Android Application Development, COMP 10073

Mohawk College, Winter 2021

Communicating Between Classes

Recall that one of the primary goals of Object Oriented Programming is information hiding. Information hiding strives to simplify programming by enforcing a paradigm where each component of an application knows or has access to only the information that it needs to know.

When we are working with a single synchronous thread in a standard object oriented program it is easier to manage the sharing of data than when we are working with multiple asynchronous threads that may potentially modify the same piece of data in unexpected ways.

A privacy leak occurs when one class provides a reference for its data to another class allowing that other class to modify its data. Privacy leaks can be handled in a few ways, the easiest way to always implement some sort of getter that copies the data.

Sharing Data with Getter()/Setter()

Add an EditText widget to the first activity. When the button is pressed extract the string from the widget and save it in a local private variable. Implement a getter for that variable so the other activity can access the data.

```
private static String loginText;
public void nextActivity (View view) {
    EditText loginEdit = (EditText) findViewById(R.id.loginText);
    loginText = loginEdit.getText().toString();
    Intent switch2Activity2 = new Intent(MainActivity.this, MainActivity2.class);
    startActivity(switch2Activity2);
}
public static String getLoginText() {return loginText;}
```

Notice that we generate a new string object via toString() that we store as part of the object in the private static variable loginText. From the second activity we can log the value of loginText by calling the getter from onCreate().

While this works for simple scenarios public static getters don't scale very well. Since the Android system may kill activities in the attempt to manage memory, sharing data references across activities may lead to crashes.

Sending Data to an Activity with putExtra()

https://developer.android.com/reference/android/content/Intent#putExtra(java.lang.String,%20java.lang.String)

Rather than using the global reference to a static getter method it would be nice if we could pass information between activities in the way that we would call a method with some arguments. The Intent object that we use to start the new activity includes a facility for bundling data with the object. Instead of storing data in the current class, we can keep the data local to the button handler and add it to the Intent in the button using the .putExtra() method.

The putExtra method takes two arguments a name and a value. The method is overloaded, you can store a variety of values other than strings.

Your button handler should look like this:

```
public void nextActivity (View view) {
    EditText loginEdit = (EditText) findViewById(R.id.loginText);
    String loginText = loginEdit.getText().toString();
    Intent switch2Activity2 = new Intent(MainActivity.this, MainActivity2.class);
    switch2Activity2.putExtra("loginText", loginText);
    startActivity(switch2Activity2);
}
```

Extracting Intent Extras with getIntent() and getExtra

https://developer.android.com/reference/android/app/Activity.html#getIntent()

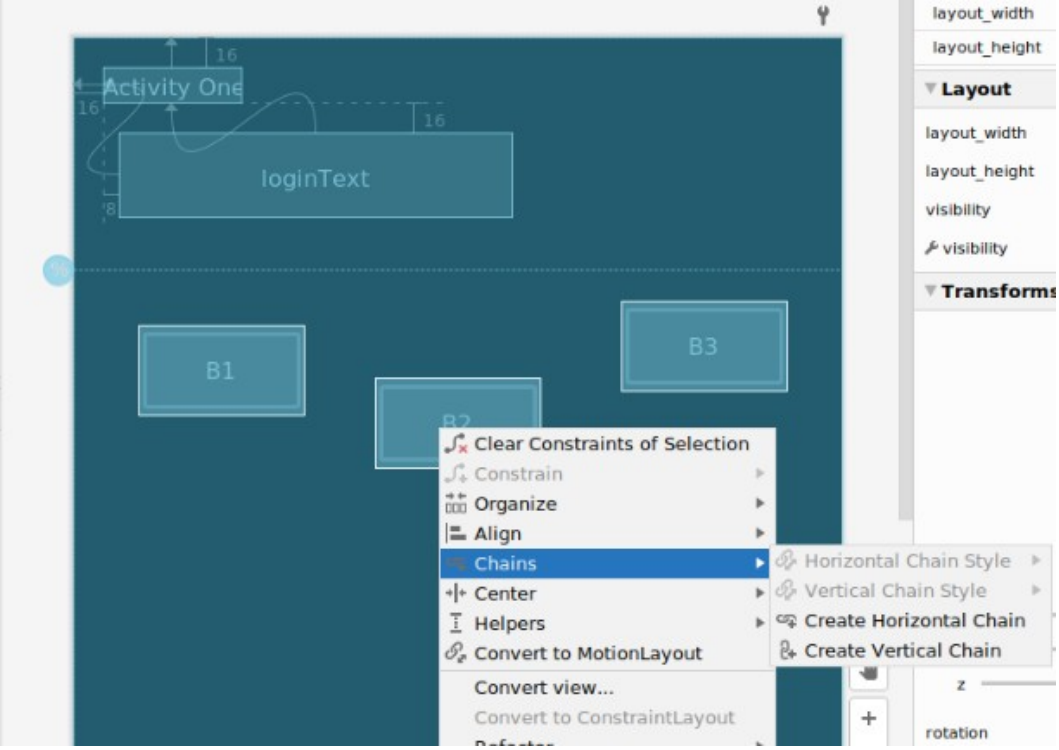
https://developer.android.com/reference/android/content/Intent#getStringExtra(java.lang.String)

