Getting Dates and Times from Users

The Android SDK provides a couple controls for getting date and time input from the user. The first is the `DatePicker` control (Figure 7.8, top). It can be used to get a month, day, and year from the user.

Figure 7.8 Date and time controls.

The basic XML layout resource definition for a `DatePicker` follows:

```
<DatePicker
    android:id="@+id/DatePicker01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

As you can see from this example, there aren't any attributes specific to the `DatePicker` control. As with many of the other controls, your code can register to receive a method call when the date changes. You do this by implementing the `onDateChanged()` method. However, this isn't done the usual way.

```
final DatePicker date = (DatePicker)findViewById(R.id.DatePicker01);
date.init(date.getYear(), date.getMonth(), date.getDayOfMonth(),
    new DatePicker.OnDateChangedListener() {
        public void onDateChanged(DatePicker view, int year,
            int monthOfYear, int dayOfMonth) {
                Date dt = new Date(year-1900,
```

```
                    monthOfYear, dayOfMonth, time.getCurrentHour(),
                    time.getCurrentMinute());
                text.setText(dt.toString());
        }
    });
```

The preceding code sets the `DatePicker.OnDateChangedListener` by a call to the `DatePicker.init()` method. A `DatePicker` control is initialized with the current date. A `TextView` is set with the date value that the user entered into the `DatePicker` control. The value of 1900 is subtracted from the year parameter to make it compatible with the `java.util.Date` class.

A `TimePicker` control (also shown in Figure 7.8, bottom) is similar to the `DatePicker` control. It also doesn't have any unique attributes. However, to register for a method call when the values change, you call the more traditional method of `TimePicker.setOnTimeChangedListener()`.

```
time.setOnTimeChangedListener(new TimePicker.OnTimeChangedListener() {
    public void onTimeChanged(TimePicker view,
        int hourOfDay, int minute) {
            Date dt = new Date(date.getYear()-1900, date.getMonth(),
                date.getDayOfMonth(), hourOfDay, minute);
                text.setText(dt.toString());
    }
});
```

As in the previous example, this code also sets a `TextView` to a string displaying the time value that the user entered. When you use the `DatePicker` control and the `TimePicker` control together, the user can set a full date and time.

Using Indicators to Display Data to Users

The Android SDK provides a number of controls that can be used to visually show some form of information to the user. These indicator controls include progress bars, clocks, and other similar controls.

Indicating Progress with `ProgressBar`

Applications commonly perform actions that can take a while. A good practice during this time is to show the user some sort of progress indicator that informs the user that the application is off "doing something." Applications can also show how far a user is through some operation, such as a playing a song or watching a video. The Android SDK provides several types of progress bars.

The standard progress bar is a circular indicator that only animates. It does not show how complete an action is. It can, however, show that something is taking place. This is useful when an action is indeterminate in length. There are three sizes of this type of progress indicator (see Figure 7.9).

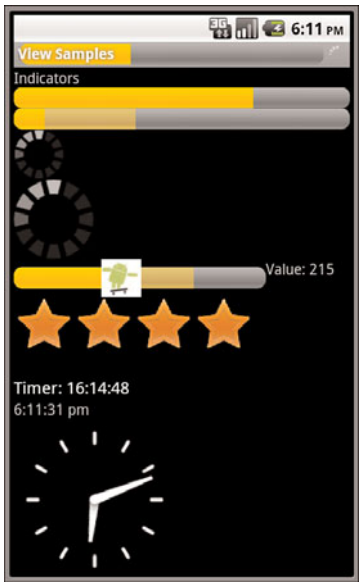

Figure 7.9 Various types of progress and rating indicators.

The second type is a horizontal progress bar that shows the completeness of an action. (For example, you can see how much of a file is downloading.) This horizontal progress bar can also have a secondary progress indicator on it. This can be used, for instance, to show the completion of a downloading media file while that file plays.

This is an XML layout resource definition for a basic indeterminate progress bar:

```
<ProgressBar
    android:id="@+id/progress_bar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

The default style is for a medium-size circular progress indicator; not a "bar" at all. The other two styles for indeterminate progress bar are `progressBarStyleLarge` and `progressBarStyleSmall`. This style animates automatically. The next sample shows the layout definition for a horizontal progress indicator:

```
<ProgressBar
    android:id="@+id/progress_bar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:max="100" />
```

We have also set the attribute for `max` in this sample to 100. This can help mimic a percentage progress bar. That is, setting the progress to 75 shows the indicator at 75 percent complete.

