

- [Eclipseプラグインを使ったAndroidアプリケーションの作成](#)
- [Eclipseプラグインを使わないAndroidアプリケーションの作成](#)
- [|#####ここまで翻訳#####|](#)
- [Implementing Activity Callbacks](#)
- [Opening a New Screen](#)
  - [Opening a Screen](#)
  - [Some Intent examples](#)
  - [Returning a Result from a Screen](#)
  - [Lifetime of the new screen](#)
- [Listening for Button Clicks](#)
- [Configuring General Window Properties](#)
- [Storing and Retrieving State](#)
  - [Storing and Retrieving Larger or More Complex Persistent Data](#)
- [Playing Media Files](#)
- [Listening For and Broadcasting Global Messages, and Setting Alarms](#)
  - [Sending the message](#)
  - [Receiving the message](#)
  - [Other system messages](#)
  - [Listening for phone events](#)
  - [Setting Alarms](#)
- [Displaying Alerts](#)
  - [Normal Alerts](#)
  - [AlertDialog](#)
  - [Notifications](#)
- [Displaying a Progress Bar](#)
- [Adding Your Application to the Favorites List](#)
- [Adding Items to the Screen Menu](#)
  - [Adding Submenus](#)
  - [Adding yourself to menus on other applications](#)
  - [The offering application](#)
- [Display a Web Page](#)
- [Binding to Data](#)
- [Capture Images from the Phone Camera](#)
- [Handling Expensive Operations in the UI Thread](#)
- [Selecting, Highlighting, or Styling Portions of Text](#)
- [List of Files for an Android Application](#)

## Eclipseプラグインを使ったAndroidアプリケーションの作成

---

Android Eclipseプラグインを使用することは、新しいAndroidアプリケーションの作成を始める最も高速で最も簡単な方法です。プラグインは自動的にあなたのアプリケーションのために正しいプロジェクト構造を生成し、リソースを自動的にコンパイルしてくれます。

Androidアプリケーションの動作原理を理解するために、[Androidアプリケーション解体新書](#)を読んでもみるのもよいでしょう。

SDKの sample/ フォルダでApiDemosアプリケーションと他のサンプルアプリケーションを見てみることもお勧めします。

最終的には[Hello Android](#)と[Notepad](#)のコードチュートリアルを行うことがEclipseでのAndroid開発を始めるには素敵な方法です。特にHello Androidチュートリアルを行うことは、Eclipseで新しいAndroidアプリケーションを作成するための優れた導入方法となるでしょう。

## Eclipseプラグインを使わないAndroidアプリケーションの作成

---

|#####ここまで翻訳#####|

---

This topic describes the manual steps in creating an Android application. Before reading this, you should read [Overview of an Android Application](#) to understand the basics of how an Android application works. You might also want to look at the sample applications that ship with Android under the `samples/` directory.

Here is a list of the basic steps in building an application.

1. Create your required resource files This includes the `AndroidManifest.xml` global description file, string files that your application needs, and layout files describing your user interface. A full list of optional and required files and syntax details for each is given in [File List for an Android Application](#).
2. Design your user interface See [Implementing a UI](#) for details on elements of the Android screen.
3. Implement your Activity (this page) You will create one class/file for each screen in your application. Screens will inherit from an `android.app` class, typically `android.app.Activity` for basic screens, `android.app.ListActivity` for list screens, or `android.app.Dialog` for dialog boxes. You will implement the required callbacks that let you draw your screen, query data, and commit changes, and also perform any required tasks such as opening additional screens or reading data from the device. Common tasks, such as opening a new screen or reading data from the device, are described below. The list of files you'll need for your application are described in [List of Files for an Android Application](#).
4. Build and install your package. The Android SDK has some nice tools for generating projects and debugging code.

## Implementing Activity Callbacks

---

Android calls a number of callbacks to let you draw your screen, store data before pausing, and refresh data after closing. You must implement at least some of these methods. See [Lifetime of a Screen](#) to learn when and in what order these methods are called. Here are some of the standard types of screen classes that Android provides:

- `android.app.Activity` - This is a standard screen, with no specialization.
- `android.app.ListActivity` - This is a screen that is used to display a list of something. It hosts a `ListView` object, and exposes methods to let you identify the selected item, receive callbacks when the selected item changes, and perform other list-related actions.
- `android.app.Dialog` - This is a small, popup dialog-style window that isn't intended to remain in the history stack. (It is not resizable or moveable by the user.)

