CONTENTS

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- Create an Android application.
- Create an activity and manage its lifecycle.
- Access resources programmatically to optimize maintenance and localization.
- Create full-featured graphical user interfaces with widgets, dialogs, menus, and event handlers.
- Control the organization of your screen with layouts.
- Design flexible user interfaces for multiple form factors using fragments.
- Communicate between applications in a loosely-coupled way with intents.
- Listen for notifications with broadcast receivers.
- Create a service to handle background tasks and use notifications to indicate the status of those tasks.
- Persist application state using preferences, files, and SQLLite.
- Use both **ArrayAdapter** and **CursorAdapter** to bind data to widgets.
- Share data with other applications with content providers.
- Use the **LoaderManager** and **CursorLoader**s to display provider data.
- Access **Contacts** and other built-in Android providers.
- Determine the current location and leverage Google Maps.
- Publish an application to an Android device.

COURSE OVERVIEW

- **Audience**: Programmers who are new to developing applications for the Android platform.

- **Prerequisites**: *Introduction to Java* or equivalent experience is required. Basic understanding of XML is required.

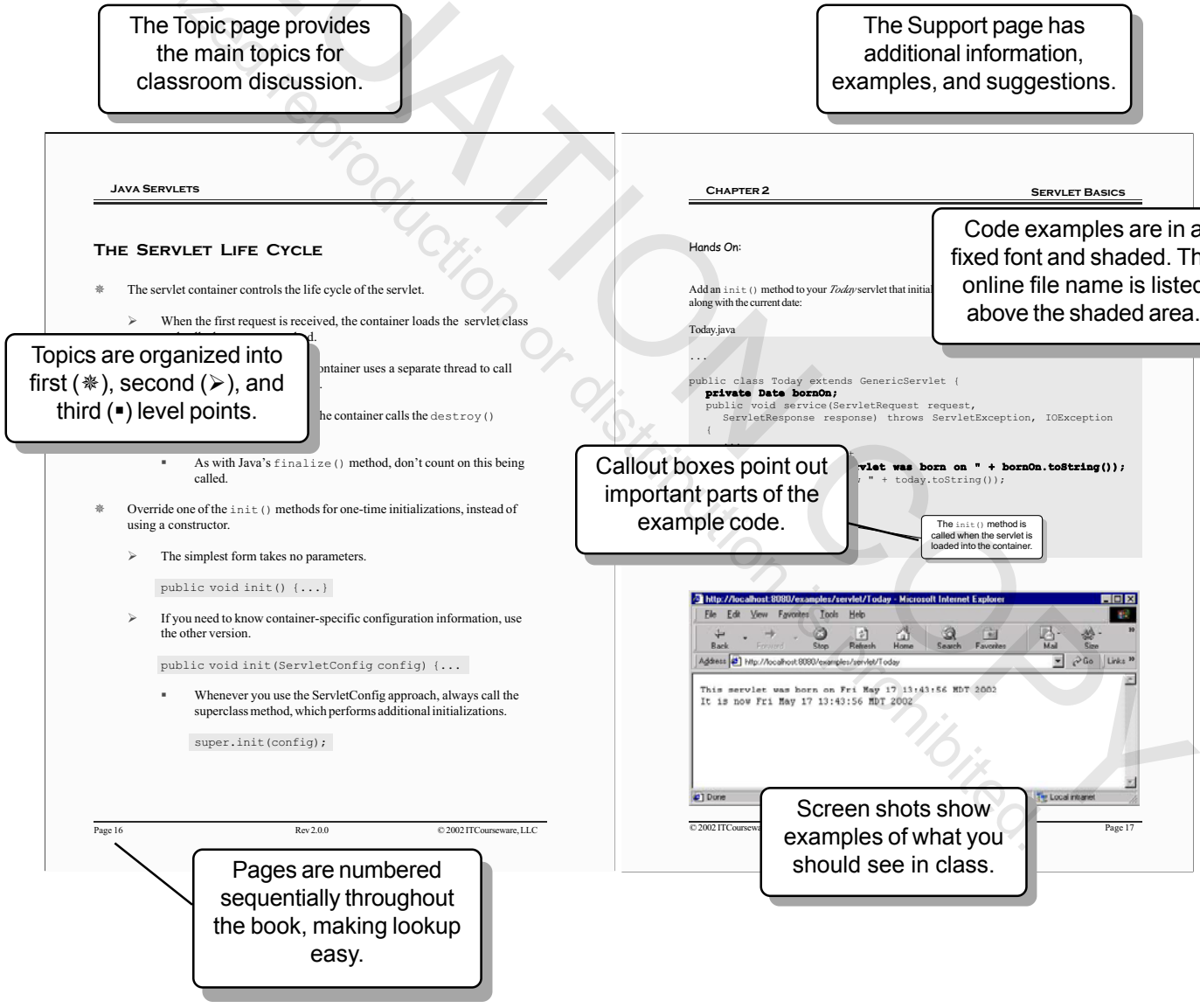
- **Classroom Environment:**

  - A workstation per student with Java 6 and the Android Developer Tools (ADT) Bundle installed.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. *Hands On* sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick lookup. Printed lab solutions are in the back of the book as well as online if you need a little help.



SUGGESTED REFERENCES

Darcey, Lauren and Shane Condor. 2012. ***Android Wireless Application Development Volume I: Android Essentials (3rd Edition).*** Addison-Wesley. Upper Saddle River, NJ. ISBN 978-0321813831.

Darcey, Lauren and Shane Condor. 2012. ***Android Wireless Application Development Volume II: Advanced Topics (3rd Edition).*** Addison-Wesley. Upper Saddle River, NJ. ISBN 978-0321813848.

Lee, Wei-Meng. 2012. ***Beginning Android 4 Application Development.*** Wiley Publishing, Inc., Indianapolis, IN. ISBN 978-1118199541.

Meier, Reto. 2012. ***Professional Android 4 Application Development (Wrox Professional Guides).*** Wiley Publishing, Inc., Indianapolis, IN. ISBN 978-1118102275.

http://developer.android.com/
http://www.stackoverflow.com/

CHAPTER 2 - GETTING STARTED

OBJECTIVES

- Describe the Android architecture and application development model.
- Install and configure an Android application development environment.
- Develop and deploy a simple Android application.

ANDROID OVERVIEW

- Android is an open-source mobile operating system that is maintained by the Open Handset Alliance (OHA).
  - The OHA consortium is led by Google and is made up of member corporations from industries such as handset manufacturers, chip makers, and mobile carriers
  - Android is licensed under the Apache v2 Software License, so there is no cost to use the platform or to develop software for it.
- In addition to the base operating system, Android provides middleware and applications to support core mobile functionality such as camera, phone, and messaging.
  - Your Android application can interact and integrate with these core applications as well as third-party applications to provide a more complete solution to the user, but keep your code more focused.
    - You can, for example, provide a share button in your application, that when clicked, gives the user the choice of whether to share the information via social media, email, messaging, or something else.
    - You simply specify that you intend to share the message and the chosen application does the rest.
    - This loose coupling is fundamental to Android application development.
- Each major Android update is named after a dessert whose first letter increments with each release.
  - Your application should support the earliest version possible to be compatible with devices that have yet to upgrade to a more recent release.
  - This course is focused on Android 4.2 – Jelly Bean.

Visit the Open Handset Alliance website at ***http://www.openhandsetalliance.com***

Access the Android source code at ***http://source.android.com***

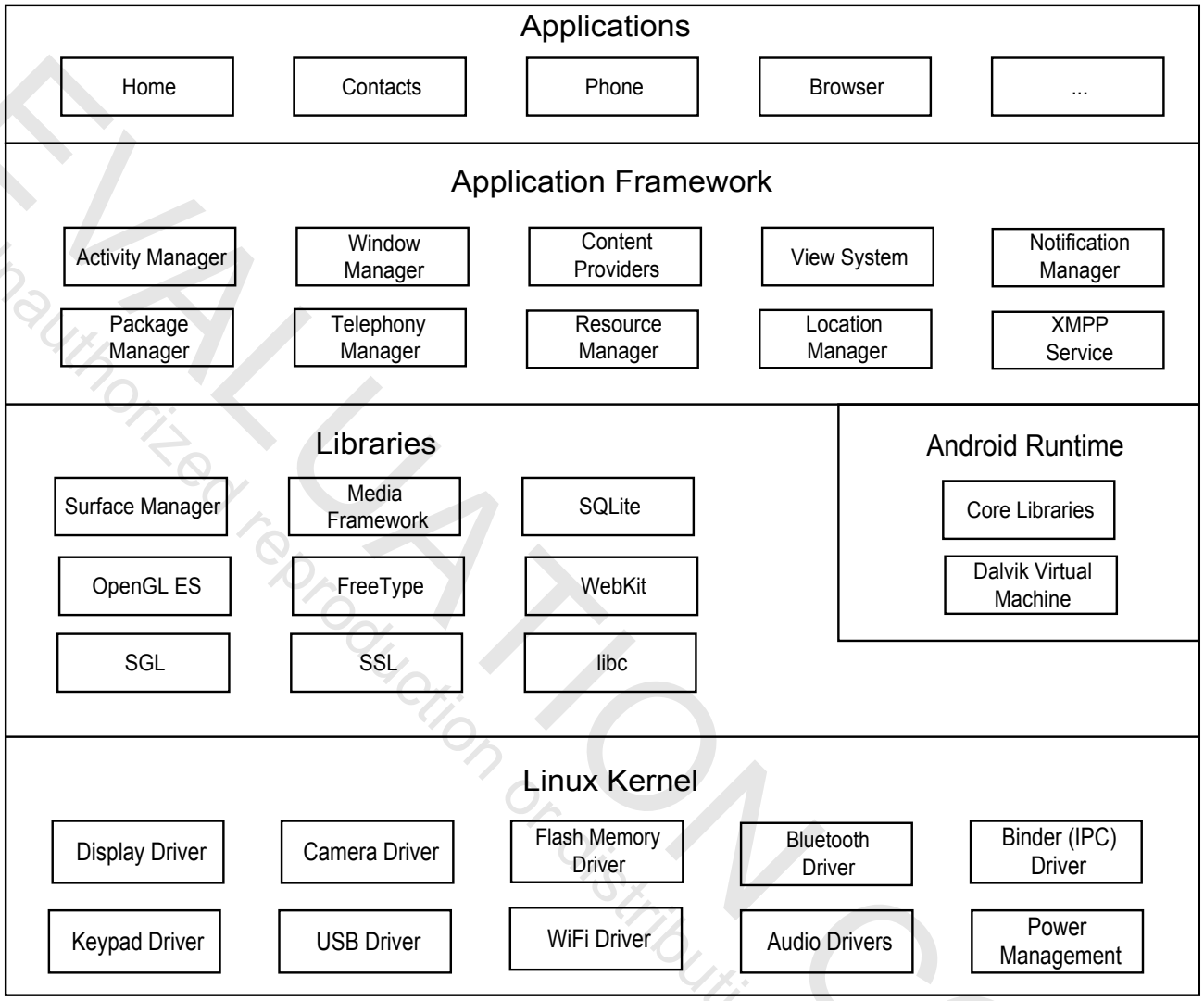
Android enjoyed explosive market growth through versions 2.2 (Froyo) and 2.3 (Gingerbread). Version 3 (Honeycomb) was a separate branch of the Android codebase intended only for tablet computers; it did not see wide distribution but was instrumental in migrating the Android UI to the tablet format. Version 4 (Ice Cream Sandwich) reintegrated Android into a single codebase for all devices.