We can set the indicator progress status programmatically as follows:

```
mProgress = (ProgressBar) findViewById(R.id.progress_bar);
mProgress.setProgress(75);
```

You can also put these progress bars in your application's title bar (as shown in Figure 7.9). This can save screen real estate. This can also make it easy to turn on and off an indeterminate progress indicator without changing the look of the screen. Indeterminate progress indicators are commonly used to display progress on pages where items need to be loaded before the page can finish drawing. This is often employed on web browser screens. The following code demonstrates how to place this type of indeterminate progress indicator on your `Activity` screen:

```
requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
requestWindowFeature(Window.FEATURE_PROGRESS);
setContentView(R.layout.indicators);
setProgressBarIndeterminateVisibility(true);
setProgressBarVisibility(true);
setProgress(5000);
```

To use the indeterminate indicator on your `Activity` objects title bar, you need to request the feature `Window.FEATURE_INDETERMINATE_PROGRESS`, as previously shown. This shows a small circular indicator in the right side of the title bar. For a horizontal progress bar style that shows behind the title, you need to enable the `Window.FEATURE_PROGRESS`. These features must be enabled before your application calls the `setContentView()` method, as shown in the preceding example.

You need to know about a couple of important default behaviors. First, the indicators are visible by default. Calling the visibility methods shown in the preceding example can set their visibility on or off. Second, the horizontal progress bar defaults to a maximum progress value of 10,000. In the preceding example, we set it to 5,000, which is equivalent to 50 percent. When the value reaches the maximum value, the indicators fade away so that they aren't visible. This happens for both indicators.

Adjusting Progress with `SeekBar`

You have seen how to display progress to the user. What if, however, you want to give the user some ability to move the indicator, for example, to set the current cursor position in a playing media file or to tweak a volume setting? You accomplish this by using the `SeekBar` control provided by the Android SDK. It's like the regular horizontal progress bar, but includes a thumb, or selector, that can be dragged by the user. A default thumb selector is provided, but you can use any `drawable` item as a thumb. In Figure 7.9 (center), we replaced the default thumb with a little Android graphic.

Here we have an example of an XML layout resource definition for a simple `SeekBar`:

```
<SeekBar
    android:id="@+id/seekbar1"
    android:layout_height="wrap_content"
    android:layout_width="240px"
    android:max="500" />
```

With this sample `SeekBar`, the user can drag the thumb to any value between 0 and 500. Although this is shown visually, it might be useful to show the user what exact value the user is selecting. To do this, you can provide an implementation of the `onProgressChanged()` method, as shown here:

```
SeekBar seek = (SeekBar) findViewById(R.id.seekbar1);
seek.setOnSeekBarChangeListener(
    new SeekBar.OnSeekBarChangeListener() {
        public void onProgressChanged(
            SeekBar seekBar, int progress,boolean fromTouch) {
            ((TextView)findViewById(R.id.seek_text))
                .setText("Value: "+progress);
            seekBar.setSecondaryProgress(
                (progress+seekBar.getMax())/2);
        }
});
```

There are two interesting things to notice in this example. The first is that the `fromTouch` parameter tells the code if the change came from the user input or if, instead, it came from a programmatic change as demonstrated with the regular `ProgressBar` controls. The second interesting thing is that the `SeekBar` still enables you to set a secondary progress value. In this example, we set the secondary indicator to be halfway between the user's selected value and the maximum value of the progress bar. You might use this feature to show the progress of a video and the buffer stream.

