Australian National University

Android Development - Part 1

Eric McCreath

Overview

In this lecture we will:

- overview the Android development framework,
- see how to make a simple "hello world" application looking at an Activity, View, resources, and the manifest file,
- look at different widgets that can be used within Android (including Button, TextView, EditText),
- look at layouts,
- how to connect your widgets up to some code, and
- how to track down debugs in your implementation.

Overview of Android Framework

In many respects the framework for implementing GUI application in Android is very similar to Swing or JavaFX. As your GUI code is executed on a single event thread. Also it has a similar collection of widgets, such us buttons, text labels, menus, etc.., that you combine to form your GUI.

However, the Android framework also combines a standard way of incorporating resources such as images and GUI layouts. This provides enormous flexibility, particularly when you wish to create application that will work on a variety of devices and device configurations.

Android Studio

Android Studio is the IDE for the development of android applications. *Android Studio* is build on *Intellij.*

Android Studio uses *gradle* as its build tool. This creates Android Packages (APK) that can be load and run on devices running the Android OS. The APK format is a compressed archive format that contains classes compiled into the *dex* format along with resources and other files for running the application.

The *dex* (Dalvik EXecutable) format is like Java byte code, however, it is register based rather than stack based and is designed to run with limited memory and lower processor speeds. Previously the *dex* code ran on the Dalvik virtual machine, however, this has been superseded by the Android Runtime (ART).

Australian National University

Hello World

Android Studio will start you off with a simple hello world application by default when you create a new project.

This application has 3 key components set up. They are:

- the manifest - this describes the application and points to the main activity for the application.
- the main activity class - this has an "onCreate" method which is executed when the application starts. The "onCreate" method sets the content view based on the layout within the resources.
- the layout for the activities "view" - this includes the "Hello World" text in the center of the view.

Australian National University

Hello World

AndroidManifest.xml

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Hello World

MainActivity.java

```
package com.example.ericm.helloworld;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

Australian National University

Hello World

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.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="com.example.ericm.helloworld.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

AndroidManifest.xml

All Android applications must have a manifest file.   It always has the exact file name "AndroidManifest.xml" and sits in the main directory that holds the "java" and "res" directories.

As there is no "main" method, the application needs to know what to start off running when the application is run.   So this is done via the manifest file.

The manifest also holds information about:

- Application components
- Intent filters
- Permisions
- Icons and labels
- required hardware and software features

Activity

An activity is a single coherent focus of a users interaction with the application. Applications will normally have a number of "activities" which the user moves between. Also a stack of activities is maintained so when you start a new activity the current one is added to the stack so once you have completed interacting with the new one you return back to the activity you left off from.

An activity you create will extent the Activity class and you will always implement the "onCreate" method.

The onCreate method would normally:

- call the super class onCreate,
- recover any state information,
- set the UI layout, and
- obtain references to widget and set up listener.

Australian National University

Resources

Particularly for GUI applications, there is often a large amount of non-code data that the application needs to run. This includes things like layouts, images, sounds, etc...   Android places them in the "res" directory and packs them into APK so the application can use them when running.   Android also provides a neat way that your code can reference and access these resources.

So within your code to get hold of an index of a resource you can reference it via "R.<type>.<name>". Noting this is just an integer index.

Say you had a check box in a layout you wish to reference in your code you first add the id in the CheckBox's xml via **android:id="@+id/checkbox"** then in the code you can:

```
CheckBox cb = (CheckBox) findViewById(R.id.checkbox);
```

Australian National University

View and ViewGroup

A **View** is the basic building block of your GUI. And as the name suggests they are something you see, so they have a "onDraw" method.   All the basic widgets extend the View class including Button, CheckBox, TextView, EditText.   Now if you would like to make your own "View" you can override the "onDraw" method.

A **ViewGroup** , which is also a View, lets you combine View object into a composite widget.   So the standard layouts, such as ConstraintLayout or LinearLayout, extent the ViewGroup class.

Buttton

A **Button** is a widget that provides a way of initiating an action when pressed. To add a button simply add it to your layout:

```
<Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        android:onClick="buttonPress" />
```

With the "onClick" attribute set it is expecting a method with that name given (in this case "buttonPress") in the activity that sets the layout as its content view.   So to your activity you add the the method:

```
public void buttonPress(View v) {
        // do the button's action
}
```

Australian National University

Buttton

You could also add buttons via code in your onCreate method of the activity, or even dynamically at other places in your code.   So the below example adds a button to a linear layout within an activity.

```
        Button b = new Button(this);
        b.setText("Button");
        LinearLayout cl = (LinearLayout) findViewById(R.id.activitymainlayout);
        cl.addView(b);
```

Rather than using the "onClick" attribute within the layouts xml resource you and add a click listener within your code. e.g.