Android version history:

Version	API Level	Version Name	Release Date
1.0	1	**Base**	9/23/2008
1.1	2	(Base_1_1)	2/9/2009
1.5	3	**Cupcake**	4/30/2009
1.6	4	**Donut**	9/15/2009
2.0	5	**Eclair**	10/26/2009
2.0.1	6	(Eclair_0_1)	12/3/2009
2.1.x	7	(Maintenance Release 1)	1/12/2010
2.2.x	8	**Froyo**	5/20/2010
2.3, 2.3.1, 2.3.2	9	**Gingerbread**	12/6/2010
2.3.3+	10	(Maintenance Release 1)	2/9/2011
3.0	11	**Honeycomb**	2/22/2011
3.1	12	(Maintenance Release 1)	5/10/2011
3.2.x	13	(Maintenance Release 2)	9/20/2011
4.0, 4.0.1, 4.0.2	14	**Ice Cream Sandwich**	10/19/2011
4.0.3, 4.0.4	15	(Maintenance Release 1)	3/29/2012
4.1, 4.1.1, 4.1.2	16	**Jelly Bean**	6/27/2012
4.2, 4.2.1, 4.2.2	17	(Maintenance Release 1)	11/13/2012
4.3	18	(Maintenance Release 2)	7/24/2013

ANDROID ARCHITECTURE

- The Android architecture is made up of five layers.
  - *Applications*, written in Java, form the highest layer.
    - This layer contains both the core applications, such as the calendar, as well as applications you add to your device.
  - The *Application Framework* consists of services that allow applications to interact with the Android environment.
    - Your applications can access the same framework services that the core applications use.
    - Use the Notification Manager, for example, to add messages to the status bar.
  - The *Android Runtime* is made up of the Core Libraries as well as the Dalvik Virtual Machine (VM).
    - The *Core Libraries* contain a subset of the Java API libraries plus Android specific packages.
    - Your Android applications run on the *Dalvik VM* which has been designed and optimized for embedded use.
  - The *Libraries* provide foundational capabilities such as support for audio, video, and graphics.
  - The *Linux 3.* Kernel* provides system services like memory and process management as well as drivers to interact with the underlying hardware.



Adapted from Android Developers Dev Guide, "What is Android", http://developer.android.com/guide/basics/what-is-android.html (accessed June 4, 2012)

The Core Libraries include a subset of the Java SE libraries. Notably missing are the AWT, Swing, and RMI packages. Android also supplements the remaining Java SE libraries with dozens of Android specific packages.

THE DALVIK VM

- You write your Android applications in the Java language and execute them within the Dalvik Virtual Machine.
  - Dalvik does not read Java *.class* files, but rather *.dex* (Dalvik Executable) files.
    - The **dx** tool that comes with the Android SDK translates Java byte code into optimized *.dex* files.
  - Dalvik is designed so that multiple instances of the VM can efficiently run simultaneously.
    - A Just In Time (JIT) compiler was added to the Dalvik VM in the Froyo release to further enhance performance.
  - Dalvik relies on the underlying kernel to handle memory management, process isolation, and threading.
- By default, each of your Android applications runs within its own process with a unique user and group ID, providing a secure sandbox.
  - Each process runs its own VM to isolate one application from another.
  - The permissions of an application's files are set so that only the process itself can access them.
    - You can configure an application to request permission to access additional data such as the camera or the user's contacts.

Dalvik was written by Dan Bornstein who named it after Dalvik, Iceland where his ancestors once lived.

An Android application can choose to share data with other applications with a service or by acting as a content provider.

Each Android application can describe what permissions it requires in order to execute by adding zero or more **<uses-permissions>** entries to the application's manifest file. When a user installs an application they will be prompted as to whether they allow the application to access the resources listed within the **<uses-permissions>** tags.

ANDROID COMPONENTS

- Android defines four components that serve as building blocks for application development.
  - An *activity* is a screen displaying user interface controls and/or graphical content.
    - Most applications are made up of multiple independent activities.
  - A *service* handles long-running background tasks, without displaying a user interface.
    - Services can be either started and run until completion, or bound to an activity to take part in inter-process communications.
  - *Broadcast receivers* are notified of system events such as battery low or device shutdown.
    - You can also create a broadcast receiver to listen for your own custom events.
  - A *content provider* manages data that other applications can query and update.
    - The data is typically stored in the file system or in a SQLite database.

While most Android applications use only one or two of the four components, there may be scenarios where you can make use of all four. For example, if you were creating a weather application, you may use the following components:

Activity – You can use an activity to create a screen that displays current weather data for a particular city and another screen to allow the user to select the city to display weather data for.

Service – You can create a service as a background task that downloads weather information from a server on the Internet.

Broadcast Receiver – You can use a broadcast receiver to react to temperature change events and update the status bar with the current temperature. Perhaps the service could broadcast the event in the first place.

Content Provider – You can create a content provider to expose historical weather data to other applications

ANDROID INSTALLATION

- To develop Android applications you should have:
  - A Java Software Development Kit (JDK), version 6.
  - The Eclipse IDE with the Android Developer Tools (ADT) plugin.
  - The Android Software Development Kit (**Android SDK**).
  - The **Android SDK Platform-tools**.
  - At least one version of the **Android Platform**.
    - You should have a platform for each version of Android for which you are actively developing and supporting your app.
- While you can download each component separately, it is much easier to download the ADT bundle from ***developer.android.com***.
  - Other than the JDK, the ADT bundle comes with everything you need to develop Android applications including, the most recent Android platform.
- Installing the ADT bundle simply involves unzipping the downloaded file to the location of your choice.
  - By default, the unzipped content will be located in a directory whose name starts with *adt-bundle-*.
  - You may be adding components and performing updates to the SDK, so unzip it where you have full permissions.
- For more about the Android SDK including downloads and installation instructions, see: ***http://developer.android.com/sdk/index.html***.

When you unzip the ADT bundle, you will find a directory called */sdk/* which contains the following:

/tools/
This directory contains a number of tools for testing and debugging your Android apps, including:
* **AVD Manager** - graphical tool for defining, managing, and launching specific Android device emulators based on platform versions and even on specific hardware devices.
* **android** - general-purpose command-line tool for creating android projects (based on templates supplied with the SDK,) and interacting with other SDK components.

```
android create project --name MyProject --target android-11 `
   --path C:\Users\student\MyProject --package com.example.myapp `
   --activity MyMainActivity
```

* **emulator** - command-line tool for launching an Android Virtual Device (emulator).

```
emulator -avd MyJellyBean
```

* **mksdcard** - create a file image of a blank SD card for use by an emulator.
* **DDMS** (Dalvik Debug Monitoring Server) - A sophisticated framework allowing access to a connected Android device or emulator, supporting debugging, file-system browsing, screen-shot capture, low-level process and memory information, and many other important capabilities.
* **monkeyrunner** - tool for running test scripts (sequences of user-generated events) against an app running on an emulator.
* **sqlite3** - command-line access to SQLite databases.
* *ant/* - build-configuration file (*build.xml*) for compiling and packaging an app using the open-source **ant** build tool.
* Other tools for tracing, viewing, and dumping application structure, execution, and logs, working with image files for Android apps, etc.

/platform-tools/
The ***Platform-tools*** install into this directory, and include:
* **adb** (Android Debug Bridge) - provides command-line access to a connected device or running emulator, allowing filesystem navigation, access to SQLite database files (via **sqlite3**), commands for pushing files onto and pulling files off of the device filesytem, installation of apps to the device, and many other activities; from the command-line, **adb** is your primary interface to the Android system.

/build-tools/
This directory includes:
* **dx** - compiles Java byte-code into Dalvik executable (.dex) format.
* **aapt** (Android Asset Packaging Tool) - produces an *R.java* file used to lookup resources in code.

/platforms/
Android ***Platforms*** install into this directory. An Android Platform includes all the components and resources used by an Android device or emulator for a specific Android version:
* Fonts, device skin images, stock resource definitions for strings, layouts, etc., user-interface and other templates, build properties, and so on.
* *android.jar* - the compiled classes that make up the Android java platform.