In your second activity, the onCreate() method can access the intent that started it with getIntent(). The extra string that was stored in the intent with putExtra(), can be fetched with getStringExtra. For simplicity we can log the data. Our onCreate() method in MainActivity2.java should look something like this:

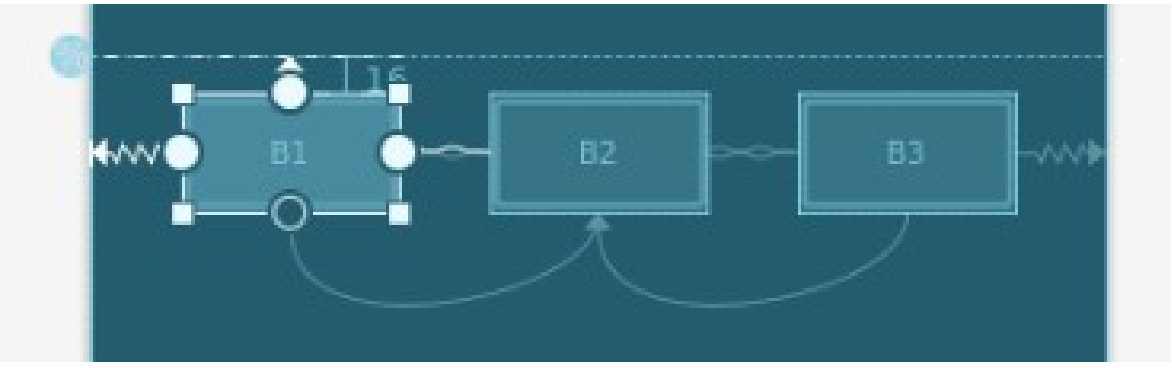
```
public class MainActivity2 extends AppCompatActivity {

    final static String tag = "==LIFECYCLE_A2==";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);

        Intent intent = getIntent();
        String incomingText = intent.getStringExtra("loginText");

        Log.d(tag,"onCreate() - received '" + incomingText + "'");
    }
```

Layout tools and the Guideline

Using our previous layout add several buttons and a horizontal guideline. The guideline has several modes, we can constrain the guideline by a fixed amount to the top or the bottom and also as a percentage. Rather than always anchoring widgets to the top of the activity we can use the guideline to simplify the layout.



Select all the buttons, and right click to bring up a submenu. Link the buttons together with a horizontal chain. This will keep them evenly spaced. With the same right click menu align the bottom edges of the buttons. This will provide vertical anchors for 2 of them. Connect the last unconstrained button to the guideline.



More about the View Object

We can use the same button handler for several widgets. The view argument that gets passed in to the onClick() method can fetch the integer id for the button and the text in the same way that we fetched text from the textEdit widget.

For this implementation we will override the implementation of onClick() that is specified in the interface view.

```
@Override
public void onClick(View view) {
    EditText loginEdit = (EditText) findViewById(R.id.loginText);
    String loginText = loginEdit.getText().toString();
    Button bview = (Button) view;
    String buttonName = bview.getText().toString();   // gets the button label
    int buttonid = view.getId();                      // gets the unique integer id
    String msg = "text = " + loginText + " from '" + buttonName + "' aka id = " + buttonid;

    Intent switch2Activity2 = new Intent(MainActivity.this, MainActivity2.class);
    switch2Activity2.putExtra("loginText", msg);
    startActivity(switch2Activity2);
}
```

We don't need to write a separate handler for each button.

Setting onClick() programmatically via View

The following example looks up the three buttons defined for the interface and sets the handler to the method in the current class named "onClick". To make this work we implement the OnClickListener, defined in the View class and reference this to set the onClick() as our listener.

```
public class MainActivity extends AppCompatActivity
        implements View.OnClickListener {

    final static String tag = "==LIFECYCLE_A1==";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d(tag, "onCreate()");

        Button bview = findViewById(R.id.button1);
        bview.setOnClickListener(this);
        bview = findViewById(R.id.button2);
        bview.setOnClickListener(this);
        bview = findViewById(R.id.button3);
        bview.setOnClickListener(this);
    }
```