```
       Button b = (Button) findViewById(R.id.button);
       b.setOnClickListener(this);
```

With the class implementing "View.OnClickListener" interface (involves implememting the onClick method).

Australian National University

TextView

A **TextView** is a widget for showing the some text.   They can simply be added to the xml layout.

```
<TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:id="@+id/textView" />
```

If you wish to modify the text then you can reference it via the id and use the setText method.

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

If you wish to change the font size you can add the below to the xml:

```
 android:textSize="28sp"
```

EditText

The **EditText** widget can be used for text entry.

```
<EditText
        android:id="@+id/editText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="text"
        android:text="" />
```

Note that the "inputType" is required. So "text" is for plain text input, however, there is a variety of other options including: number, textPassword, or date. These constrain/alter user input helping obtain correct input.

The EditText is a TextView so the "getText" method can be used to obtain the value of the text the user input.

If you wish to be notified of changes in the text as they happen you can add a "TextWatcher" via the "addTextChangeListener" method.

Layouts

Layouts provide a way of setting out your widgets on your activities screen. Noting that you may use different layouts for different screen sizes or orientations (landscape/portrait). This can be done by adding alternate layout xml files within your resources.

The two commonly used layouts are:

- the **LinearLayout** this provides a simple way of configuring widgets in a horizontal or vertical line.
- the **ConstraintLayout** in which you can describe the relationship between elements to be layed out.

Layouts can be nested so for example you could have a linear layout of buttons that sits within a constraint layout.

LinearLayout

The LinearLayout can be orientated either as *horizontal* or *vertical* . The orientation is given within the xml attributes. To control the alignment of views within this layout the "gravity" attribute can be used. So the below xml is vertically layed out with element aligned to the left.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:gravity="left">

   <!-- add widgets that are elements within this layout -->

 </LinearLayout>
```

ConstraintLayout

The ContraintLayout provides a very flexible and powerful way of setting out widgets within your activities screen. Basically you describe the relationship the widget have with the parent they are contained within along with the relationship they have between each other. This enables you to describible relationships such as: above, below, left of, right of, alignment and margins.

In the example below the Button is placed below the TextView.

```
<android.support.constraint.ConstraintLayout
     :    : >
    <TextView
         :
        app:layout_constraintTop_toTopOf="parent"
        android:id="@+id/textView" />
    <Button
         :
        android:id="@+id/button"
        app:layout_constraintTop_toBottomOf="@+id/textView" />
</android.support.constraint.ConstraintLayout>
```

Debugging

Android studio has a debugger. With it you can set break points, run your app in debug mode (press the run arrow that has a 'bug' on it), step through code execution and examine variables. To set a breakpoint single left click next to the line number of the code you wish the debugger to stop at.

Also a simple way of debugger code is to add log messages at key points in your code. This enables you to check if the code is doing what you are expecting it to do. To add a log message just add:

```
Log.d("tag","message"); // The tag enables you to have a number of
                        // related logs which you can quickly filter.
```

Also when a problem happens an exception or error is thrown. You can use this to track down problems.

Australian National University

Android Development - Part 2

Eric McCreath

Overview

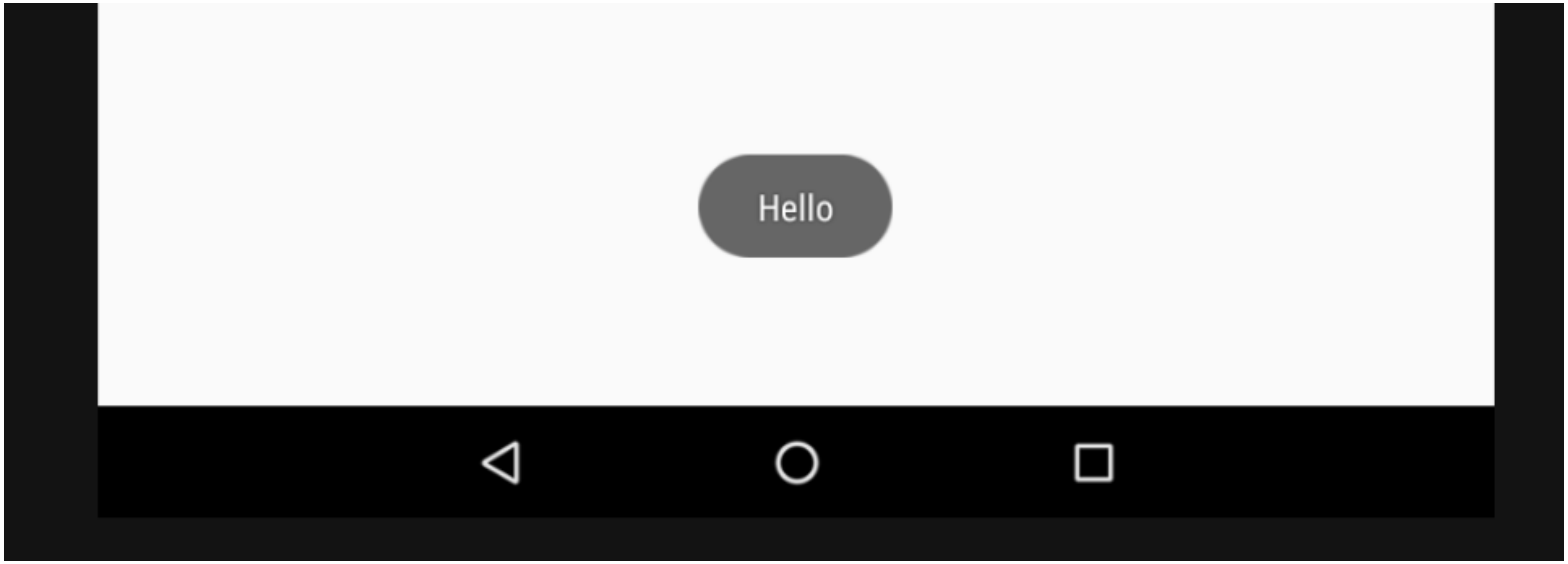
In this lecture we will explore some more advanced features of Android development including:

- toasts
- menus
- checkboxes
- intents

Toasts

Toasts provide a simple way of displaying a short message to the user. The user does not need to respond to the message, if you require a response then a notification should be used. The current activity remains visible and active. To make a toast one can simply add:

```
Toast.makeText(getApplicationContext(), "Hello", Toast.LENGTH_SHORT).show();
```



Australian National University

Menus

Menus provide a simple way of giving options to your user that are less frequently selected then button or other widgets you place in the main view area.   The best way of adding menus is to use an xml resource file to describe the menu structure.   The below example, called my_menu.xml, was added to the "menu" directory within your "res" directory. Noting you will need to create this "menu" directory.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/doit"
        android:title="DoIt" />
    <item
        android:id="@+id/info"
        android:title="Info" />
</menu>
```

Your can also add sub-menus by adding a "menu", with it's own items, within the list of top level items.

Menus

To add the menu to your activity you override the "onCreateOptionsMenu" method within the activities class.   This inflates the xml resource.

```
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.my_menu, menu);
        return true;
    }
```

To listen to menu item selections you override "onOptionsItemSelected" within your activity.

```
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.doit:
                // add code here to deal with "doit" selection
                return true;
            case R.id.info:
                // add code here to deal with "info" selection
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
```

CheckBox

A **CheckBox** provides a way of obtaining boolean information from the user.   Within your layout add:

```
<CheckBox
        android:id="@+id/checkBox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="CheckBox" />
```

Then to determine if the checkbox has been checked you can use:

```
CheckBox cb = (CheckBox) findViewById(R.id.checkBox);
boolean checked = cb.isChecked();
```

If you wish to be notified when the user changes the checkbox you can add

```
 android:onClick="onCheckboxClicked"
```

to the checkbox entry in the layout xml file.   Then just add the "void onCheckboxClicked(View v)" method to your activity.

Australian National University

Intent

An **Intent** is used to start another activity, request a service, or broadcast a message. To start a different activity (in this example called "SubActivity") you can:

```
        Intent intent = new Intent(this, SubActivity.class);
        startActivity(intent);
```

So the Intent is a message that travels across to the new activity. You can attach a key-value pairs to the intent to send some information to the activity you are calling. e.g.

```
        intent.putExtra("KEY","the data to send");
```

In the receiving activity you can extract the information from within the message via:

```
        Intent intent = getIntent();
        String info = intent.getStringExtra("KEY");
```

Android Development - Part 3

Eric McCreath

Overview

In this lecture we will explore some more advanced features of Android development including:

- extending the View class,
- painting to a canvas,
- touch listeners, and
- a Handler for timer events.

Extending the View class

Although there is a large range of standard widgets that you can use, sometime you wish to create your own widget. This can be done by extending the View class.   This is particular useful if you wish to have an region in your application that you can draw to and obtain user touch input from.

Extending the View class

To get this working add a new class to your project that extends the View class. This class should have a constructor with a context and attributes and it should call the "super" constructor. The most common method you would override is "onDraw", this enables you to customize the drawing. To cause your view to be redrawn call the "invalidate()" method when required.

```java
public class MyView extends View {

    public MyView(Context c, AttributeSet as) {
        super(c, as);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        Paint p = new Paint();
        p.setColor(Color.BLUE);
        p.setStrokeWidth(5.0f);
        canvas.drawLine(0.0f,0.0f,100.0f,100.0f,p);  // draw a thick blue line
    }
}
```

Extending the View class

Now in your layout you add your custom view. Note the full package details are required.