## Opening a New Screen

---

Your Activity will often need to open another Activity screen as it progresses. This new screen can be part of the same application or part of another application, the new screen can be floating or full screen, it can return a result, and you can decide whether to close this screen and remove it from the history stack when you are done with it, or to keep the screen open in history. These next sections describe all these options.  
Floating or full?

When you open a new screen you can decide whether to make it transparent or floating, or full-screen. The choice of new screen affects the event sequence of events in the old screen (if the new screen obscures the old screen, a different series of events is called in the old screen). See [Lifetime of an Activity](#) for details.

Transparent or floating windows are implemented in three standard ways:

- Create an `app.Dialog` class
- Create an `app.AlertDialog` class
- Set the `Theme_Dialog` theme attribute to `@android:style/Theme.Dialog` in your `AndroidManifest.xml` file. For example:

```
<activity class="AddRssItem" android:label="Add an item" android:theme="@android:style/Theme.Dialog" />
```

Calling `startActivity()` or `startSubActivity()` will open a new screen in whatever way it defines itself (if it uses a floating theme it will be floating, otherwise it will be full screen).

## Opening a Screen

When you want to open a new screen, you can either explicitly specify the activity class to open, or you can let the operating system decide which screen to open, based upon the data and various parameters you pass in. A screen is opened by calling `startActivity` and passing in an `Intent` object, which specifies the criteria for the handling screen. To specify a specific screen, call `Intent.setClass` or `setClassName` with the exact activity class to open. Otherwise, set a variety of values and data, and let Android decide which screen is appropriate to open. Android will find one or zero Activities that match the specified requirements; it will never open multiple activities for a single request. More information on `Intents` and how Android resolves them to a specific class is given in the [Intent](#) topic.

## Some Intent examples

The following snippet loads the `com.google.android.samples.Animation1` class, and passes it some arbitrary data.:

```

Intent myIntent = new Intent();
myIntent.component = "com.google.android.samples.Animation1";
myIntent.putExtra("com.google.android.samples.SpecialValue", "Hello, Joe!"); // key/value pair, where key needs current pa
startActivity(myIntent);

```

The next snippet requests that a Web page be opened by specifying the VIEW action, and a URI data string starting with "http://" schema:

```

Intent myIntent = new Intent("android.intent.action.VIEW", "http://www.google.com");
myIntent.putExtra("com.google.android.samples.SpecialValue", "Hello, Joe!"); // key/value pair, where key needs current pa

```

Here is the intent filter from the AndroidManifest.xml file for com.google.android.browser:

```

<intent-filter>
  <action value="android.intent.action.VIEW" />
  <category value="android.intent.category.DEFAULT" />
  <scheme value="http" />
  <scheme value="https" />
  <scheme value="file" />
</intent-filter>

```

Android defines a number of standard values, for instance the action constants defined by Intent. You can define custom values, but both the caller and handler must use them. See the <intent-filter> tag description in AndroidManifest.xml File Details for more information on the manifest syntax for the handling application.

