method you will add the logic that you want to happen when an item from your list is clicked.

```
public class FirstFragment extends Fragment implements OnItemClickListener {

@Override
public void onItemClick(MovieModel item) {
        //here goes the logic that will execute after a movie is clicked
        Toast.makeText(getContext(), item.getTitle(), Toast.LENGTH_LONG).show();
}
```

Now your adapter needs the listener parameter, and because your fragment implements the listener you can now send it as parameter with the keyword **this.**

```
CustomAdapter adapter = new CustomAdapter(movieList, this);
```

That's it, hit run and click any of your movie cards.

Fragment Navigation

Next you will open a new fragment when the one item from the recycler view is clicked.

Create a new fragment (SecondFragment) and a layout for it that holds a TextView, for the movie title.

At runtime, a **FragmentManager** can add, remove, replace, and perform other actions with fragments. Each set of fragment changes that you commit is called a *transaction*.

You can save each transaction to a back stack managed by the **FragmentManager**, allowing the user to navigate backward through the fragment changes—similar to navigating backward through activities.

```
FragmentManager fragmentManager = getActivity().getSupportFragmentManager();
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

fragmentTransaction.replace(R.id.fragment_container, new SecondFragment())
        .addToBackStack(null);

fragmentTransaction.commit();
```

Here, the fragment that is in your container (in the activity layout) will be replaced with an instance of a new fragment class, the SecondFragment. Calling replace() is equivalent to calling remove() and then add() the new fragment.

By default the changes made in the FragmentTransactions are not added to the back stack. This means that if you get rid of the `.addToBackStack()`, your second fragment will show and if back is pressed your app will exit. In order to save the changes you call addToBackStack.

You might never interact with FragmentManager directly if you're using the Jetpack Navigation library, as it works with the FragmentManager on your behalf. But this lab will not cover it.

To send data to your SecondFragment, you need to make a bundle and put all the data in it. The bundle will be added in the arguments of the fragment.

```
Bundle bundle = new Bundle();
bundle.putString(MOVIE_TITLE, item.getTitle());

SecondFragment fragment = new SecondFragment();
fragment.setArguments(bundle);

fragmentTransaction.replace(R.id.fragment_container, fragment)
        .addToBackStack(null);
```

In order to get your data from the bundle, in your second fragment:

```
Bundle bundle = this.getArguments();
if (bundle != null) {
  String title = bundle.getString(FirstFragment.MOVIE_TITLE);
  ((TextView) view.findViewById(R.id.title)).setText(title);
}
```

If you want to send not only the title but all the fields in the movie model, you do not have to put every field independently in the bundle. You can use parcelable.

```
bundle.putParcelable(MOVIE,  item);
```

You will see now that AndroidStudio shows an error, that your item is MovieModel, not Parcelable, what it expects. This means that your model needs to implement Parcelable and also define a CREATOR and override some methods.

```
public class MovieModel implements Parcelable {

  ...

  @Override
  public void writeToParcel(Parcel dest, int flags) {
    dest.writeStringArray(new String[] {this.title,
          this.duration,
          this.imageId.toString()});
  }
  public static final Parcelable.Creator CREATOR = new Parcelable.Creator() {
    public MovieModel createFromParcel(Parcel in) {
      return new MovieModel(in);
    }
    public MovieModel[] newArray(int size) {
      return new MovieModel[size];
    }
  };

  public MovieModel(Parcel in){
    String[] data = new String[3];

    in.readStringArray(data);
    // the order needs to be the same as in writeToParcel() method
    this.title = data[0];
    this.duration = data[1];
    this.imageId = Integer.valueOf(data[2]);
  }}
```

Notifications

Notifications provide short, timely information about events in your app.

Notifications are created with NotificationCompat. For this you should make sure that your app's gradle file contains the following dependency:

```
implementation 'androidx.appcompat:appcompat:1.2.0'
```

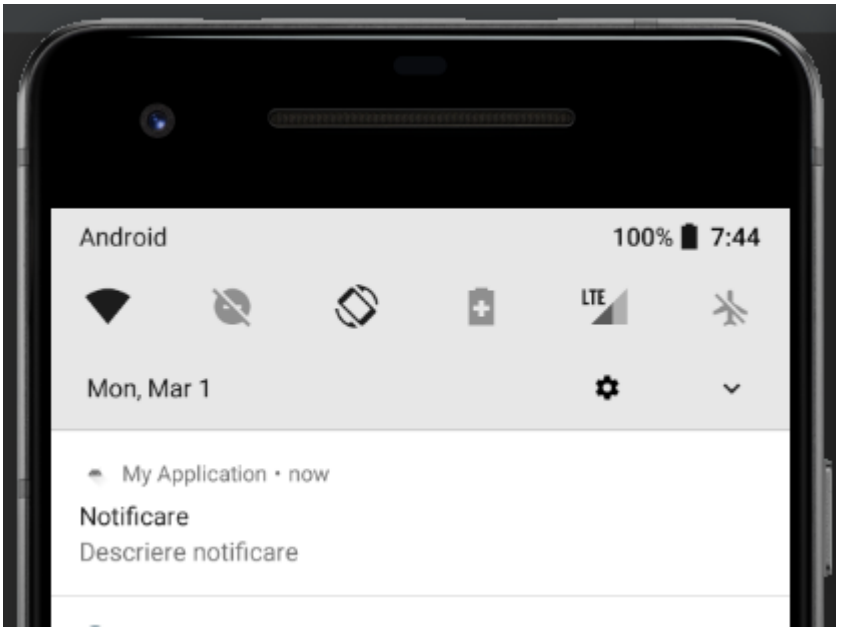
A notification in its most basic and compact form (also known as collapsed form) displays an icon, a title, and a small amount of content text.