```
<anu.ericm.andoidp3.MyView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/myview"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```

I have also added an "id" this enables me to get hold the "MyView" object within my code using:

```
     MyView mv = (MyView) findViewById(R.id.myview);
```

Australian National University

Extending the View class

When your class is constructed you don't know its size. So the "onMeasure" method is called to tell your custom view the constraints on its dimensions. Within this method you call the "setMeasuredDimension" to tell the parent your custom view's dimensions (they should be within the provided width and height specifications).

```
    @Override
    protected void onMeasure(int wspec, int hspec) {
        Log.d("MyView", "w:" + MeasureSpec.toString(wspec) +
                       " h:" + MeasureSpec.toString(hspec));
        if (!(MeasureSpec.getMode(wspec) == MeasureSpec.AT_MOST &&
              MeasureSpec.getMode(hspec) == MeasureSpec.AT_MOST))
            throw new AssertionError();
        setMeasuredDimension(MeasureSpec.getSize(wspec),MeasureSpec.getSize(hspec))
    }
```

The width and height specification are encoded integers that you can use the static methods of "MeasureSpec" to pull apart. Key methods are "getMode" and "getSize", the mode is one of UNSPECIFIED, EXACTLY, or AT_MOST.

Canvas

The **Canvas** class has the methods for drawing to a 2D bitmap image. So if you wish to draw to a BitMap you would create a Canvas that is attached to the BitMap. Also a Canvas is provided in the onDraw method you override for a custom view, you use this for drawing to your custom view. The canvas class includes:

```
drawLine(float x1, float y1, float x2, float y2, Paint p)
drawText(String text, float x, float y, Paint p)
drawRect(float x1, float y1, float x2, float y2, Paint p)
drawCircle(float cx, float cy, float r, Paint p)
drawBitMap(Bitmap bm, float left, float top, Paint p)
drawColor(int color)
```

You can get the height and width of the canvas via getHeight() and gitWidth() respectively.

Australian National University

Paint

The paint class provides a way of describing the attributes of a "pen". So you can set up a "pen" to be thick and blue and another one to be thin and red. So as you draw to a canvas you can alternate between your two "pens". The paint class also captures fonts characteristic for when you are drawing text. Key methods include:

```
    setColor(int color)    - set the painting color
    setStrokeWidth(float width) - set the width of the stoke
    setTextSize(float textSize) - set text font size
    setTypeface(Typeface typeface) - set a different font
```

Say I wanted to set up a paint object that draws thick blue lines and uses a large sans serif font I could:

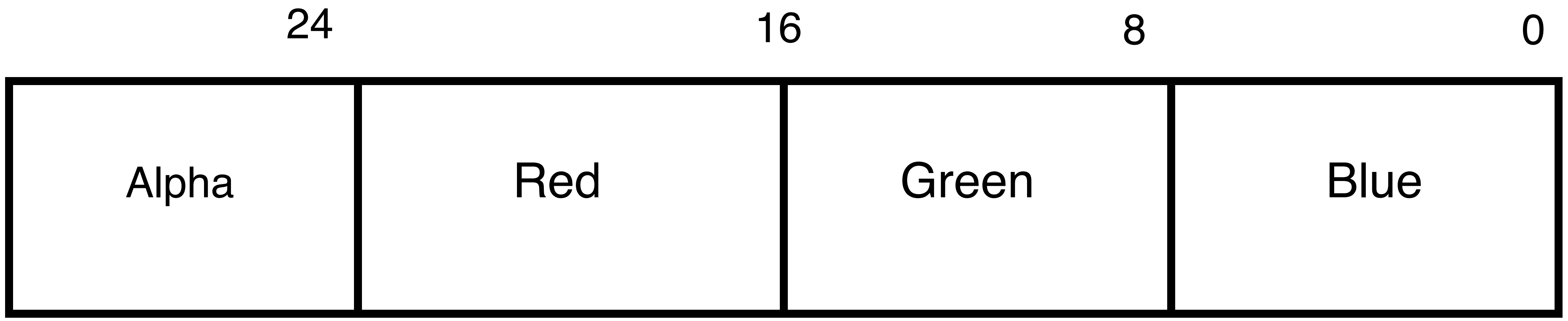
```
        Paint p = new Paint();
        p.setColor(Color.BLUE);
        p.setStrokeWidth(5.0f);
        p.setTextSize(50.0f);
        p.setTypeface(Typeface.create(Typeface.SANS_SERIF, Typeface.BOLD));
```

Australian National University

Color

Colour may be represented by an integer.   So a 32 bit integer is partitioned into 4 bytes and these correspond to the red, green, blue, and alpha channels.

```
 (alpha & 0xff) << 24 | (red & 0xff) << 16 | (green & 0xff) << 8 | (blue & 0xff)
```



You can directly manipulate these bytes, however, it is often easier to just use the static methods in the Color class to create a colour and extra individual channels from the colour.

```
int darkGray = Color.rgb(60,60,60);
int lightBlue = Color.rgb(150,150,255);
int bluePartOfDarkGray = Color.blue(darkgray)
```

Australian National University

Color

The channel values range from 0 to 255, and in RGB they correspond to the amount of colour of that channel that is added to form the final color.   So:

- Color.rgb(255,0,0) is a bright red,
- Color.rgb(0,0,0) is black, and
- Color.rgb(255,255,255) is white.