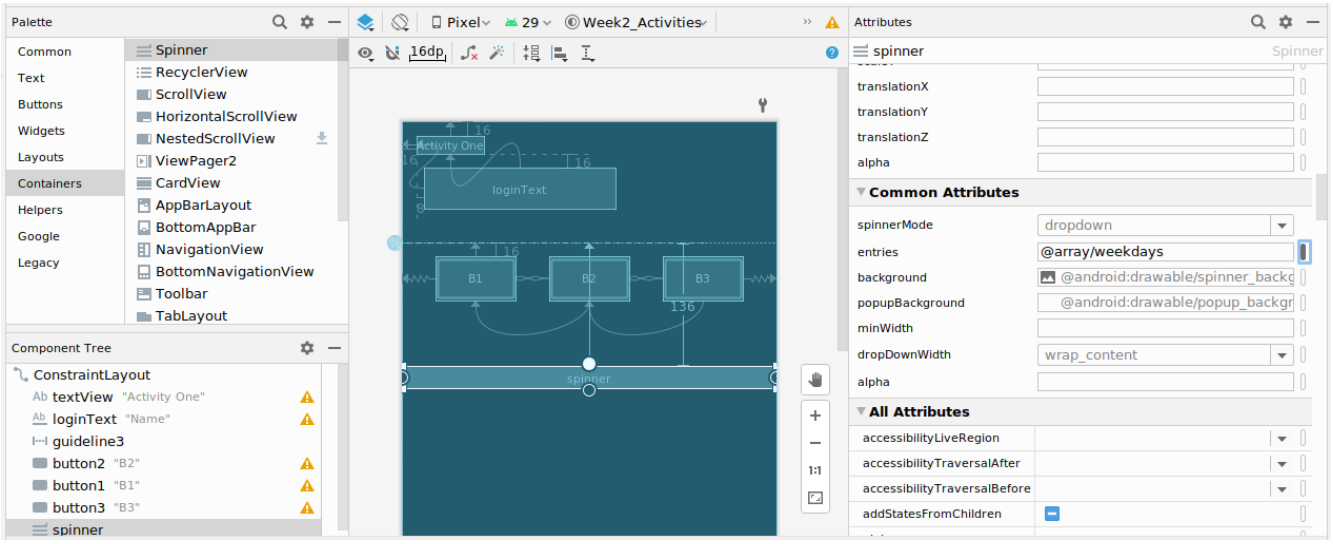
Spinner Widget

https://developer.android.com/guide/topics/ui/controls/spinner

Multiple buttons can become cumbersome. Add a spinner to the layout. The spinner can present a collection of menu items. Add an array of strings representing the days of the week to the strings resource.

```
<resources>
    <string name="app_name">Week2_Activities</string>
    <string-array name="weekdays">
        <item>Monday</item>
        <item>Tuesday</item>
        <item>Wednesday</item>
        <item>Thursday</item>
        <item>Friday</item>
    </string-array>
</resources>
```

Find the spinner under the palette item for containers.



Add the array by clicking on the tab under Common Attributes for entries.

Spinner Setup

In order to setup the Spinner interface we need to implement two default methods.

https://developer.android.com/reference/android/widget/AdapterView.OnItemSelectedListener

onItemSelected() Callback method to be invoked when a new item in this view has been selected. It is not called if there is no change in selection.

onNothingSelected() Callback method to be invoked when the selection disappears from this view or when the adapter becomes empty.

The MainActivity to be modified so that it implements the interface for the spinner. Specify that the interface is implemented and initialize the spinner in the onCreate() method.

```
public class MainActivity extends AppCompatActivity
        implements View.OnClickListener, AdapterView.OnItemSelectedListener {

    final static String tag = "==LIFECYCLE_A1==";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d(tag, "onCreate()");

        Spinner mySpinner = findViewById(R.id.spinner);
        // Set a selection before setting listener,
        // to prevent "ghost" selection on start
        mySpinner.setSelection(0, false);
        mySpinner.setOnItemSelectedListener(this);


        ...
```

The interface for the spinner requires you to override two different methods.

The two methods that will respond to the spinner look like this:

```
@Override
public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
    // Load string array from resources
    String[] weekdays = getResources().getStringArray(R.array.weekdays);
    // log the spinner's choice
    Log.d(tag, weekdays[position]);
}
@Override
public void onNothingSelected(AdapterView<?> parent) {
    // log the spinner's choice
    Log.d(tag, "Nothing Selected");
}
```

Other ways to set up Handlers

We have mentioned two specific ways to add handlers. The easiest way is to use the layout editor. The spinner requires an interface that is more complex than a single function, so it isn't as straightforward to add its handlers.

Add a button with the id a2_button to the second activity. Instead of declaring that our app implements the onClickListener and overriding onClick(), we can call setOnClickListener(this::logSomething), avoiding the implementation keyword.

```
public class MainActivity2 extends AppCompatActivity {

    final static String tag = "==LIFECYCLE_A2==";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);

        Button bview = findViewById(R.id.a2_button);
        bview.setOnClickListener(this::logSomething);

        Intent intent = getIntent();
        String incomingText = intent.getStringExtra("loginText");
        Log.d(tag,"onCreate() - received '" + incomingText);
    }

    public void logSomething(View v) {
        Log.d(tag,"log something button pressed");
    }

    . . .
```

You will also see lots of examples that use anonymous classes. I think the syntax for anonymous classes does not add to the readability of code. I would avoid it.

```
public class MainActivity2 extends AppCompatActivity {

    final static String tag = "==LIFECYCLE_A2==";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);

        Button bview = findViewById(R.id.a2_button);
        bview.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Log.d(tag,"log something button pressed via anonymous class");
            }
        });

        Intent intent = getIntent();
        String incomingText = intent.getStringExtra("loginText");
        Log.d(tag,"onCreate() - received '" + incomingText);
    }

    . . .
```

Anonymous classes protect the privacy of the method minimizing the class interface, but include extra lines of code and can lead to a lot of cut and paste.