For more details about each part of a notification, read about the notification anatomy.

To get started, you need to set the notification's content and channel using a NotificationCompat.Builder object. The following example shows how to create a notification with the following:

- A small icon, set by setSmallIcon(). This is the only user-visible content that's required.
- A title, set by setContentTitle().
- The body text, set by setContentText().
- The notification priority, set by setPriority(). The priority determines how intrusive the notification should be on Android 7.1 and lower. (For Android 8.0 and higher, you must instead set the channel importance)

```
NotificationCompat.Builder builder =
                new NotificationCompat.Builder(getContext(), CHANNEL_ID)
  .setSmallIcon(R.drawable.ic_launcher_foreground)
  .setContentTitle("Notificare")
  .setContentText("Descriere notificare")
  .setPriority(NotificationCompat.PRIORITY_DEFAULT);

NotificationManagerCompat notificationManager =
NotificationManagerCompat.from(getContext());
// notificationId is a unique int for each notification that you must define
notificationManager.notify(id, builder.build());
```



This is how the notification looks, but only if the api of the emulator is below 26. If it is more or equal to 26, your notification will not be displayed, because it needs a channel. Add this to your method.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
  CharSequence name = getString(R.string.channel_name);
  int importance = NotificationManager.IMPORTANCE_DEFAULT;
  NotificationChannel channel = new NotificationChannel(CHANNEL_ID, name,
importance);


  NotificationManager notificationManager =
requireActivity().getSystemService(NotificationManager.class);
  notificationManager.createNotificationChannel(channel);
}
```

For more features of the Notification component, check out:
https://developer.android.com/training/notify-user/build-notification

DataBinding

Data binding is a very helpful library. It allows you to bind UI components (Views or ViewGroups) in your layouts to data sources in your app using a declarative format rather than programmatically.

Simply said, it gives you a more elegant and minimal alternative for the view declaration and application logic to connect to the interface elements

```
TextView textView = findViewById(R.id.sample_text);
textView.setText("...");
```

Becomes

```
dataBinding.sampleText.setText("...");
```

First, to use data binding in your application you need to enable it in the build.gradle file in the app module

```
android {
    ...
    buildFeatures {
        dataBinding true
    }
}
```

Second, your layout needs a little update to let the app know that we want to use data binding on it.

Go to your second fragment layout and wrap it with a **layout** tag.

```
<layout>

        <data> </data>

         <RelativeLayout
            xmlns:android="http://schemas.android.com/apk/res/android"
            android:layout_width="match_parent"
            android:layout_height="match_parent">
                ...
          </RelativeLayout>
</layout>
```

The data hold the layout variables, you can add types of classes with the **import** tag and variables with the **variable** tag. A variable must have a name and a type.

```
<import type=""/>
<variable
  name=""
  type="" />
```

Every layout file that has the **layout** tag as root has its own generated data binding file. The library does that for you. By default, the class name is the name of the layout file with the word binding at the end. In your case it is *FragmentSecondBinding*.

To use the data binding in the fragment, you must first declare a dataBinding variable of the generated class of the layout. Then, in the fragment's onCreateView method, initialize it and return its root. Be careful you delete the default return of the onCreateView method, otherwise your data binding will not work.

```
private FragmentSecondBinding dataBinding;

@Override
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup
container, @Nullable Bundle savedInstanceState) {
  dataBinding = DataBindingUtil.inflate(
      inflater,
      R.layout.fragment_second,
      container,
      false);

  return dataBinding.getRoot();
}
```

Now you can see that with the help of data binding you can access all the views in your layout, as long as they have an id.

In the activity the initialization is a little bit different

```
ActivityMainBinding binding =
      DataBindingUtil.setContentView(this,
      R.layout.activity_main);
```

Nice and easy. You can do a lot more with data binding, for more information: https://developer.android.com/topic/libraries/data-binding

You can also find the source code of this lab in here:
https://github.com/DobrescuAnca/LaboratorAndroid2021