The Color class also provides a set of common colours which are just integer constants including: Color.RED, Color.BLACK, Color.BLUE, ...

Sometimes you wish to use a float ranging from 0.0 to 1.0 to express the amount of a particular channel. From API level 26 the Color class also provides a static method for making colours from floats, so Color.rgb(0.0f,0.0f,1.0f) would be blue.

onTouchEvent

To get touch events from the user you can add a "boolean onTouchEvent(MotionEvent me)" method to your custom view class. If the event was handled correctly it should return true. The MotionEvent holds information about touch event, including the x and y location of the event. Noting it also holds a history of previous touch events that can be used for more complex gestures. So the below code captures the touch events and stores the position of the touch in the xpos and ypos fields. These fields can be used in the onDraw method.

```
    @Override
    public boolean onTouchEvent(MotionEvent me) {
        xpos = me.getX();
        ypos = me.getY();
        invalidate();  // force the view to be redrawn
        return true;
    }
```

Handler

To get timing events so you can update and redraw your view at regular time steps you can use a **Handler** .   So the "postDelayed" method of a Handler is given a Runnable object along with a delay time (in ms), once the delay has elapsed the run method is executed.   So in our example we:

```
public class MyView extends View implements Runnable {
   Handler timer;
```

Add to the constructor:

```
   timer = new Handler();
   timer.postDelayed(this,10);
```

Add the run method:

```
    @Override
    public void run() {
         xpos+=1.0; // update the state of the simulation
         this.invalidate(); // cause the view to be redrawn
         timer.postDelayed(this,10);  // restart the timer
    }
```

Bitmap

A **Bitmap** provides a way of holding image pixel data. From it you can read and set the color of individual pixels. Also you can use a Canvas to draw to the Bitmap.

If you wish to create your own muttable Bitmap you can (in this case 300 pixels wide and 200 high):

```
bm = Bitmap.createBitmap(300,200, Bitmap.Config.ARGB_8888);
```

To make a canvas so you can draw to this Bitmap you can:

```
Canvas can = new Canvas(bm);
```

You can also add png images to your drawable resource directory and then get hold of these resources via (in this case I add the supertux.png image):

```
tuxImage  = BitmapFactory.decodeResource(getResources(),R.drawable.supertux);
```

Facade Design Pattern

Eric McCreath

Overview

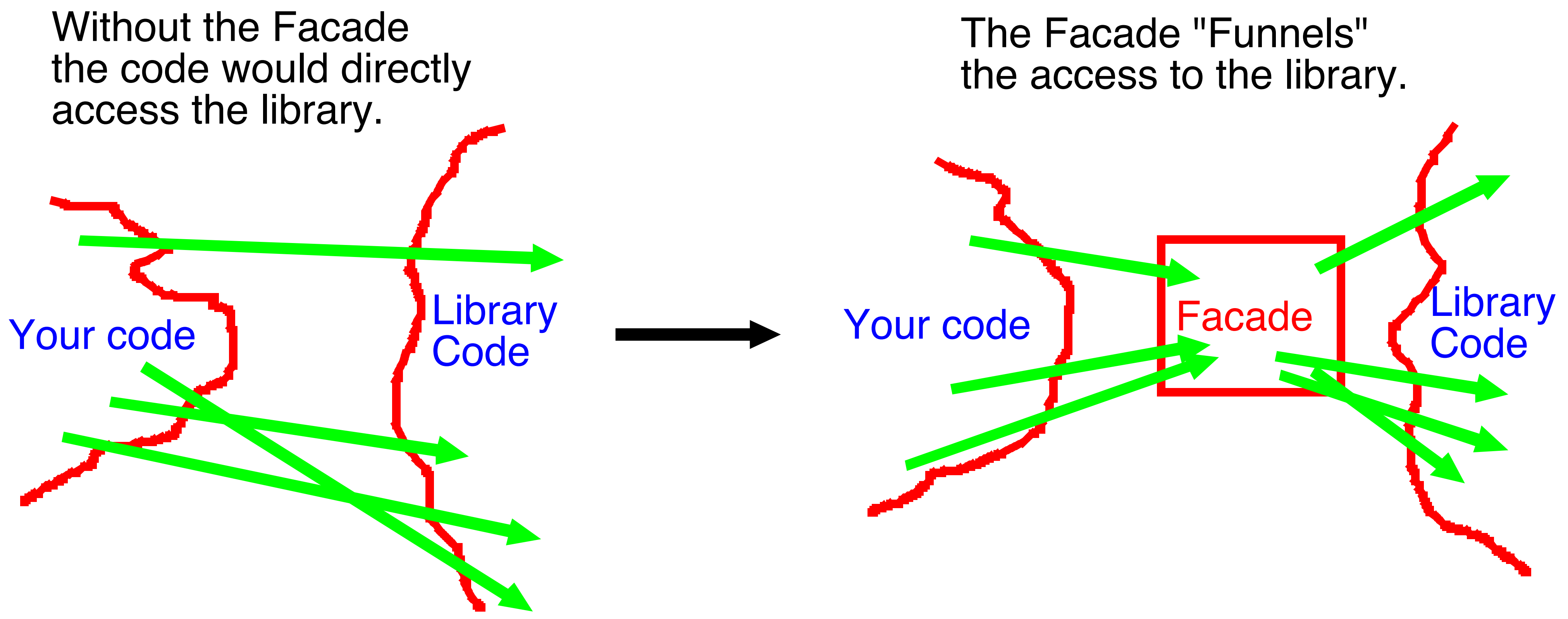
In this lecture we will:

- define the facade design pattern,
- review a simple example,
- discuss when the facade is useful, and
- provide some examples of using this design pattern.

Australian National University

Definition

The facade design pattern uses a class to provide a simplified Application Programming Interface (API) to a complex module (such as a collection of classes or a library).


Without the Facade the code would directly access the library.
Your code
Library Code
The Facade "Funnels" the access to the library.
Your code
Facade
Library Code

Simple Example

Say a program you are developing wishes to load and save small text files. This could be achieved by directly accessing the standard java.io package. However, java.io is large and somewhat complex, thus one may create a **Facade** class for this loading and saving of a file. Part of this code is shown below (the full listing is available in the example code repo).

```
public class LoadSaveFacade {

    static String load(String filename) {
                       :    :
    }

    static boolean save(String filename, String text) {
                       :    :
    }
}
```

So now the rest of your code will use the "load" and "save" methods, rather than the java.io package.

Advantages and Disadvantages

Advantages of the facade design pattern include:

- simplifies the use of a complex library/collection of classes,
- concentrates the code for that library/collection of classes into one point in your code, and
- reduces the coupling between your code and the library/collection of classes.

Disadvantages include:

- adds a layer of indirection which may affect performance,
- may make your code base bigger, and
- your developers will need to learn to use this bespoke API (whereas they may be very familiar with the library).

Australian National University

Other Examples

There are many situations this design pattern may be used. These include;

- access to a database,
- access to io devices (e.g. webcam, sound card),
- file io,
- using visualization libraries,
- network communication,
- using machine learning libraries,
- using computer vision libraries ......

Example Exam Questions

- Explain the Facade design pattern. Illustrate this with an example of when using the Facade design pattern would be useful.
- The "PassTheMessage" project available at: git@gitlab.cecs.anu.edu.au:u4033585/passthemessage.git

  has 4 classes. Would you describe any of them as being a "Facade"? If yes, explain why. If not, explain why not.
- The java.lang.Boolean class provides a simple class for storing booleans. It also provides some basic methods for operating on booleans. Would it be a good idea to construct a Facade class for the Boolean class? Explain.