## Returning a Result from a Screen

A window can return a result after it closes. This result will be passed back into the calling Activity's onActivityResult() method, which can supply an integer result code, a string of data, and a Bundle of arbitrary data, along with the request code passed to startActivityForResult(). Note that you must call the startActivityForResult() method that accepts a request code parameter to get this callback. The following code demonstrates opening a new screen and retrieving a result.

```

// Open the new screen.
public void onClick(View v){
  // Start the activity whose result we want to retrieve. The
  // result will come back with request code GET_CODE.
  Intent intent = new Intent(this, com.example.app.ChooseYourBoxer.class);
  startActivityForResult(intent, CHOOSE_FIGHTER);
}

// Listen for results.
protected void onActivityResult(int requestCode, int resultCode,
  String data, Bundle extras){
  // See which child activity is calling us back.
  switch (resultCode) {
    case CHOOSE_FIGHTER:
      // This is the standard resultCode that is sent back if the
      // activity crashed or didn't doesn't supply an explicit result.
      if (resultCode == RESULT_CANCELED){
        myMessageboxFunction("Fight cancelled");
      }
      else {
        myFightFunction(data, extras);
      }
    default:
      break;
  }
}

// Class SentResult
// Temporary screen to let the user choose something.
private OnClickListener mLincolnListener = new OnClickListener(){
  public void onClick(View v) {
    Bundle stats = new Bundle();
    stats.putString("height", "6\ '4\ ");
    stats.putString("weight", "190 lbs");
    stats.putString("reach", "74\ ");
    setResult(RESULT_OK, "Lincoln", stats);
    finish();
  }
}

```

```
};

private OnClickListener mWashingtonListener = new OnClickListener() {
    public void onClick(View v){
        Bundle stats = new Bundle();
        stats.putString("height", "6\ '2\ ");
        stats.putString("weight", "190 lbs");
        stats.putString("reach", "73\ ");
        setResult(RESULT_OK, "Washington", Bundle);
        finish();
    }
};
```

## Lifetime of the new screen

An activity can remove itself from the history stack by calling `Activity.finish()` on itself, or the activity that opened the screen can call `Activity.finishSubActivity()` on any screens that it opens to close them.

## Listening for Button Clicks

---

Button click and other UI event capturing are covered in [Listening for UI Notifications](#) on the [UI Design](#) page.

## Configuring General Window Properties

---

You can set a number of general window properties, such as whether to display a title, whether the window is floating, and whether it displays an icon, by calling methods on the `Window` member of the underlying `View` object for the window. Examples include calling `getWindow().requestFeature()` (or the convenience method `requestWindowFeature(some_feature)`) to hide the title. Here is an example of hiding the title bar:

```
//Hide the title bar
requestWindowFeature(Window.FEATURE_NO_TITLE);
```

## Storing and Retrieving State

---

If your application is dumped from memory because of space concerns, it will lose all user interface state information such as checkbox state and text box values as well as class member values. Android calls `Activity.onFreeze` before it pauses the application. This method hands in a `Bundle` that can be used to store name/value pairs that will persist and be handed back to the application even if it is dropped from memory. Android will pass this `Bundle` back to you when it calls `onCreate()`. This `Bundle` only exists while the application is still in the history stack (whether or not it has been removed from memory) and will be lost when the application is finalized. See the topics for `onFreeze(Bundle)` and `onCreate(Bundle)` for examples of storing and retrieving state.

Read more about the life cycle of an application in [Lifetime of an Activity](#).

## Storing and Retrieving Larger or More Complex Persistent Data

Your application can store files or complex collection objects, and reserve them for private use by itself or other activities in the application, or it can expose its data to all other applications on the device. See [Storing, Retrieving, and Exposing Data](#) to learn how to store and retrieve private data, how to store and retrieve common data from the device, and how to expose your private data to other applications.

## Playing Media Files

---

Please see the document [Android Media APIs](#) for more details.

## Listening For and Broadcasting Global Messages, and Setting Alarms

---

You can create a listening class that can be notified or even instantiated whenever a specific type of system message is sent.

The listening classes, called intent receivers, extend `IntentReceiver`. If you want Android to instantiate the object whenever an appropriate intent notification is sent, define the receiver with a `<receiver>` element in the `AndroidManifest.xml` file. If the caller is expected to instantiate the object in preparation to receive a message, this is not required. The receiver will get a call to their `IntentReceiver.onReceiveIntent()` method. A receiver can define an `<intent-filter>` tag that describes the types of messages it will receive. Just as Android's `IntentResolver` will look for appropriate `Activity` matches for a `startActivity()` call, it will look for any matching `Receivers` (but it will send the message to all matching receiver, not the "best" match).

To send a notification, the caller creates an Intent object and calls `Activity.broadcastIntent()` with that Intent. Multiple recipients can receive the same message. You can broadcast an Intent message to an intent receiver in any application, not only your own. If the receiving class is not registered using `<receiver>` in its manifest, you can dynamically instantiate and register a receiver by calling `Context.registerReceiver()`.