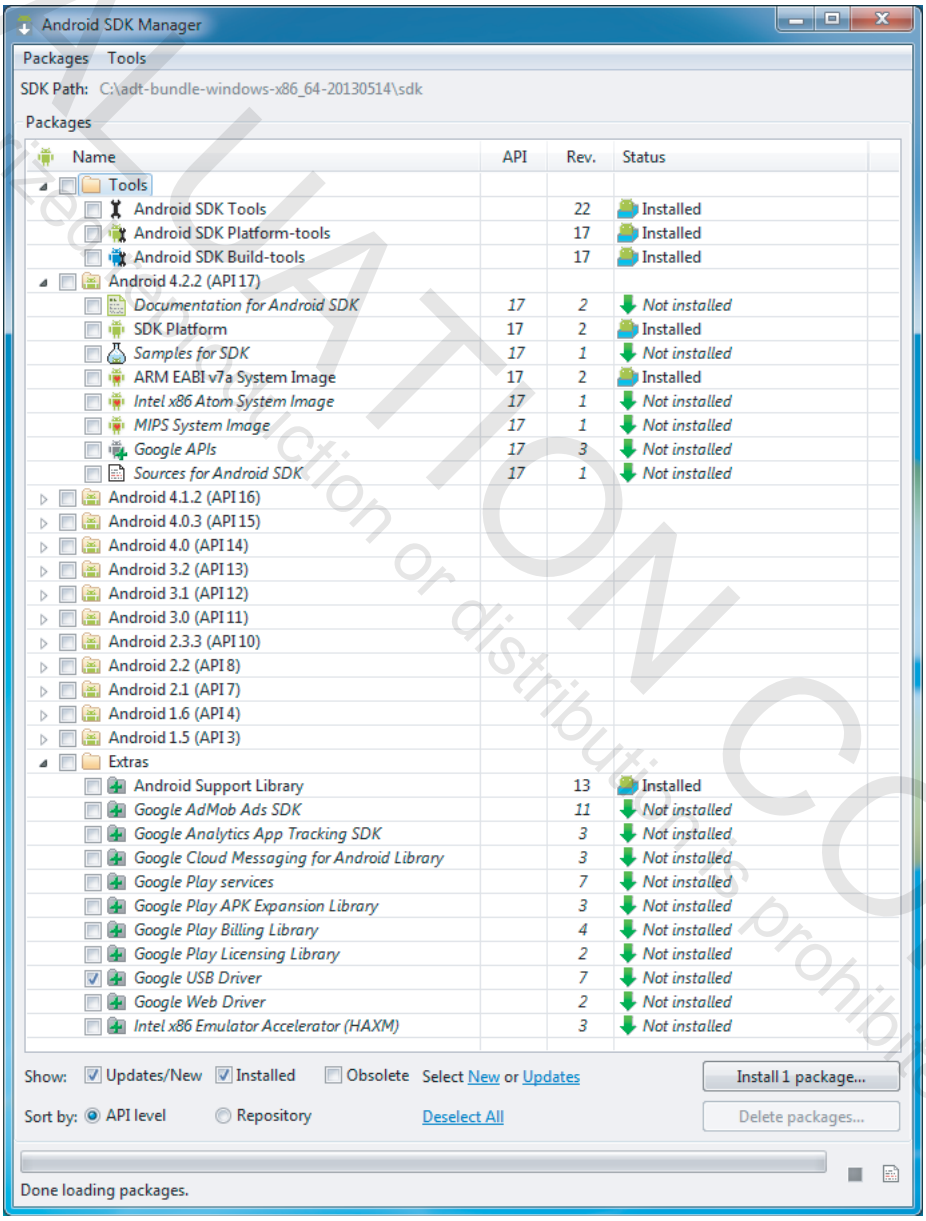
SDK Platform-tools and the SDK Manager

- The Android SDK Platform-tools install under the SDK installation directory, but are maintained and versioned separately.
  - Each new version of the SDK typically has a corresponding Platform-tools update.
  - Platforms also install under the SDK installation directory.
- From the ADT bundle directory, you can run the **SDK Manager**.
  - The **SDK Manager** provides a graphical user interface in which you can see what is currently installed in your SDK.
  - You can also choose to install additional packages by selecting the checkbox next to the item and then clicking on the **Install ... Packages** button.

In addition to the currently installed packages, you may want to install:

* **Google USB Driver** - allows you to connect an Android device to your Windows system's USB port and use it in development and testing of your apps.

* **Android Support Library** - provides additional APIs that aren't part of the standard Android framework, and compatibility libraries to support the latest APIs on older Android platforms.



ECLIPSE AND THE ADT PLUGIN

- It's possible to develop, test, and deploy an Android application from the command line using only a text editor, a JDK, the SDK, the SDK Platform-tools and an Android platform.
  - An Integrated Development Environment (IDE) makes development much more convenient and efficient, reducing time to market.
- Eclipse is the officially-supported IDE for Android development.
  - The **Android Developer Tools** (ADT) Plugin for Eclipse integrates and automates interaction with the Android SDK, Platform-tools, and emulators, making it easy to create, compile, debug, and deploy Android Applications.
  - The Eclipse installation that is included within the ADT bundle contains a preconfigured ADT Plugin.
- If you prefer, you can use an existing Eclipse installation rather than the one that comes in the bundle.
  - Install the ADT Plugin into Eclipse with the Eclipse Update Manager under the Eclipse **Help | Install New Software...** menu item.
    - Don't forget to point the ADT at your Android SDK installation within **Window | Preferences | Android**.

Hands On:
Follow the steps below to import the supporting files for this class into Eclipse.

1. Start Eclipse. Your instructor will tell you where to find Eclipse and what workspace to use.

2. In Eclipse, choose **File | Import... | General | Existing Projects into Workspace** and click **Next**.

3. Choose **Select archive file** and browse to the location of the student zip file. Your instructor will tell you where to find the student zip file.

4. Click **Finish**.

Note:
The JDK and ADT bundle should have already been installed on your system, but you can use the steps below if you wish to install the individual components separately, rather than using the ADT bundle. Please note that these steps assume that both the JDK and Eclipse are already installed.

1. Download the most recent SDK Tools executable file (for example: *installer_r22.0.4-windows.exe*) from ***http://developer.android.com/sdk/index.html***.
2. Run the executable to install the SDK.
3. Run **SDK Manager.exe.** Select the following four checkboxes: **Android SDK Platform-tools** beneath **Tools, SDK Platform** and **ARM EABI v7a System Image** beneath **Android 4.2.2 (API 17)**, and **Android Support Library** under **Extras**. Uncheck any other checkboxes.
4. Click the **Install 4 packages**... button.
5. Click the **Install** button on the **Choose Packages to Install** window.
6. When the installation completes, dismiss any open windows.
7. Start Eclipse.
8. Select **Help | Install New Software...**
9. Click the **Add...** button in the upper right hand corner of the **Available Software** window.
10. In the **Add Repository** window, type "ADT Plugin" for the **Name** and "https://dl-ssl.google.com/android/eclipse/" for the **Location** and click **OK**.
11. Back in the **Available Software** window, select the checkbox next to **Developer Tools** and click **Next**.
12. Click **Next** on the **Install Details** page.
13. On the **Review Licenses** page, accept the license agreement and click **Finish**.
14. The **Installing Software** window will display the installation progress. If you get a security warning, click **OK**.
15. Upon completion of the installation, restart Eclipse.
16. When Eclipse restarts, go to **Window | Preferences | Android**.
17. Verify the **SDK Location** is set to the location in which you installed the SDK in step 2.
18. Choose **OK**.

A SIMPLE ANDROID APPLICATION

- The ADT plugin makes it easy to create a new Android application by using the **New Android Project** wizard within the Eclipse IDE.
  - As part of the wizard, you can generate a simple activity within a specified package.
- Each Android application is made up of multiple files and directories within an Eclipse project.
  - Add the Java source code for your application to packages embedded within the *src* directory.
  - Place resources you plan to use in your code, such as images and strings, within the *res* directory.
    - Place an XML layout file for each of your activities within the *res/layout* directory
    - Include any string resources your application depends on within the *res/values* directory.
  - Configure your application within *AndroidManifest.xml*.
  - Eclipse and the SDK also generate extra files to support your application:
    - The SDK creates the *gen* directory and the embedded *R.java* file that Android relies on to resolve resource references.
    - The *bin* directory contains the generated *classes.dex* file as well as an *Application Package* (.apk) file which contains a zipped up version of your application that can be distributed to an Android device.
    - The *project.properties* file identifies the target API level that your application supports.

Hands On:
Create a new Android project within Eclipse (exact sequence may differ depending on your Eclipse version):

1. In Eclipse choose **File | New | Android Application Project**

2. On the **New Android Application** screen:
   a. **Application Name:** HelloAndroid
   b. **Project Name:** HelloAndroid
   c. **Package Name:** com.example.helloandroid
   d. **Minimum Required SDK:** API 8: Android 2.2 (Froyo)
   e. **Target SDK:** API 17: Android 4.2 (Jelly Bean)
   f. **Compile With:** API 17: Android 4.2 (Jelly Bean)
   g. **Theme:** Holo Light with Dark Action Bar
   h. Click **Next**

3. On the **New Android Application (Configure Project)** screen:
   a. Uncheck **Create custom launcher icon**
   b. Click **Next**

4. On the **Create Activity** screen:
   a. Choose **Blank Activity**
   b. Click **Next**

5. On the **Blank Activity** screen:
   a. **Activity Name:** HelloAndroidActivity
   b. **Layout Name:** activity_hello_android
   c. Click **Finish**

6. Click on the various directories and files that were created to get an idea of what a simple Android project look like. Your instructor will go into much greater detail about each of these files/directories later in the class. At a minimum, do the following:

   a. Open *src/com.example.helloandroid/HelloAndroidActivity.java* to view a simple activity.

   b. Open *res/layout/activity_hello_android.xml* to view the layout file for the above activity. Click the **activity_hello_android.xml** tab at the bottom of the editor to switch from the Graphical Layout view to an XML editor.

   c. Open *res/values/strings.xml* to view the strings resource file for this application. Click the **strings.xml** tab at the bottom of the editor to switch from the Resources view to an XML editor.

      i. Modify the string entry named **hello_world** to contain the value **Hello Android World!**

   d. Open *AndroidManifest.xml* to view the configuration for this application. Click the **AndroidManifest.xml** tab at the bottom of the editor to switch from the Manifest view to an XML editor.

THE ANDROID EMULATOR

- The most useful tool included in the SDK is an Android emulator that you can use to test your applications without having to deploy to an actual hardware device.
  - The emulator provides most of the functionality that you would have with a physical device including the ability to click on navigation and control keys.
  - You can configure multiple different emulators using *Android Virtual Device* (AVD) configurations.
    - AVDs allow you to test on both Android 2.3 and 4.2, for example.
- Configure and start an emulator using the ADT provided **Window | Android Virtual Device Manager** menu within Eclipse.
  - Click the **New...** button to create an AVD giving it a name and target API level.
    - You can also specify the size of a virtual SD Card.
  - Click the **Start...** button to launch an AVD from Eclipse.
    - Starting a new AVD may take a long time depending upon your computer's hardware specifications.
    - Once the AVD starts, follow its instructions to unlock the screen.
- To run an Android application within the emulator, simply click on the **Run | Run** Eclipse menu and choose to run the application as an Android Application.
  - If an AVD is already running, your application will be deployed to it, otherwise one will be launched for you.

Hands On:

Create a new AVD configuration and run the **Hello Android** application by following the steps below:

1. In Eclipse choose **Window | Android Virtual Device Manager**
2. Click the **New...** button to configure a new AVD.
    a. **Name:** MyJellyBean
    b. **Device:** 4.0" WVGA (480x800 hdpi)
    c. **Target:** Android 4.2.2 – API Level 17
    d. **CPU/ABI:** ARM (armabi-v7a)
    e. **Back Camera:** Emulated
    f. **SD Card Size:** 100 MiB
    g Click the **OK** button.
3. Choose the new AVD in the list and click the **Start...** button.
    a. Click the **Launch** button.
    b. After waiting for some time, the emulator should launch.
    c. If the screen is locked, click and drag the lock icon outside the circle.
4. Close the Android Virtual Device Manager window.
5. Run the **Hello Android** application by choosing the project name in the **Package Explorer** and clicking on the **Run | Run** Eclipse menu option.
    a. Choose **Android Application** within the **Run As** dialog.
    b. Click **OK**
    c. If you are asked whether to automatically monitor logcat output, choose Yes.