Observer Design Pattern

Eric McCreath

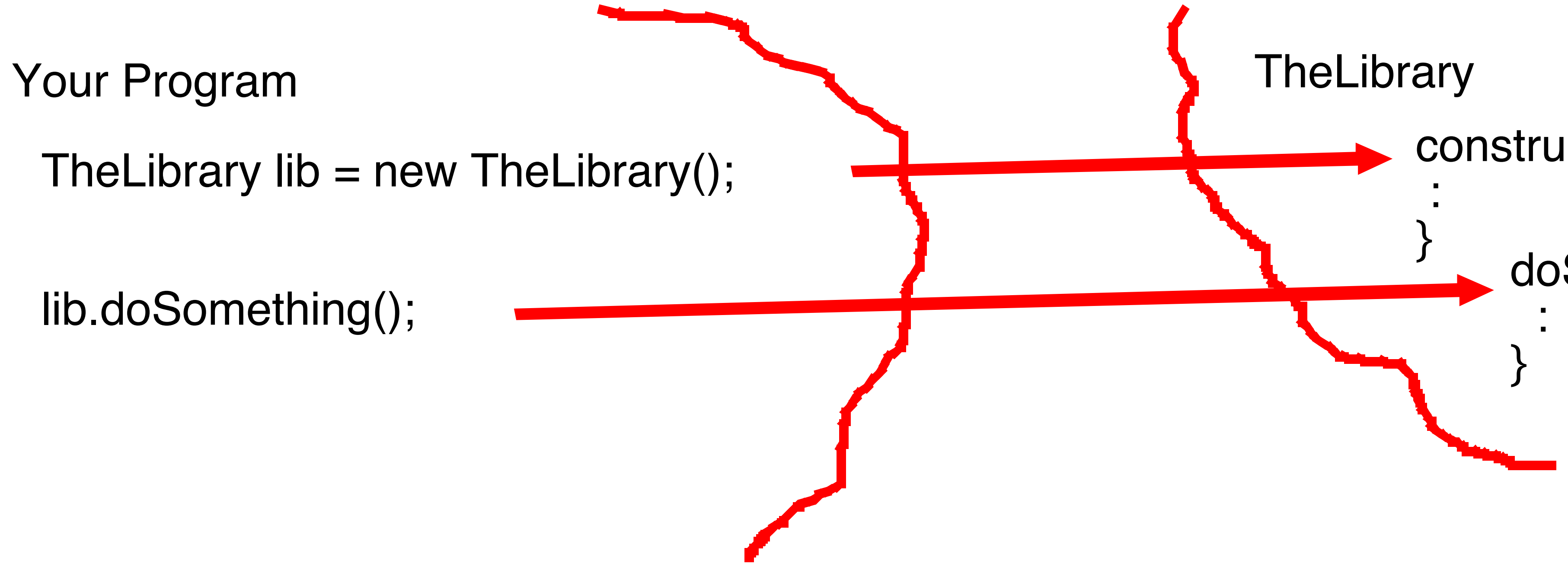
Overview

In this lecture we will:

- motivate and define the Observer design pattern,
- give some examples of where this design pattern is used,
- provide a template for implementing your own classes that use this pattern, and
- in the following lecture a "live code" example.

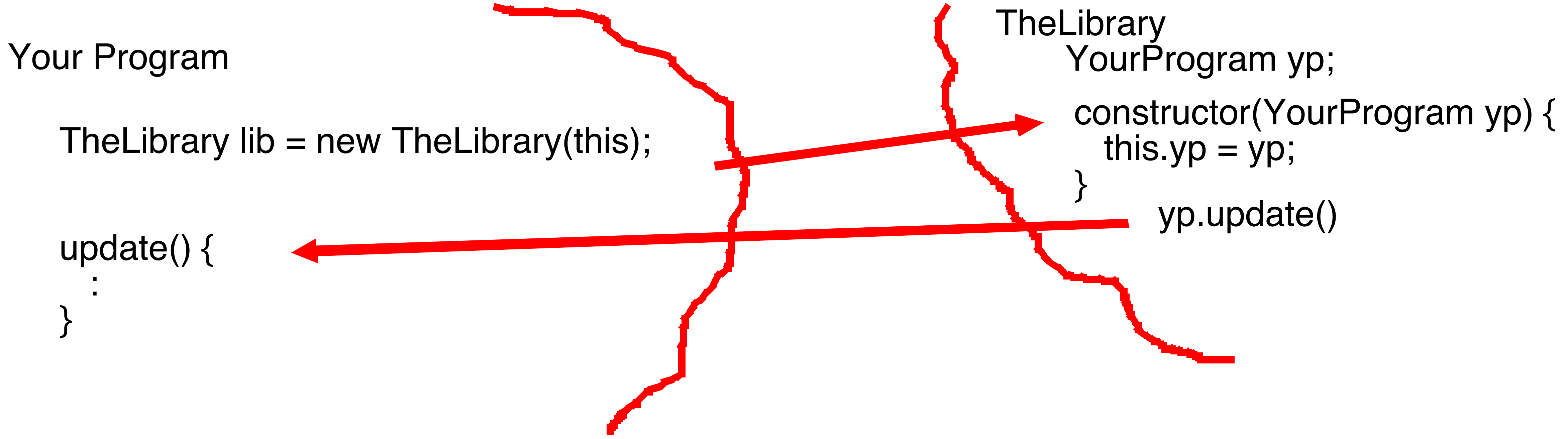
The Context

- Normally when your program uses a library (or another class) it constructs an object that references that library. To then get the library to do something you call methods with respect to this object.
- The great thing about this approach is the library is not coupled to your program and it can be reused for other programs.



The Context

- In some cases you want the library to inform your program of an event that occured in the library.
- Now we could provide the library a reference to your program and then when the event occurs in the library it simply calls a method in your program. The down side of this approach is that the library is now tightly coupled with your program (it only works with your program and you would need to modify the Library to use it in another situation).