Receivers can include intent filters to specify what kinds of intents they are listening for. Alternatively, if you expect a single known caller to contact a single known receiver, the receiver does not specify an intent filter, and the caller specifies the receiver's class name in the Intent by calling `Intent.setClassName()` with the recipient's class name. The recipient receives a Context object that refers to its own package, not to the package of the sender.

Note: If a receiver or broadcaster enforces permissions, your application might need to request permission to send or receive messages from that object. You can request permission by using the `<uses-permission>` tag in the manifest.

Here is a code snippet of a sender and receiver. This example does not demonstrate registering receivers dynamically. For a full code example, see the `AlarmService` class in the `ApiDemos` project.

## Sending the message

```
// We are sending this to a specific recipient, so we will
// only specify the recipient class name.
Intent intent = new Intent(this, AlarmReceiver.class);
intent.putExtra("message", "Wake up.");
broadcastIntent(intent);
```

## Receiving the message

Receiver `AndroidManifest.xml` (because there is no intent filter child, this class will only receive a broadcast when the receiver class is specified by name, as is done in this example):

```
<receiver class=".AlarmReceiver" />
```

Receiver Java code:

```
public class AlarmReceiver extends IntentReceiver{
    // Display an alert that we've received a message.
    @Override
    public void onReceiveIntent(Context context, Intent intent){
        // Send a text notification to the screen.
        NotificationManager nm = (NotificationManager)
        context.getSystemService(Context.NOTIFICATION_SERVICE);
        nm.notifyWithText(R.id.alarm,
            "Alarm!!!",
            NotificationManager.LENGTH_SHORT,
            null);
    }
}
```

## Other system messages

You can listen for other system messages sent by Android as well, such as USB connection/removal messages, SMS arrival messages, and timezone changes. See `Intent` for a list of broadcast messages to listen for. Messages are marked "Broadcast Action" in the documentation.

## Listening for phone events

The telephony package overview page describes how to register to listen for phone events.

## Setting Alarms

Android provides an `AlarmManager` service that will let you specify an Intent to send at a designated time. This intent is typically used to start an application at a preset time. (Note: If you want to send a notification to a sleeping or running application, use `Handler` instead.)

## Displaying Alerts

There are two major kinds of alerts that you may display to the user: (1) Normal alerts are displayed in response to a user action, such as trying to perform an action that is not allowed. (2) Out-of-band alerts, called notifications, are displayed as a result of something happening in the background, such as the user receiving new e-mail.

### Normal Alerts

Android provides a number of ways for you to show popup notifications to your user as they interact with your application.

Class	Description
app.AlertDialog or Context. showAlert()	A popup alert dialog with two buttons (typically OK and Cancel) that take callback handlers. It can be created separately, or launched using the Application helper method Context.showAlert(). See the section after this table for more details.
ProgressDialog	A dialog box used to indicate progress of an operation with a known progress value or an indeterminate length (setProgress(bool)). See Views > Progress Bar in ApiDemos for examples.
Activity	By setting the theme of an activity to android:theme="android:style/Theme.Dialog", your activity will take on the appearance of a normal dialog, floating on top of whatever was underneath it. You usually set the theme through the android:theme attribute in your AndroidManifest.xml. The advantage of this over Dialog and AlertDialog is that Application has a much better managed lifecycle than dialogs: if a dialog goes to the background and is killed, you cannot recapture state, whereas Application exposes a Bundle of saved values in onCreate() to help you maintain state.