User Interface Layouts

- A typical Android activity contains user interface components that display static text as well as dynamic components that allow for user interaction.
  - Use a **TextView** component for non-editable text and an **EditText** for editable content.
  - Use a **Button** to trigger event handling code when the button is pressed.
- Define components within the layout XML file specifying at least three attributes.
  - Define the **android:id** attribute to provide an identifier that you can use to reference the component in code or elsewhere in the layout file.
  - Set the **android:layout_width** and **android:layout_height** to provide sizing information for the component.
    - Use the value **match_parent** to stretch the component to fill the same space that the parent component consumes.
    - Use the value **wrap_content** to squeeze the component to only fill as much space as it needs to display its contents.
- Each component also has more specific attributes that you can use to refine its look and feel.
  - Add the **android:hint** attribute to an **EditText** component to display a hint to the user as to what they should type into the text box.
  - Add the **android:text** attribute to any component that can display text to define its text content.
    - You can specify a string literal as the value of the attribute or add a reference to a predefined string in *res/values/strings.xml*.

Hands On:

Modify *HelloAndroid/res/layout/activity_hello_android.xml* to match the content below:

```
<RelativeLayout
     xmlns:android="http://schemas.android.com/apk/res/android"
     xmlns:tools="http://schemas.android.com/tools"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     android:paddingBottom="@dimen/activity_vertical_margin"
     android:paddingLeft="@dimen/activity_horizontal_margin"
     android:paddingRight="@dimen/activity_horizontal_margin"
     android:paddingTop="@dimen/activity_vertical_margin"
     tools:context=".HelloAndroidActivity" >

    <EditText
         android:id="@+id/name"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:layout_alignParentLeft="true"
         android:layout_alignParentTop="true"
         android:hint="Please enter your name" />

     <Button
         android:id="@+id/sayHello"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_below="@id/name"
         android:text="Say Hello" />

    <TextView
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:id="@+id/hello"
         android:layout_below="@id/sayHello" />

</RelativeLayout>
```

Place the **EditText** box in the upper left corner of the screen.

Place the **Button** below the **EditText** box.

Place the **TextView** below the **Button**.

Click on the Graphical Layout tab below the editor to see how your components will display when you run the application within the emulator.

You can use either the Graphical Layout view or the XML editor to modify user interface layouts.

ANDROID EVENT HANDLERS

- Android activities use listeners, typically defined as inner classes, for their event handling code.
  - A **Button**, for example, will trigger an **onClick** event that can be handled by any class that implements the **View.OnClickListener** interface.

```
class MyListener implements View.OnClickListener() {
   public void onClick(View v) {
      // Add your code here
   }
}
```

  - The handler is passed a reference to the view in which the event was triggered, so a single listener object can handle events for multiple views.
  - Retrieve a reference to the **Button** object, or any UI component, by calling the **findViewById()** method.

```
Button myButton = (Button) findViewById(R.id.myButton);
```

    - Pass in the identifier you specified in the XML layout file using the generated **R** class to access it.
  - Call the **setOnClickListener()** method on the **Button** passing in the listener object to associate the **Button** and the event listener.

```
myButton.setOnClickListener(new MyListener());
```


Hands On:

Modify *HelloAndroid/src/com.example.helloandroid/HelloAndroidActivity.java* to match the content below:

```
package  com.example.helloandroid;

import  android.os.Bundle;
import  android.app.Activity;
import  android.view.View;
import  android.widget.*;

public class HelloAndroidActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Lookup the button and add a listener for button clicks
        Button sayHelloButton = (Button) findViewById(R.id.sayHello);
        sayHelloButton.setOnClickListener(new MyButtonListener());
    }

    class MyButtonListener implements View.OnClickListener {
        public void onClick(View v) {
            // Lookup the gui controls
            EditText nameEditText = (EditText) findViewById(R.id.name);
            TextView helloTextView = (TextView)findViewById(R.id.hello);

            // Retrieve the name from the EditText box
            String name = nameEditText.getText().toString();
            // Say 'Hello' in the TextView
            helloTextView.setText("Hello " + name + "!");
        }
    }
}
```

Click the **Run | Run** menu item in Eclipse to redeploy the application to the emulator. Enter your name in the **EditText** box and click the **Button** to see the application say "Hello" to you.

LOGCAT

- Write messages to the Android logging system using the **android.util.Log** class.
  - Call the **Log.v()**, **Log.d()**, **Log.i()**, **Log.w()**, or **Log.e()** methods for verbose, debug, information, warning, or error messages, respectively.
  - Each of these methods takes two **String** arguments.
    - Use the first **String** as a tag that identifies which class or activity invoked the log call.
    - Use the second **String** to pass the actual message to be logged.
  - Alternatively, you can call the **Log.println()** method passing a priority constant as the first parameter and then the two **String**s.
    - The priority can be (from lowest to highest): **VERBOSE**, **DEBUG**, **INFO**, **WARN**, **ERROR**, or **ASSERT**
  - You can add a **TAG** constant to your class to store the **String** to pass as the first parameter to the **Log** methods.

```
private static final String TAG = "HelloActivity";
```

- You can view the log messages using the LogCat tool.
  - Add LogCat to your Eclipse perspective by clicking on **Window | Show View | Other... | Android | LogCat**
  - You can filter the log messages that LogCat displays by priority, tag, and more.

Hands On:

Modify *HelloAndroid/src/com.example.helloandroid/HelloAndroidActivity.java* to include the bold lines below:

```
package  com.example.helloandroid;

import  android.app.Activity;
import  android.os.Bundle;
import  android.util.Log;
import  android.view.View;
import  android.widget.*;

public class HelloAndroidActivity extends Activity {
    private static final String TAG = "HelloAndroidActivity";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // Lookup the button and add a listener for button clicks
        Button sayHelloButton = (Button) findViewById(R.id.sayHello);
        sayHelloButton.setOnClickListener(new MyButtonListener());
    }

    class MyButtonListener implements View.OnClickListener {
        public void onClick(View v) {
            // Lookup the gui controls
            EditText nameEditText = (EditText) findViewById(R.id.name);
            TextView helloTextView = (TextView)findViewById(R.id.hello);

            // Retrieve the name from the EditText box
            String name = nameEditText.getText().toString();
            // Say 'Hello' in the TextView
            helloTextView.setText("Hello " + name + "!");

            // Log the name
            Log.v(TAG, "name = " + name);
        }
    }
}
```

Use **Log.v()** to log verbose messages.

Click **Window | Show View | Other... | Android | LogCat** in Eclipse to enable the LogCat view. Redeploy the application to the emulator. Enter your name in the **EditText** box and click the **Button**. Check the LogCat window to see your output.

LABS

Solution code for labs ❶ and ❷ is available in the **HelloAndroid2** project.

❶ Change the **HelloAndroid** application that you created during the lecture to use different text for the **Button** label and the **EditText** hint.

❷ Change the application again so that the greeting says "Hi" rather than "Hello" before your name.

Solution code for labs ❸ and ❹ is available within the **Converter** project.

❸ Create an Android project that converts between degrees fahrenheit and degrees celsius. Have the user type the fahrenheit temperature in an **EditText** box. When the user clicks on a **Button**, have a listener update a **TextView** to display the converted value.

❹ (Optional) Add a button to convert from celsius to fahrenheit, as well.

Celsius = (5.0/9.0) * (Fahrenheit -32)

Fahrenheit = ((9.0/5.0) * Celsius) + 32

CHAPTER 3 - ACTIVITIES

OBJECTIVES

- Create an activity.
- Implement callback methods to listen for activity lifecycle events.
- Transition from one activity to another with an intent.
- Configure an application via the *AndoidManifest.xml* file.
- Package an Android application within an *.apk* file.

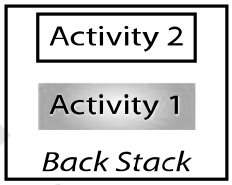
ACTIVITIES

- A typical Android application is made up of one or more *activities*.
  - Each activity provides a screen that you can interact with.
  - Designate at most one of the activities in your application as the main activity.
    - The *main activity* is run when you first launch an application.
  - Your main activity can invoke additional activities which are presented as additional screens.
- The *back stack* maintains a list of activities that represent the current activity and all of the activities that came before it stored in a last in, first out manner.
  - When you invoke a new activity, it is pushed to the top of the stack.
    - The previous activity is stopped and remains on the stack.
  - When you click on the **Back** key, the current activity is popped off the back stack and destroyed, and the next activity on the back stack takes user focus, becomes active, and its user interface is restored.

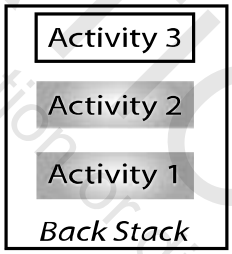
When an application is first launched, its main activity is the only resident of the back stack.

If the first activity chooses to launch another activity, then the second activity will move to the top of the back stack.

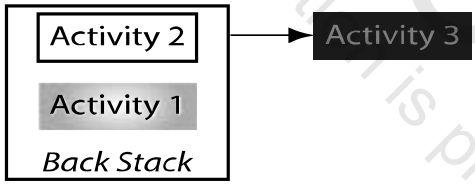
.



When a third activity is launched from the second activity, the stack now contains three items.



If you press the **Back** key, then the third activity is popped off the stack and terminated and the second activity becomes active.