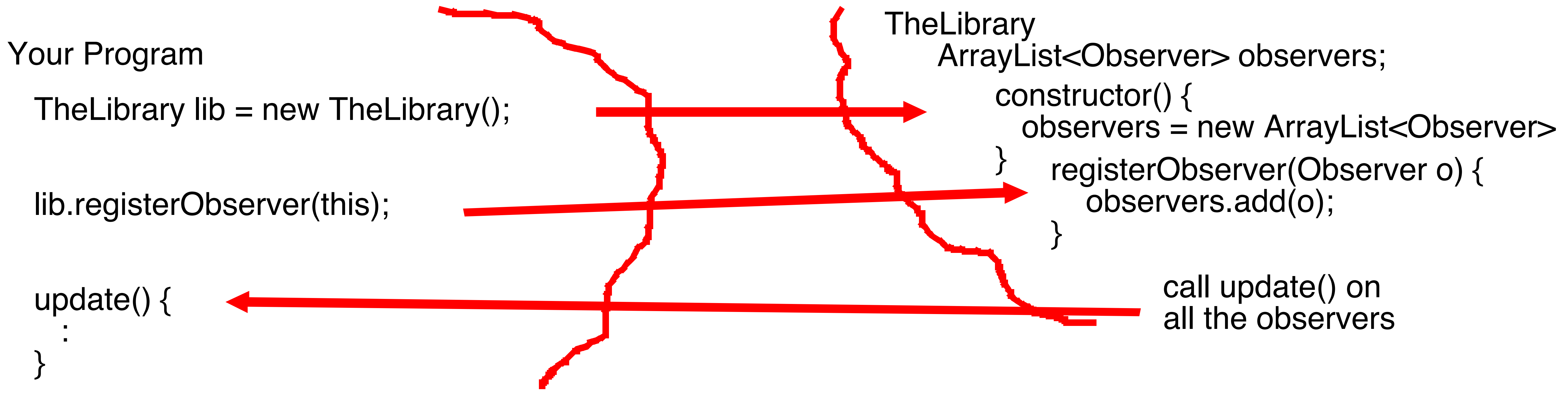


```
Your Program

    TheLibrary lib = new TheLibrary(this);


    update() {
      :
    }
```

```
TheLibrary
      YourProgram yp;

   constructor(YourProgram yp) {
      this.yp = yp;
   }
         yp.update()
```

The Observer Design Pattern

- The observer design pattern has the code using the library, register observers with the library. When the library wishes to notify the code using the library of an event, it simply calls the notification method in all the registered observers (in the diagram below it is the update method).
- The advantage is this library is no longer tightly coupled to your program. Also different parts of your program can register a number of observers which can all be notified of an event.



Your Program

```
TheLibrary lib = new TheLibrary();

lib.registerObserver(this);

update() {
   :
}
```

TheLibrary

```
ArrayList<Observer> observers;
    constructor() {
      observers = new ArrayList<Observer>
    }
      registerObserver(Observer o) {
         observers.add(o);
      }
```

call update() on
all the observers

The Observer Design Pattern

- Normally you would also have a way of de-registering observers that no longer wish to be informed of an "update".
- java.util has a standard Observer interface and a class that is called Observable which may be used. Also you can just create your own interface and observable class.
- We also see this design pattern in langauges which are not object-oriented. The approach is often described as "call-back" where a function or reference to a function is provided to a library via a function call. The provided function or function reference is later "called-back" from the library. Once again the library can initiate the program's code without being coupled to the program.

Australian National University

Examples of Use

We see this design pattern used in situations like:

- GUI libraries (like Swing, Android),
- network IO,
- parsing, and
- error handling.

Code Template

We need an interface for the observers:

```
public interface MyObserver {
   void update();
}
```

The class that receives the events must implement the MyObserver interface:

```
public MyProgram implements MyObserver {
   :
public void update() {
    // do what is required when informed of an event.
}
   :
```

Your program code would also construct an instance of the library:

```
lib = new TheLibrary();
```

and then register itself as an observer of this library:

```
lib.registerObserver(this);
```

Code Template

The library code would keep track of a list of all the observers:

```
ArrayList<MyObserver> observers;
```

This would need to be initialized in the library's constructor:

```
observers = new ArrayList<MyObserver>();
```

Obersevers would be registered via:

```
public void registerObserver(MyObserver o) {
     observers.add(o);
}
```

and when the library wishes to inform all the observers of an event it would call a notifyObservers method:

```
private void notifyObservers() {
   for (MyObserver o : observers) o.update();
}
```

Example Exam Questions

- Explain the observer design pattern. What key design issues does this pattern address?
- Suppose you are implementing a library that uses the observer design pattern to inform the users of the library of an event that has happened. However only part of this code has been completed in the library. Fill in the "....." to complete this implementation.

```
class TheLibrary {
   ArrayList<MyObserver> observers;
    public TheLibrary() {   observers = ..... ;   }
    private void informObserversOfEvent() {
         .....
    }
    public void registerObserver(MyObserver o) {
        .....
    }
```

We also have:

```
public interface MyObserver {   void update(); }
```