### AlertDialog

This is a basic warning dialog box that lets you configure a message, button text, and callback. You can create one by calling the Application helper method Context.showAlert(), as shown here.

```
private Handler mHandler = new Handler() {
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case ACCEPT_CALL:
                answer(msg.obj);
                break;

            case BOUNCE_TO_VOICEMAIL:
                voicemail(msg.obj);
                break;
        }
    }
};

private void IncomingMotherInLawCall(Connection c) {
    String Text;

    // "Answer" callback.
    Message acceptMsg = Message.obtain();
    acceptMsg.target = mHandler;
    acceptMsg.what = ACCEPT_CALL;
    acceptMsg.obj = c.getCall();

    // "Cancel" callback.
    Message rejectMsg = Message.obtain();
    rejectMsg.target = mHandler;
    rejectMsg.what = BOUNCE_TO_VOICEMAIL;
    rejectMsg.obj = c.getCall();

    showAlert(null, "Phyllis is calling", "Answer", acceptMsg, true, rejectMsg);
}
```

### Notifications

Out-of-band alerts should always be displayed using the NotificationManager, which allows you to tell the user about something they may be interested in without disrupting what they are currently doing. A notification can be anything from a brief pop-up box informing the user of the new information, through displaying a persistent icon in the status bar, to vibrating, playing sounds, or flashing lights to get the user's attention. In all cases, the user must explicitly shift their focus to the notification before they can interact with it.

The following code demonstrates using `NotificationManager` to display a basic text popup when a new SMS message arrives in a listening service, and provides the current message count. You can see several more examples in the `ApiDemos` application, under `app/` (named `notification*.java`).

```
static void setNewMessageIndicator(Context context, int messageCount){
    // Get the static global NotificationManager object.
    NotificationManager nm = NotificationManager.getDefault();

    // If we're being called because a new message has been received,
    // then display an icon and a count. Otherwise, delete the persistent
    // message.
    if (messageCount > 0) {
        nm.notifyWithText(myApp.NOTIFICATION_GUID, // ID for this notification.
            messageCount + " new message" + messageCount > 1 ? "s":"", // Text to display.
            NotificationManager.LENGTH_SHORT); // Show it for a short time only.
    }
}
```

To display a notification in the status bar and have it launch an intent when the user selects it (such as the new text message notification does), call `NotificationManager.notify()`, and pass in vibration patterns, status bar icons, or Intents to associate with the notification.

## Displaying a Progress Bar

An activity can display a progress bar to notify the user that something is happening. To display a progress bar in a screen, call `Activity.requestWindowFeature(Window.FEATURE_PROGRESS)`. To set the value of the progress bar, call `Activity.getWindow().setFeatureInt(Window.FEATURE_PROGRESS, level)`. Progress bar values are from 0 to 9,999, or set the value to 10,000 to make the progress bar invisible.

You can also use the `ProgressDialog` class, which enables a dialog box with an embedded progress bar to send a "I'm working on it" notification to the user.

## Adding Your Application to the Favorites List

You can't. Only a user can add an application to the Favorites list.

## Adding Items to the Screen Menu

Every Android screen has a default menu with default options, such as adding the activity to the favorites menu. You can add your own menu entries to the default menu options by implementing `Activity.onCreateOptionsMenu` or `Activity.onPrepareOptionsMenu()`, and adding `Item` objects to the `Menu` passed in. To handle clicks implement `Activity.onOptionsItemSelected()` to handle the click in your `Activity` class. You may also pass the `Item` object a handler class that implements the `Runnable` class (a handler) but this is less efficient and discouraged.

An application receives a callback at startup time to enable it to populate its menu. Additionally, it receives callbacks each time the user displays the options menu to let you perform some contextual modifications on the menu. To populate the menu on startup, override `Activity.onCreateOptionsMenu`; to populate it when the menu is called (somewhat less efficient), you can override `Activity.onPrepareOptionsMenu()`. Each `Activity` has its own menu list.