CREATING AN ACTIVITY

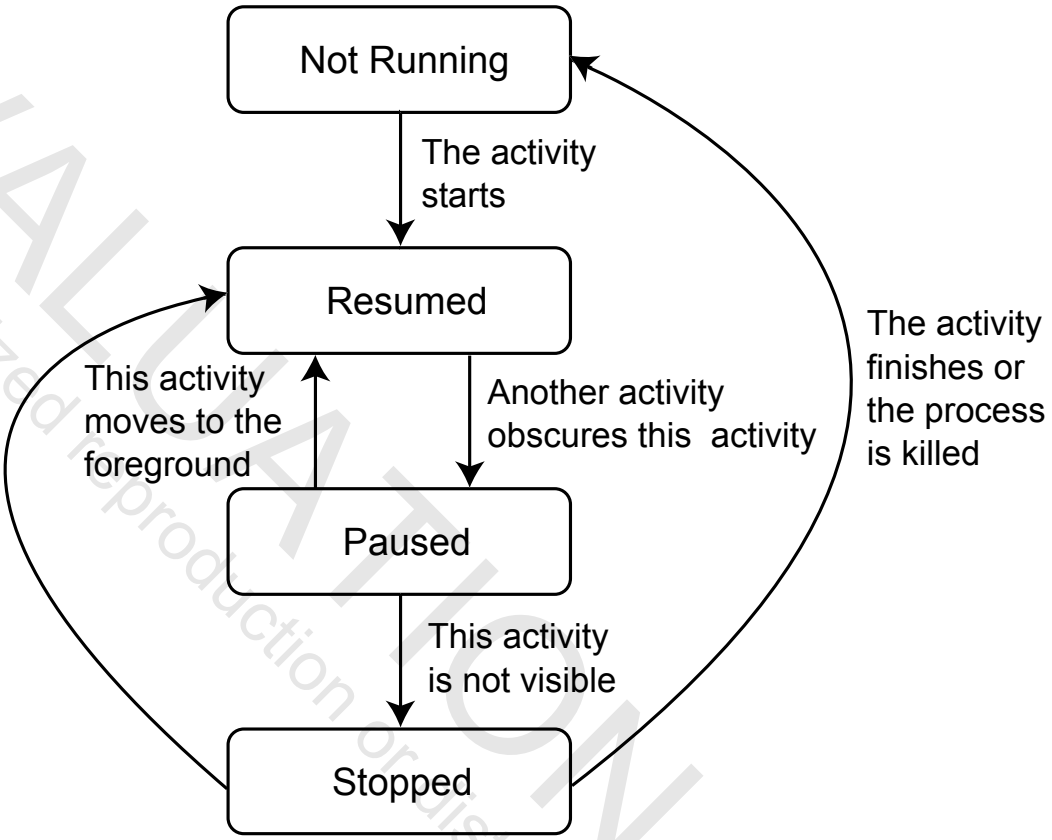
- Define an activity as a subclass of **android.app.Activity**.
- Override the **onCreate()** method to initialize your activity.
  - **onCreate()** is called by the system when your activity is first started.
  - Your implementation typically consists of the following:
    1. Call the superclass's implementation to initialize the parent.
    2. Call the **setContentView()** method to specify the user interface layout for this activity.
    3. Retrieve references to your GUI components using the **findViewById()** method.
    4. Associate the components with event handling listener classes.

HelloAndroid3/src/com.example.helloandroid/HelloAndroidActivity.java

```
package  com.example.helloandroid;

import  android.app.Activity;
import  android.os.Bundle;
import  android.view.View;
import  android.widget.*;

public class HelloAndroidActivity extends Activity {
    private TextView helloTextView;

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

        Button sayHelloButton = (Button) findViewById(R.id.sayHello);
        sayHelloButton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                EditText nameEditText =
                        (EditText) findViewById(R.id.name);
                String name = nameEditText.getText().toString();
                helloTextView.setText("Hi " + name + "!");
            }
        });
        ...
    }
    ...
}
```

ACTIVITY LIFECYCLE

- An activity can be in one of four states: *Not Running*, *Resumed*, *Paused*, or *Stopped*.
  - A Resumed, or running, activity is currently active and has the focus on the screen.
    - Resumed activities are stored at the top of the back stack.
  - A Paused activity is partially obscured by another activity, for example when a dialog pops up.
    - The other activity has the focus, but doesn't cover the whole screen.
  - A Stopped activity is completely covered by another activity, for example when a new activity is displayed.
    - Paused and Stopped activities are nested one or more layers deep within the back stack.
- Both Paused and Stopped activities remain alive and any data they contain is preserved in memory.



CALLBACK METHODS

- You can implement various callback methods within your activity to be notified of state transitions within your activity's lifecycle.

- The whole lifetime of your activity is encompassed between calls to the **onCreate**() and **onDestroy**() methods.

  - Initialize the application within the **onCreate**() method and cleanup any activity wide resources within the **onDestroy**() method.

    - You can call the **finish()** method on an activity to shut it down.

- The visible lifetime of your activity is encompassed between calls to the **onStart**() and **onStop**() methods.

  - Initialize any displayable content within the **onStart**() method and destroy it within the **onStop**() method.

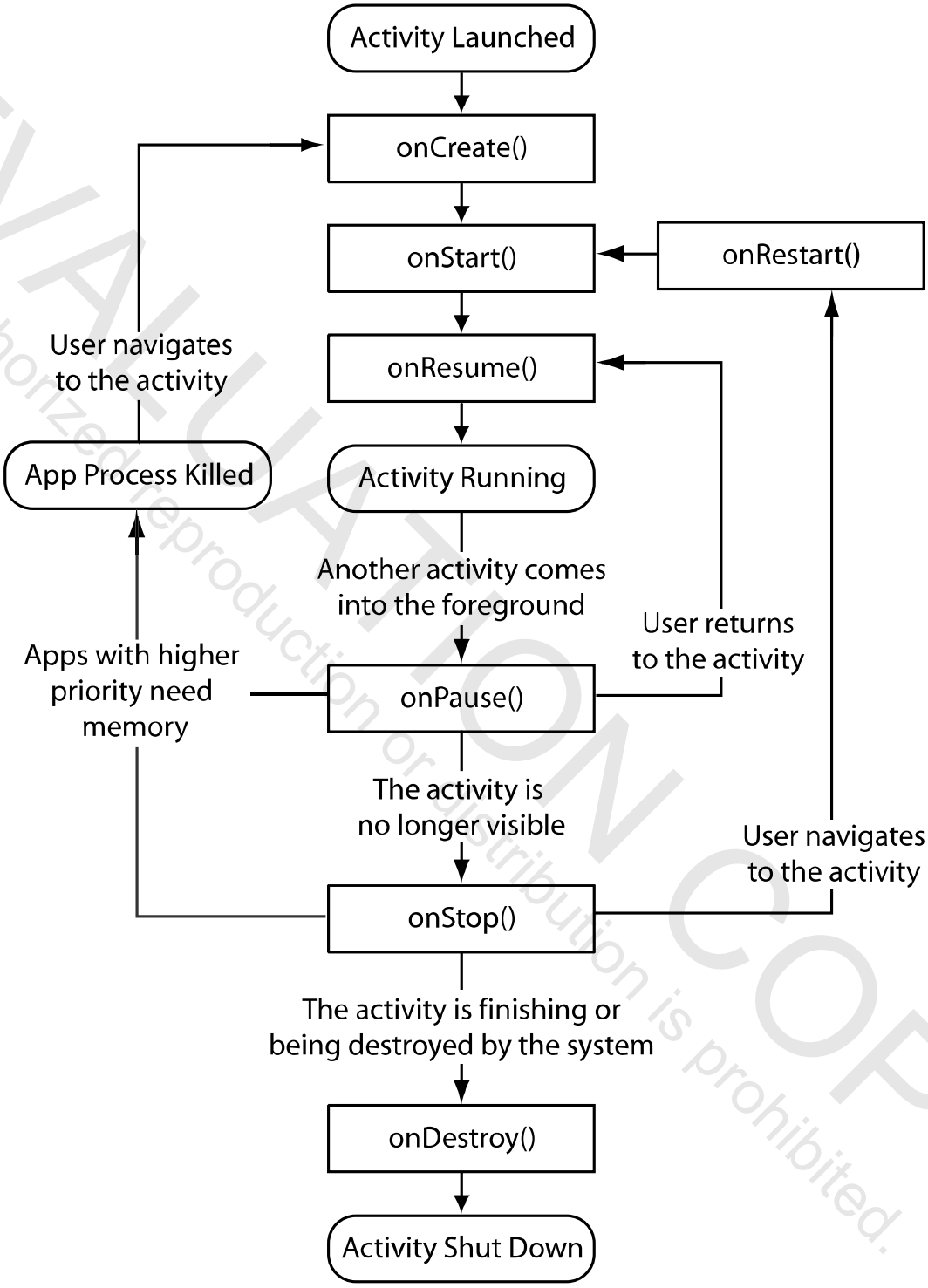
  - The **onRestart**() method will also be called before **onStart**() if the activity was previously stopped.

- The foreground lifetime of your activity is encompassed between calls to the **onResume**() and **onPause**() methods.

  - Cleanup resources, stop CPU consuming operations (i.e. audio), and persist unsaved data within the **onPause**() method and restore them within the **onResume**() method.

  - Transitions from a paused to a resumed state occur often so you should keep the code in these methods simple.

  - Don't forget to call the superclass implementation of the callback method from your method to ensure that default processing occurs.



Adapted from Android Developer API Guide, http://developer.android.com/guide/components/activities.html (accessed May 22, 2013)

RESOURCE CONSERVATION

- If the system encounters a low memory situation, stopped activities are removed before paused activities.
  - The system will either call the **finish()** method on an activity to ask it to finish, or it will kill the process associated with the activity.
  - The process can be killed after calls to **onPause**(), **onStop**(), or **onDestroy**().
    - Persist essential data within the **onPause**() method, because a killed activity may not have its **onStop**() or **onDestroy**() methods called.
  - If the system needs to resume the activity again (for example, when you click the **Back** button), it will have to recreate it.
- You can store a killed activity's state to a **Bundle** object so it can be restored later.
  - Override the **onSaveInstanceState()** callback method to store data to the passed in **Bundle** object before the activity is killed.
    - Store data within the **Bundle** object as name/value pairs.
    - You should use the **onPause()** method to store long-term persistent data and the **onSaveInstanceState()** method to save transient user interface state.
  - Use **onCreate()**'s **Bundle** parameter to retrieve saved instance state from a previously killed invocation.
    - New activities will be passed **null** since there is no saved data.

If you explicitly close an activity via a **Back** button click, then **onSaveInstanceState**() will not be called since the activity will not have to be restored.

You may not need to override the **onSaveInstanceState**() method if the default implementation is sufficient. It invokes the **onSaveInstanceState** () method on each view within your user interface. Each view can then store its own UI specific information into the bundle. This happens automatically for any view component that you've given an identifier. Make sure to call **super.onSaveInstanceState**() if you choose to provide your own implementation to ensure that this default processing occurs.

Configuration Changes

Android will restart your activity whenever a configuration change is detected. Restarting the activity results in a call to **onDestroy**() followed by **onCreate**(). One common example of a configuration change is a screen rotation. For this reason, it is always a good idea to handle activity restoration properly via **onSaveInstanceState**() and **onCreate**().