Menu items are displayed in the order added, though you can group them as described in the `Menu.add` documentation. The following code snippet adds three items to the default menu options and handles them through the overridden `Activity.onOptionsItemSelected()` method. You can show or hide menu items by calling `setItemShown()` or `setGroupShown()`.

```
// Called only the first time the options menu is displayed.
// Create the menu entries.
// Menu adds items in the order shown.
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);

    // Parameters for menu.add are:
    // group -- Not used here.
    // id -- Used only when you want to handle and identify the click yourself.
    // title
    menu.add(0, 0, "Zoom");
    menu.add(0, 1, "Settings");
}
```

```

        menu.add(0, 2, "Other");
        return true;
    }

    // Activity callback that lets your handle the selection in the class.
    // Return true to indicate that you've got it, false to indicate
    // that it should be handled by a declared handler object for that
    // item (handler objects are discouraged for reasons of efficiency).
    @Override
    public boolean onOptionsItemSelected(Menu.Item item){
        switch (item.getId()) {
            case 0:
                showAlert("Menu Item Clicked", "Zoom", "ok", null, false, null);
                return true;
            case 1:
                showAlert("Menu Item Clicked", "Settings", "ok", null, false, null);
                return true;
            case 2:
                showAlert("Menu Item Clicked", "Other", "ok", null, false, null);
                return true;
        }
        return false;
    }
}

```

You can add key shortcuts by calling the `Item.setAlphabeticShortcut()` or `Item.setNumericShortcut()` methods, as demonstrated here to add a "C" shortcut to a menu item:

```
thisItem.setAlphabeticShortcut(0, 'c');
```

## Adding Submenus

Add a submenu by calling `Menu.addSubMenu()`, which returns a `SubMenu` object. You can then add additional items to this menu. Menus can only be one level deep, and you can customize the appearance of the submenu menu item.

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);

    // Parameters for menu.add are:
    // group -- Not used here.
    // id -- Used only when you want to handle and identify the click yourself.
    // title
    menu.add(0, 0, "Send message");
    menu.add(0, 1, "Settings");
    menu.add(0, 2, "Local handler");
    menu.add(0, 3, "Launch contact picker");

    // Add our submenu.
    SubMenu sub = menu.addSubMenu(1, 4, "Days of the week");
    sub.add(0, 5, "Monday");
    sub.add(0, 6, "Tuesday");
    sub.add(0, 7, "Wednesday");
    sub.add(0, 8, "Thursday");
    sub.add(0, 9, "Friday");
    sub.add(0, 10, "Saturday");
    sub.add(0, 11, "Sunday");
    return true;
}

```

## Adding yourself to menus on other applications

You can also advertise your Activity's services so that other Activities can add your activity to their own option menu. For example, suppose you implement a new image handling tool that shrinks an image to a smaller size and you would like to offer this as a menu option to any other Activity that handles pictures. To do this, you would expose your capabilities inside an intent filter in your manifest. If another application that handles photos asks Android for any Activities that can perform actions on pictures, Android will perform intent resolution, find your Activity, and add it to the other Activity's options menu.

## The offering application

The application offering the service must include an `<intent-filter>` element in the manifest, inside the `<activity>` tag of the offering Activity. The intent filter includes all the details describing what it can do, such as a `<type>` element that describes the MIME type of data that it can handle, a custom `<action>` value that describes what your handling application can do (this is so that when it receives the Intent on opening it knows what it is expected to do), and most important, include a `<category>` filter with the value `android.intent.category.ALTERNATIVE` and/or `android.intent.category.SELECTED_ALTERNATIVE` (`SELECTED_ALTERNATIVE` is used to handle only the currently selected element on the screen, rather than the whole Activity intent).

Here's an example of a snip of a manifest that advertises picture shrinking technology for both selected items and the whole screen.

```
<activity class="PictureShrink">          <!-- Handling class -->
  <intent-filter label="Shrink picture">    <!-- Menu label to display -->
    <action value="com.example.sampleapp.SHRINK_IT" />
    <type value="image/*" />              <!-- MIME type for generic images -->
    <category value="android.intent.category.ALTERNATIVE " />
    <category value="android.intent.category.SELECTED_ALTERNATIVE" />
  </intent-filter>
</activity>
```

The menu-displaying application

An application that wants to display a menu that includes any additional external services must, first of all, handle its menu creation callback. As part of that callback it creates an intent with the category `Intent.ALTERNATIVE_CATEGORY` and/or `Intent.SELECTED_ALTERNATIVE`, the MIME type currently selected, and any other requirements, the same way as it would satisfy an intent filter to open a new Activity. It then calls `menu.addIntentOptions()` to have Android search for and add any services meeting those requirements. It can optionally add additional custom menu items of its own.

You should implement `SELECTED_ALTERNATIVE` in `onPrepareOptionsMenu()` rather than `onCreateOptionsMenu()`, because the user's selection can change after the application is launched.

Here's a code snippet demonstrating how a picture application would search for additional services to display on its menu.

```
@Override public boolean
onCreateOptionsMenu(Menu menu){
    super.onCreateOptionsMenu(menu);

    // Create an Intent that describes the requirements to fulfill to be included
    // in our menu. The offering app must include a category value of Intent.ALTERNATIVE_CATEGORY.
    Intent intent = new Intent(null, getIntent().getData());
    intent.addCategory(Intent.ALTERNATIVE_CATEGORY);

    // Search for, and populate the menu with, acceptable offering applications.
    menu.addIntentOptions(
        0, // Group
        0, // Any unique IDs we might care to add.
        MySampleClass.class.getName(), // Name of the class displaying the menu--here, its this class.
        null, // No specifics.
        intent, // Previously created intent that describes our requirements.
        0, // No flags.
        null); // No specifics.

    return true;
}
```

## Display a Web Page

---

Use the `webkit.WebView` object.

## Binding to Data

---

You can bind a `ListView` to a set of underlying data by using a shim class called `ListAdapter` (or a subclass). `ListAdapter` subclasses bind to a variety of data sources, and expose a common set of methods such as `getItem()` and `getView()`, and uses them to pick View items to display in its list. You can extend `ListAdapter` and override `getView()` to create your own custom list items. There are essentially only two steps you need to perform to bind to data:

1. Create a `ListAdapter` object and specify its data source
2. Give the `ListAdapter` to your `ListView` object.

That's it!

Here's an example of binding a ListActivity screen to the results from a cursor query. (Note that the setListAdapter() method shown is a convenience method that gets the page's ListView object and calls setAdapter() on it.)

```
// Run a query and get a Cursor pointing to the results.
Cursor c = People.query(this.getContentResolver(), null);
startManagingCursor(c);

// Create the ListAdapter. A SimpleCursorAdapter lets you specify two interesting things:
// an XML template for your list item, and
// The column to map to a specific item, by ID, in your template.
ListAdapter adapter = new SimpleCursorAdapter(this,
    android.R.layout.simple_list_item_1, // Use a template that displays a text view
    c, // Give the cursor to the list adapter
    new String[] {People.NAME} , // Map the NAME column in the people database to...
    new String[] {"text1"}); // The "text1" view defined in the XML template
setListAdapter(adapter);
```

See view/List4 in the ApiDemos project for an example of extending ListAdapter for a new data type.

## Capture Images from the Phone Camera

You can hook into the device's camera onto your own Canvas object by using the CameraDevice class. See that class's documentation, and the ApiDemos project's Camera Preview application (Graphics/Camera Preview) for example code.

## Handling Expensive Operations in the UI Thread

Avoid performing long-running operations (such as network I/O) directly in the UI thread — the main thread of an application where the UI is run — or your application may be blocked and become unresponsive. Here is a brief summary of the recommended approach for handling expensive operations:

1. Create a Handler object in your UI thread
2. Spawn off worker threads to perform any required expensive operations
3. Post results from a worker thread back to the UI thread's handler either through a Runnable or a Message
4. Update the views on the UI thread as needed

The following outline illustrates a typical implementation:

```
public class MyActivity extends Activity {

    [ . . . ]
    // Need handler for callbacks to the UI thread
    final Handler mHandler = new Handler();

    // Create runnable for posting
    final Runnable mUpdateResults = new Runnable() {
        @Override public void run() {
            updateResultsInUi();
        }
    };

    @Override
    protected void onCreate(Bundle icle) {
        super.onCreate(icle);