HelloAndroid3/src/com.example.helloandroid/HelloAndroidActivity.java

```
...
public class HelloAndroidActivity extends Activity {
    private TextView helloTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        helloTextView = (TextView) findViewById(R.id.hello);
        if (savedInstanceState != null) {
            helloTextView.setText(savedInstanceState
                    .getString("greeting"));
        }
    }

    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putString("greeting", helloTextView.getText()
                .toString());
    }
}
```

INTENTS

- Use an **Intent** to launch an activity from another activity.

```
Intent i = new Intent(this, NewActivity.class);
startActivity(i);
```

  - Create a new instance of an **Intent** object providing your activity as the first constructor argument and the **Class** of the activity to start as the second argument.
  - Call the **startActivity()** method passing in the newly created **Intent** object to start the new activity.

HelloAndroid3/src/com.example.helloandroid/WelcomeActivity.java

```
package  com.example.hello;

import  android.app.Activity;
import  android.content.Intent;
import  android.os.Bundle;
import  android.view.View;
import  android.widget.Button;

public class WelcomeActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.welcome);

        Button nextButton = (Button) findViewById(R.id.next);
        nextButton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                Intent i = new Intent();
                i.setClass(WelcomeActivity.this,
                        HelloAndroidActivity.class);
                startActivity(i);
            }
        });
    }
}
```

Access the enclosing activity from an inner class using **ClassName.this** notation.

Try It:

Open the **HelloAndroid3** project. Select **Run | Run** from the menu and choose **AndroidApplication** from the **Run As** dialog to view the **WelcomeActivity**. Click the **Next** button to launch the **HelloAndroidActivity** using an **Intent**.

ANDROIDMANIFEST.XML

- Configure your application within the manifest file called *AndroidManifest.xml*.
- Set the Android namespace to ***http://schemas.android.com/apk/res/android*** within the root **<manifest>** element.
  - Specify the package name you used for your activities within the **<manifest>** element's **package** attribute.
- Set the minimum and target Android SDK versions that you support within the **<uses-sdk>** element.
- Declare your application within the appropriately named **<application>** element.
  - Set the icon, label, and theme for your application using attributes.
    - The icon and label are viewable in the application launcher or as a shortcut on the **Home** screen.
    - The theme references an application-wide style configured in an XML file under the *res/values/* directory.
- Every activity you create must be declared within an **<activity>** element.
  - Specify the activity's class name using the **android:name** attribute.
    - Prepend the activities class name with a dot to indicate that the class is relative to the package declared in the **package** attribute within the <**manifest>** tag.
  - Use an **<intent-filter>** with an **action** of **android.intent.action.MAIN** to identify an activity as a main activity.
    - Set the **category** to **android.intent.category.LAUNCHER** to specify that this activity should be listed in the application launcher.

HelloAndroid3/AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.helloandroid"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".WelcomeActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".HelloAndroidActivity"
            android:label="@string/app_name" >
        </activity>
    </application>

</manifest>
```

The **android:name** values elsewhere in this document are relative to this package.

Allow this application's data to participate in backups.

PACKAGING

- The Dalvik VM interprets *.dex* (Dalvik Executable) files rather than standard Java class files.
  - The *.dex* bytecode format is optimized for memory and processor constrained environments.
- You don't distribute the *.dex* files directly to your Android device, instead you package them within an *.apk* file.
  - An *.apk* file contains everything you need to run an Android application, including *.dex* files, resources, and certificates.
  - *.apk* files are similar to JAR files and must end in an *.apk* extension.
- Eclipse creates the *.dex* and *.apk* files automatically for you, but you can also build them manually using the following steps:
  1. Use the Android Asset Packaging Tool (**aapt**) to compile the various XML files, including the manifest file, and to produce an *R.java* file.
  2. Compile your source code as well as the *R.java* file into *.class* files using the standard Java compiler.
  3. Convert the *.class* files from step 2, as well as *.class* files from libraries you depend on into *.dex* files using the **dx** tool.
  4. Use the **apkbuilder** tool to package the *.dex* files as well as compiled resources and non-compiled resources like images into an *.apk* file.
  5. Sign the *.apk* file with a debug key or private key and optionally optimize it with the **zipalign** tool to decrease its memory usage for production.
  6. Distribute the *.apk* file to your emulator or, if the *.apk* is signed with a private key, your Android device.

Hands On:

While building an Android application on the command line is less common than building it in Eclipse, it is beneficial to see the process outside of the IDE. The Android SDK includes a command line tool to create a basic Android project. The generated project contains an Ant script to build the *.apk* file.

Open a command prompt and run the **android create project** command to build a new project in the c:*android* directory:

```
android create project --target 1 --name AndroidApp  --path C:/android/
AndroidAppProject --activity AndroidAppActivity --package
com.example.commandline
```

Examine the newly created files, then change directories to the *C:\android\AndroidAppProject* directory.

```
cd C:\android\AndroidAppProject
```

Run the **ant debug** command to create the *.apk* file.

```
ant debug
```

Examine the ant output and the generated files.

To deploy the generated *.apk* file to the emulator you must start the emulator and install the *.apk* to it. If an emulator is not already running, then you can start one by running the **emulator** command passing in the name of the emulator to run:

```
emulator -avd MyJellyBean
```

Wait for the emulator to start, then run the **adb** command to install the *.apk* to the emulator:

```
adb install C:\android\AndroidAppProject\bin\AndroidApp-debug.apk
```

Look for the newly installed application within the application launcher on the emulator. If multiple emulators are running, then you must specify the emulator to install to with the **adb –s** option. To list the current devices run **adb devices**.

To uninstall an application, run **adb uninstall** followed by the package name of the activity to uninstall:

```
adb uninstall com.example.commandline
```

LABS

Use either the **Converter** project from the student files or your own **Converter** solution from a previous lab as a starting point for lab ❶.

You can find solution code for labs ❶ -❺ in the **Converter2** project.

❶ Create a distance converter activity within the **Converter** application that converts miles to kilometers and kilometers to miles. Start by creating an XML layout for the new activity, then create the new class that extends **android.app.Activity**. The rest of the code will be similar to the temperature conversion activity.

❷ Make your distance converter activity the **MAIN** activity for this application within the manifest (be sure to change the name of the temperature converter activity). Test the activity within the emulator.

❸ Create a new activity that provides buttons to choose between the temperature converter or distance converter and transition to each via intents when the button is selected. Make sure to update the manifest to make this new activity the **MAIN** activity.

❹ Add callback methods to the menu activity for the various callbacks that were discussed in the chapter. Make sure to call the superclass method, then add trace messages using LogCat to track when the various methods are called.
Hint: In Eclipse, use **Source | Override/Implement Methods** and select the methods you want to override, to autogenerate method stubs.

❺ Perform the following operations on your application and watch the LogCat messages as they are displayed:

a. Start the **Converter** application.
b. Click the **Temperature** button to transition to a new activity.
c. Click the Android **Back** button to transition back to the menu activity
d. Click the Android **Back** button again to shutdown the menu activity.
e. Restart the **Converter** application through the application launcher.
f. Click the **Home** button, then go to the application launcher and click **Settings | Apps | Converter** within the emulator. Uninstall the **Converter** application to kill the process associated with the activity.

A solution for lab ❻ is under **extras/clockapp**.

❻ Use command line tools to create a new project and activity that displays the current time. Install and test it in your emulator.

kilometers = miles * 1.60934
miles = kilometers * 0.621371

CHAPTER 9 - MENUS

OBJECTIVES

- Add options, context, and popup menus to your application.
- Create submenus with radio buttons and checkboxes.

MENUS AND MENU ITEMS

- Add menus to your application to allow your users to access additional activities, customize the application, or perform other actions.
  - An *options menu* will appear in the action bar at the top of the screen.
    - On devices running Android 2.3 or lower, the options menu is only available when the user touches the **MENU** button.
  - A *context menu* will appear as a floating menu when the user long touches a control that has an associated menu.
  - *Contextual action mode* displays a contextual action bar at the top of the screen when a view with an associated menu is selected.
    - Contextual action mode is preferred over context menus in Android 3.0 and above.
  - New to Android 3.0, a *popup menu* is a modal floating menu that is anchored to a view and displayed programmatically.
- Define your menu as an XML resource within the */res/menu/* directory.
  - Declare your root element as a **<menu>** with the standard **xmlns:android** namespace attribute.
  - Within the **<menu>**, add child **<item>** tags to define each menu item.
  - Each **<item>** should include **android:id** and **android:title** attributes.

Converter4/res/menu/optionsmenu.xml

```
<menu  xmlns:android="http://schemas.android.com/apk/res/android">
    <item
         android:title="@string/distance_converter"
         android:id="@+id/distance_converter"
         android:icon="@drawable/ic_menu_ruler"
         android:showAsAction="ifRoom"/>
    <item
         android:title="@string/mass_converter"
         android:id="@+id/mass_converter"
         android:icon="@drawable/ic_menu_anchor"
         android:showAsAction="ifRoom"/>
    <item
         android:title="@string/temperature_converter"
         android:id="@+id/temperature_converter"
         android:icon="@drawable/ic_menu_settings"
         android:showAsAction="ifRoom"/>
    ...
</menu>
```

With an option menu, you can define an image to associate with your menu item using an **android:icon** attribute.

Note:

You can create menus programmatically with **Menu.add(int groupId, int itemId, int order, CharSequence title)**:

CheckRegister/src/com.example.checkregister/CheckRegisterActivity.java

```
  ...
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
       super.onCreateOptionsMenu(menu);
       MenuItem prefItem = menu.add(0,0,0,"Preferences");
        prefItem.setIcon(android.R.drawable.ic_menu_preferences);
       MenuItem fileItem = menu.add(0,1,1,"Logfile");
        fileItem.setIcon(android.R.drawable.ic_menu_recent_history);
       MenuItem viewItem = menu.add(0,2,2,"History");
        viewItem.setIcon(android.R.drawable.ic_menu_view);
       MenuItem editItem = menu.add(0,3,3,"Edit");
        editItem.setIcon(android.R.drawable.ic_menu_edit);
       MenuItem monthlyItem = menu.add(0,4,4,"Monthly");
        monthlyItem.setIcon(android.R.drawable.ic_menu_view);
       return true;
  }
  ...
```

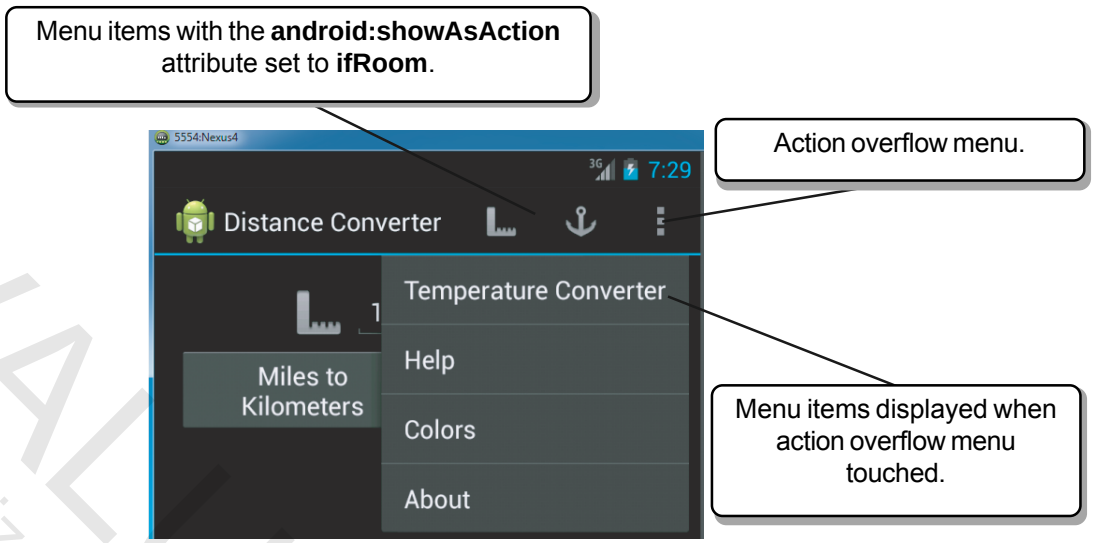
Like layouts however, menus are much more conveniently laid out and maintained in XML resource files.

OPTIONSMENU

- Use an options menu to display activity-wide items on the action bar.
- An options menu **<item>** can specify the **android:showAsAction** attribute to influence where the menu item is placed on the action bar.
  - Set the attribute to **ifRoom** to have that item display itself directly on the action bar if space is available.
  - Omit or set the attribute to **never** to keep the item in the overflow menu.
- Inflate an XML menu resource into a **Menu** object as part of the **onCreateOptionsMenu()** callback method in your activity or fragment.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
   MenuInflater inflater = getMenuInflater();
   inflater.inflate(R.menu.myMenu, menu);
   return true;
}
```

  - A **Fragment** must have called **setHasOptionsMenu(true)** to receive this method call.
  - Get a **MenuInflator** instance from the **getMenuInflater()** method.
  - Call the **inflate()** method on a **MenuInflator** passing in the menu resource ID and a **Menu** object to populate as parameters.
  - Return a **boolean** indicating whether or not to display the menu.
- The **onCreateOptionsMenu()** method is called once by the system when your activity or fragment is first created.
  - Android will cache the **Menu** object you build so it can display it again in the future without having to recreate it.



Converter4/src/com.example.converter/MenuActivity.java

```
  ...
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
     MenuInflater inflater = getMenuInflater();
     inflater.inflate(R.menu.optionsmenu, menu);
    return true;
  }
  ...
```

You can add the **onPrepareOptionsMenu()** callback method if you want to modify the menu before it is displayed. This method is called each time the menu is selected rather than just once like **onCreateOptionsMenu()**.

Converter4/src/com.example.converter/DistanceConverterActivity.java

```
...
public class DistanceConverterActivity extends MenuActivity {
  ...
  @Override
  public boolean onPrepareOptionsMenu(Menu menu) {
     super.onPrepareOptionsMenu(menu);
     menu.findItem(R.id.distance_converter).setEnabled(false);
    return true;
  }
  ...
}
```

Disable the menu item that corresponds to this activity.

Note:

When you choose the **Menu** button in Android 2.3 and lower, an options menu will display at the bottom of the screen with up to six individual menu items complete with icons and text titles. If you have more than six menu items, then a **More** menu item will appear instead of the sixth item. Clicking on the **More** menu item will display an overflow menu which contains the additional menu items.

REACTING TO MENU ITEM SELECTIONS

- Add the **onOptionsItemSelected()** callback method to your activity to react to options menu item selections.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    ...
}
```

  - Call the **getItemId()** method on the passed in **MenuItem** to determine which menu item was selected.
    - The returned ID is the same one you specified with the **android:id** attribute in your XML menu resource definition, or with the **itemId** argument to **Menu.add()**.
  - Return **true** to indicate that the menu item was successfully handled in this method.
  - If you don't recognize the **MenuItem** that was passed in, then call the super implementation of this method to allow it to handle the selection.
  - If you are using fragments, the method is called on your activity first, then on each fragment in the order it was added until one returns **true**.

- If multiple activities within the same application will share the same menu items, then you should define the **onCreateOptionsMenu()** and **onOptionsItemSelected()** methods in a superclass that your other activities extend.
  - You can still add additional menu items by overriding the methods and invoking their super implementations appropriately.

Converter4/src/com.example.converter/MenuActivity.java

```
...
public class MenuActivity extends Activity {
    ...
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        switch (id) {
        case R.id.temperature_converter:
            showTemperatureActivity();
            break;
        case R.id.distance_converter:
            showDistanceActivity();
            break;
        case R.id.mass_converter:
            showMassActivity();
            break;
        ...
        default:
            return super.onOptionsItemSelected(item);
        }
        return true;
    }

    public void aboutClicked(MenuItem mi) {
        showAboutDialog();
    }
    ...
}
```

MenuActivity is the superclass of the other activities in this project.

Since Android 3.0, you can create a public method with a **MenuItem** parameter as your callback method. Associate that method by name with a menu item using the **android:onClick** attribute.

Converter4/res/menu/optionsmenu.xml

```
<menu  xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    <item
        android:title="@string/about"
        android:id="@+id/about"
        android:onClick="aboutClicked"/>
</menu>
```

Try It:

Run the **Converter4** application on the emulator. Click the **MENU** key or choose the action overflow menu to display the remaining options menu items. Choose a menu item to see the results of the **onOptionsItemSelected()** callback method.

CONTEXTMENU

- Long press a view to display one or more context sensitive menu items in a **ContextMenu**.
  - Android won't display menu item icons in a **ContextMenu**.
- Register a view as requiring a **ContextMenu** by calling the **registerForContextMenu()** method passing in the **View** object as a parameter.

```
View myview = findViewById(R.id.myview);
registerForContextMenu(myview);
```

- Define the **onCreateContextMenu()** callback method to provide the code for menu inflation in your activity or fragment.

```
public void onCreateContextMenu(ContextMenu menu, View v,
     ContextMenuInfo menuInfo) {
   super.onCreateContextMenu(menu, v, menuInfo);
   MenuInflater inflater = getMenuInflater();
   inflater.inflate(R.menu.contextmenu, menu);
}
```

  - Unlike **onCreateOptionsMenu()**, this method returns **void**.
- Use the **onContextItemSelected()** callback method to define the code to react to menu selections.

```
public boolean onContextItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      ...
   }
}
```

  - Similar to **onOptionsItemSelected()**, return **true** to indicate that the menu item was successfully handled in this method and call the super implementation in the **switch**'s **default** statement.

Converter4/src/com.example.converter/DistanceConverterActivity.java

```
...
public class DistanceConverterActivity extends MenuActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        View image = findViewById(R.id.distanceImage);
        registerForContextMenu(image);
       ...
    }
    ...
}
```

Converter4/src/com.example.converter/MenuActivity.java

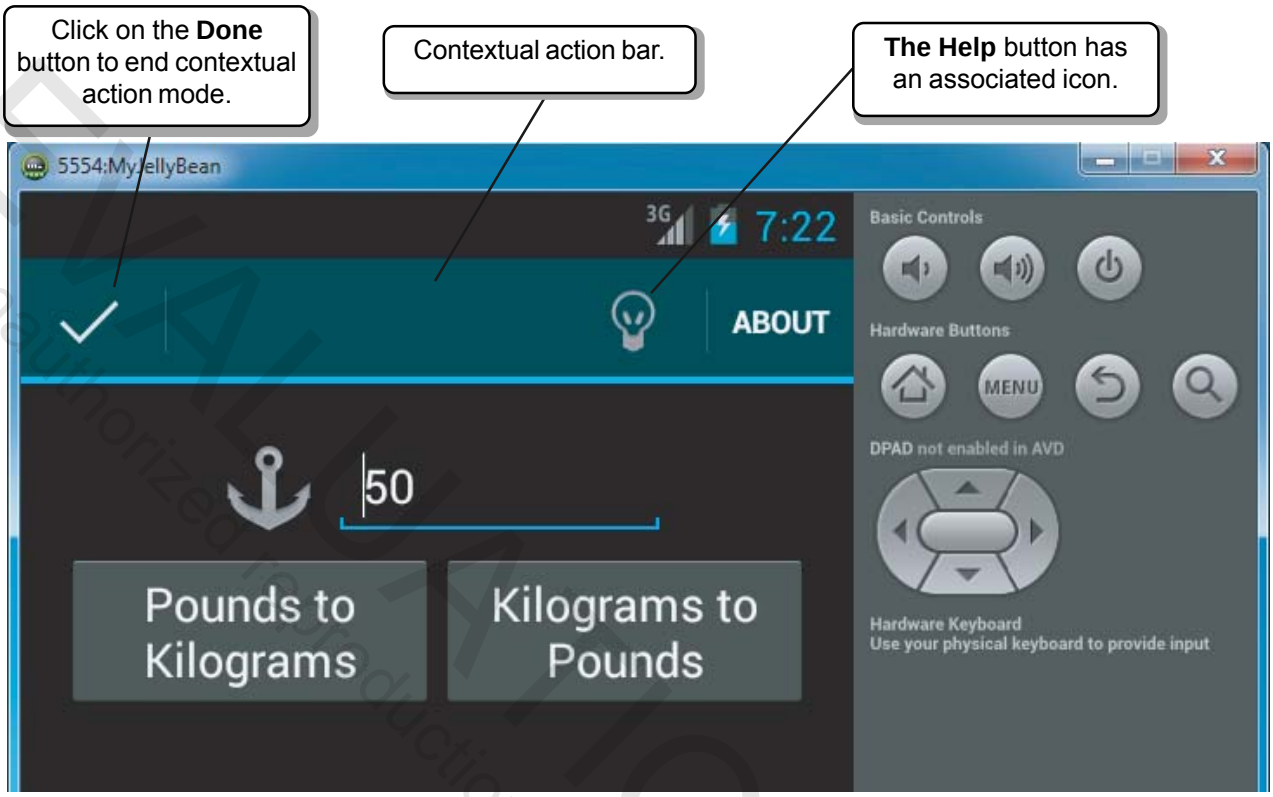
```
...
    @Override
    public void onCreateContextMenu(ContextMenu menu, View v,
            ContextMenuInfo menuInfo) {
        if (MenuActivity.enableContextSensitiveHelp) {
            super.onCreateContextMenu(menu, v, menuInfo);
            MenuInflater inflater = getMenuInflater();
            inflater.inflate(R.menu.contextmenu, menu);
        }
    }