        [ . . . ]
    }

    protected void startLongRunningOperation() {

        // Fire off a thread to do some work that we shouldn't do directly in the UI thread
        Thread t = new Thread() {
            public void run() {
                mResults = doSomethingExpensive();
                mHandler.post(mUpdateResults);
            }
        };
        t.start();
    }

    private void updateResultsInUi() {
```

```

        // Back in the UI thread -- update our UI elements based on the data in mResults
        [ . . . ]
    }
}

```

For further discussions on this topic, see [Developing Responsive Applications](#) and the [Handler](#) documentation.

## Selecting, Highlighting, or Styling Portions of Text

You can highlight or style the formatting of strings or substrings of text in a `TextView` object. There are two ways to do this:

- If you use a string resource, you can add some simple styling, such as bold or italic using HTML notation. So, for example, in `res/values/strings.xml` you could declare this:

```

<resource>
    <string id="@+id/styled_welcome_message">We are <b><i>so</i></b> glad to see you.</string>
</resources>

```

- To style text on the fly, or to add highlighting or more complex styling, you must use the `Spannable` object as described next.

To style text on the fly, you must make sure the `TextView` is using `Spannable` storage for the text (this will always be true if the `TextView` is an `EditText`), retrieve its text with `getText()`, and call `setSpan(Object, int, int, int)`, passing in a new style class from the `android.text.style` package and the selection range.

The following code snippet demonstrates creating a string with a highlighted section, italic section, and bold section, and adding it to an `EditText` object.

```

// Get our EditText object.
EditText vw = (EditText)findViewById(R.id.text);

// Set the EditText's text.
vw.setText("Italic, highlighted, bold.");

// If this were just a TextView, we could do:
// vw.setText("Italic, highlighted, bold.", TextView.BufferType.SPANNABLE);
// to force it to use Spannable storage so styles can be attached.
// Or we could specify that in the XML.

// Get the EditText's internal text storage
Spannable str = vw.getText();

// Create our span sections, and assign a format to each.
str.setSpan(new StyleSpan(android.graphics.Typeface.ITALIC), 0, 7, Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
str.setSpan(new BackgroundColorSpan(0xFFFFF00), 8, 19, Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
str.setSpan(new StyleSpan(android.graphics.Typeface.BOLD), 21, str.length() - 1, Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);

```

## List of Files for an Android Application

The following list describes the structure and files of an Android application. Many of these files can be built for you (or stubbed out) by the `activityCreator.py` application shipped in the `tools/` menu of the SDK. See [Building an Android Sample Application](#) for more information on using `activityCreator.py`.

MyApp/	
AndroidManifest.xml	(required) Advertises the screens that this application provides, where they can be launched (from the main program menu or elsewhere), any content providers it implements and what kind of data they handle, where the implementation classes are, and other application-wide information. Syntax details for this file are described in <code>AndroidManifest.xml</code> .
src/ /myPackagePath/.../MyClass.java	(required) This folder holds all the source code files for your application, inside the appropriate package subfolders.
res/	(required) This folder holds all the resources for your application. Resources are external data files or description files that are compiled into your code at build time. Files in different folders are compiled differently, so you must put the proper resource into the proper folder. (See <a href="#">Resources</a> for details.)

anim/ animation1.xml ...	(optional) Holds any animation XML description files that the application uses. The format of these files is described in Resources.
drawable/ some_picture.png some_stretchable.9.png some_background.xml ...	(optional) Zero or more files that will be compiled to android.graphics.drawable resources. Files can be image files (png, gif, or other) or XML files describing other graphics such as bitmaps, stretchable bitmaps, or gradients. Supported bitmap file formats are PNG (preferred), JPG, and GIF (discouraged), as well as the custom 9-patch stretchable bitmap format. These formats are described in Resources.
layout/ screen_1_layout.xml ...	(optional) Holds all the XML files describing screens or parts of screens. Although you could create a screen in Java, defining them in XML files is typically easier. A layout file is similar in concept to an HTML file that describes the screen layout and components. See Implementing a UI for more information about designing screens, and Layout Resources for the syntax of these files.
values/ arrays classes.xml colors.xml dimens.xml strings.xml styles.xml values.xml	(optional) XML files describing additional resources such as strings, colors, and styles. The naming, quantity, and number of these files are not enforced--any XML file is compiled, but these are the standard names given to these files. However, the syntax of these files is prescribed by Android, and described in Resources.
xml/	(optional) XML files that can be read at run time on the device.
raw/	(optional) Any files to be copied directly to the device.