    @Override
    public boolean onContextItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case R.id.about:
            showAboutDialog();
            break;
        case R.id.help:
            showContextSensitiveHelpDialog();
            break;
        default:
            return super.onContextItemSelected(item);
        }
        return true;
    }
...
```

Try It:

Long press the icon to the left of the **EditText** in the **DistanceConverter** activity to see a context menu.

CONTEXTUAL ACTION MODE

- When a view is selected, you can display context sensitive clickable items, called *contextual actions*, within a contextual action bar at the top of the screen .
  - What it means to "select a view" is left to the application designer.
    - For example, you can trigger contextual action mode based on a long click, similar to how context menus work.
    - Or, if your view is a checkbox (or something similar), then selecting it can trigger contextual action mode.
  - The view that is selected can be an individual view or something like a **ListView** or **GridView** where you can select multiple items and have the contextual actions apply to them all as a batch.
  - Your activity will remain in contextual action mode and the contextual action bar will remain visible until one of the following occurs:
    - The BACK button is pressed.
    - The item is deselected.
    - The *Done* action, denoted by a checkmark, is selected.
    - You dismiss it programmatically with the **finish()** method.
- You should use contextual action mode instead of context menus for applications that target Android 3.0 and higher.
- Unlike context menus, you can use icons in contextual action mode.



Try It:

Long press the icon to the left of the **EditText** in the **MassConverter** activity to see contextual action mode.

DEFINING CONTEXTUAL ACTIONS

- Implement the **ActionMode.Callback** interface to specify callback methods for your contextual action mode.
  - Define the **onCreateActionMode()** method to inflate a menu that represents the contents of the contextual action bar.
    - This method is called when the action mode is first created and returns **true** to indicate that the action mode should be created.
  - Implement the **onPrepareActionMode()** method to refresh the action mode when it is invalidated.
    - Return **false** if no changes were made.
  - Implement the **onDestroyActionMode()** method to be notified when the action mode is exited.
  - Write the **onActionItemClicked()** method to specify what should occur when each action is clicked.
    - This method is similar to **onContextItemSelected()**, except for the additional **ActionMode** parameter.
    - Call **finish()** on the **ActionMode** parameter if you wish to end the contextual action mode and close the contextual action bar.
- Call the **startActionMode()** method when the view is selected passing an instance of your **ActionMode.Callback**.

```
ActionMode.Callback cb = new MyActionModeCallback();
ActionMode am = startActionMode(cb);
```

  - This triggers a call to the **onCreateActionMode()** method.

Converter4/src/com.example.converter/MassConverterActivity.java

```
...
public class MassConverterActivity extends MenuActivity {
    private ActionMode actionMode;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        View image = findViewById(R.id.massImage);
        final Callback callback = new ActionModeCallback();
        image.setOnLongClickListener(new View.OnLongClickListener() {
            public boolean onLongClick(View view) {
                if (actionMode != null) {
                    return false;
                }
                actionMode = startActionMode(callback);
                return true;
            }
        });
        ...
    }
    ...
    class ActionModeCallback implements ActionMode.Callback {
        public boolean onCreateActionMode(ActionMode mode, Menu menu) {
            MenuInflater inflater = mode.getMenuInflater();
            inflater.inflate(R.menu.contextmenu, menu);
            return true;
        }
        public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
            return false;
        }
        public void onDestroyActionMode(ActionMode mode) {
            actionMode = null;
        }
        public boolean onActionItemClicked(ActionMode mode,
                MenuItem item) {
            switch (item.getItemId()) {
            case R.id.about:
                showAboutDialog();
                mode.finish();
                return true;
            case R.id.help:
            ...
            }
        }
    }
}
```

Do not start another action mode if one is currently active.

Set the **actionMode** field to **null** so that future item selections will start the action mode.

Call **finish()** to dismiss the action mode.

POPUPMENU

- Use a **PopupMenu** in Android 3.0 and above when you wish to display a modal menu anchored to a view.
  - The menu will display below the view if space is available, otherwise above it.
  - Dismiss a popup by simply touching outside of it or by programmatically calling the **dismiss()** method on it.
- Implement **PopupMenu.OnMenuItemClickListener** and define the **onMenuItemClick()** method to react to menu selections.
  - The implementation is similar to what you would use for an options or context menu.
- When it is time to display the menu, follow the four steps below to create and display the popup:
  1. Create an instance of a **PopupMenu** passing the current application context and the view it is associated with to the constructor.
  2. Associate the **OnMenuItemClickListener** with your **PopupMenu** by calling the **setOnMenuItemClickListener()** method.
  3. Inflate a menu by calling the **inflate()** method on your **PopupMenu** instance passing the resource ID of the menu to inflate.
  4. Finally, call the **show()** method to display the popup.

Converter4/src/com.example.converter/TemperatureConverterActivity.java

```
...
public class TemperatureConverterActivity extends MenuActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.temperature);

        View image = findViewById(R.id.temperatureImage);
        image.setOnLongClickListener(new View.OnLongClickListener() {
            public boolean onLongClick(View view) {
                PopupMenu popup = new PopupMenu(
                        TemperatureConverterActivity.this, view);
                popup.setOnMenuItemClickListener(new PopupListener());
                popup.inflate(R.menu.contextmenu);
                popup.show();
                return true;
            }
        });
        ...
    }
    ...
    class PopupListener implements PopupMenu.OnMenuItemClickListener {
        public boolean onMenuItemClick(MenuItem item) {
            switch (item.getItemId()) {
            case R.id.about:
                showAboutDialog();
                return true;
            case R.id.help:
                showContextSensitiveHelpDialog();
                return true;
            default:
                return false;
            }
        }
    }
}
```

The **inflate()** method was added in API Level 14.

Try It:

Long press the icon to the left of the **EditText** in the **TemperatureConverter** activity to see a **PopupMenu**.

SUBMENUS

- Your menus can have submenus.
  - A submenu opens when its parent menu item is selected.
  - Submenus allow you to group together similar functionality under a common heading.
- Create a submenu in XML by adding a **<menu>** element as a child of the associated menu item's **<item>** tag.
  - Add each submenu item using **<item>** children of the **<menu>**
  - Submenus cannot have submenus of their own, nor can they have icons.
- When you select an item in a submenu, the appropriate **on*xxx*ItemSelected()** is called based upon whether the submenu is associated with an options or context menu.

Converter4/res/menu/optionsmenu.xml

```
<menu  xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    <item
        android:title="@string/help"
        android:id="@+id/help_choices"
        android:icon="@drawable/ic_menu_light">
        <menu>
            <item
                android:id="@+id/help_distance"
                android:title="@string/help_distance"/>
            <item
                android:id="@+id/help_mass"
                android:title="@string/help_mass"/>
            <item
                android:id="@+id/help_temperature"
                android:title="@string/help_temp"/>
            <item
                android:id="@+id/context_sensitive_help_enabled"
                android:title="@string/enable_context_help"
                android:checkable="true"
                android:checked="true"/>
        </menu>
    </item>
    ...
</menu>
```

Create a submenu by placing a **<menu>** element within a parent menu **<item>** element.

Converter4/src/com.example.converter/MenuActivity.java

```
...
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        ...
        case R.id.help_distance:
            showDistanceHelpDialog();
            break;
        case R.id.help_mass:
            showMassHelpDialog();
            break;
        case R.id.help_temperature:
            showTemperatureHelpDialog();
            break;
        ...
    }
...
```

Try It:

Choose the **Help** options menu item to display the submenu.

CHECKBOXES AND RADIO BUTTONS IN MENU ITEMS

- You can also display menu items with radio button or checkbox functionality in a submenu.
  - For radio buttons, declare your menu items within a **<group>** element with the **android:checkableBehavior** attribute set to **single** in your XML menu resource.
    - Only one menu item within the group can be selected at a time.
  - For checkboxes, either set the **<group>**'s **android:checkableBehavior** attribute to **all** or directly set each **<item>**'s **android:checkable** attributes to **true**.
  - Set individual **<item>** elements as pre-checked by setting the **android:checked** attribute to **true**.
- For both radio buttons and checkboxes, the **on*xxx*ItemsSelected()** method is called when a corresponding menu item is chosen.
  - Android does not automatically update the state of the radio button or checkbox for you.
    - You must invoke the **setChecked()** method on the selected menu item to set its value.
    - For checkboxes, call the **isChecked()** method first to determine the starting value of the checkbox, then call the **setChecked()** method passing in the negated value.
- You can also use the **<group>** element to enable bulk operations on all items in the group.
  - For example, you can enable all members of a group by passing the group's id to **Menu**.**setGroupEnabled()**.

Converter4/res/menu/optionsmenu.xml

```
...
      <item
         android:id="@+id/context_sensitive_help_enabled"
         android:title="@string/enable_context_help"
        android:checkable="true"
        android:checked="true"/>
    </menu>
  </item>
  <item
    android:title="@string/colors"
    android:id="@+id/colors"
     android:icon="@drawable/ic_menu_colorpicker">
    <menu>
      <group android:id="@+id/rbs"
             android:checkableBehavior="single">
        <item
          android:id="@+id/color_black"
          android:title="@string/black"
          android:checked="true"/>
        <item
          android:id="@+id/color_gray"
          android:title="@string/gray"/>
        ...
      </group>
    </menu>
  </item>
...
```

Checkbox

Radio buttons

Converter4/src/com.example.converter/MenuActivity.java

```
...
         case R.id.context_sensitive_help_enabled:
             MenuActivity.enableContextSensitiveHelp = !item
                    .isChecked();
             item.setChecked(!item.isChecked());
            break;
        case R.id.color_black:
        case R.id.color_gray:
        case R.id.color_blue:
            item.setChecked(true);
            changeColor(id);
            break;
...
```

Try It:

Choose the **Colors** options menu item to display the submenu with radio buttons.

LABS

Use the **Countdown** project as starter code for these labs. The solutions can be found in the **Countdown2** project.

❶ The **Countdown** application currently displays the number of days until Christmas. Add an options menu to the application with menu items that allow you to customize the event name and event date.

❷ Add two more menu items to allow the user to set the background and foreground colors using radio buttons.

Hint:
You can set the foreground and background colors to XML color resource IDs with the following lines of code:

```
getWindow().setBackgroundDrawableResource(R.color.myColor);
textView.setTextColor(getResources().getColor(R.color.myColor));
```