Displaying Rating Data with `RatingBar`

Although the `SeekBar` is useful for allowing a user to set a value, such as the volume, the `RatingBar` has a more specific purpose: showing ratings or getting a rating from a user. By default, this progress bar uses the star paradigm with five stars by default. A user can drag across this horizontal to set a rating. A program can set the value, as well. However, the secondary indicator cannot be used because it is used internally by this particular control.

Here's an example of an XML layout resource definition for a `RatingBar` with four stars:

```
<RatingBar
    android:id="@+id/ratebar1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:numStars="4"
    android:stepSize="0.25" />
```

This layout definition for a RatingBar demonstrates setting both the number of stars and the increment between each rating value. Here, users can choose any rating value between 0 and 4.0, but only in increments of 0.25, the stepSize value. For instance, users can set a value of 2.25. This is visualized to the users, by default, with the stars partially filled. Figure 7.9 (center) illustrates how the RatingBar behaves.

Although the value is indicated to the user visually, you might still want to show a numeric representation of this value to the user. You can do this by implementing the onRatingChanged() method of the RatingBar.OnRatingBarChangeListener class.

```
RatingBar rate = (RatingBar) findViewById(R.id.ratebar1);
rate.setOnRatingBarChangeListener(new
    RatingBar.OnRatingBarChangeListener() {
        public void onRatingChanged(RatingBar ratingBar,
            float rating, boolean fromTouch) {
                ((TextView)findViewById(R.id.rating_text))
                    .setText("Rating: "+ rating);
    }
});
```

The preceding example shows how to register the listener. When the user selects a rating using the control, a TextView is set to the numeric rating the user entered. One interesting thing to note is that, unlike the SeekBar, the implementation of the onRatingChange() method is called after the change is complete, usually when the user lifts a finger. That is, while the user is dragging across the stars to make a rating, this method isn't called. It is called when the user stops pressing the control.

Showing Time Passage with the Chronometer

Sometimes you want to show time passing instead of incremental progress. In this case, you can use the Chronometer control as a timer (see Figure 7.9, bottom). This might be useful if it's the user who is taking time doing some task or in a game where some action needs to be timed. The Chronometer control can be formatted with text, as shown in this XML layout resource definition:

```
<Chronometer
    android:id="@+id/Chronometer01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:format="Timer: %s" />
```

You can use the Chronometer object's format attribute to put text around the time that displays. A Chronometer won't show the passage of time until its start() method is called. To stop it, simply call its stop() method. Finally, you can change the time from which the timer is counting. That is, you can set it to count from a particular time in the past instead of from the time it's started. You call the setBase() method to do this.

> **Tip**
>
> The `Chronometer` uses the `elapsedRealtime()` method's time base. Passing `android.os.SystemClock.elapsedRealtime()` in to the `setBase()` method starts the `Chronometer` control at 0.

In this next example code, the timer is retrieved from the `View` by its resource identifier. We then check its base value and set it to 0. Finally, we start the timer counting up from there.

```
final Chronometer timer =
    (Chronometer)findViewById(R.id.Chronometer01);
long base =  timer.getBase();
Log.d(ViewsMenu.debugTag, "base = "+ base);
timer.setBase(0);
timer.start();
```

> **Tip**
>
> You can listen for changes to the `Chronometer` by implementing the `Chronometer.OnChronometerTickListener` interface.

Displaying the Time

Displaying the time in an application is often not necessary because Android devices have a status bar to display the current time. However, there are two clock controls available to display this information: the `DigitalClock` and `AnalogClock` controls.

Using the `DigitalClock`

The `DigitalClock` control (Figure 7.9, bottom) is a compact text display of the current time in standard numeric format based on the users' settings. It is a `TextView`, so anything you can do with a `TextView` you can do with this control, except change its text. You can change the color and style of the text, for example.

By default, the `DigitalClock` control shows the seconds and automatically updates as each second ticks by. Here is an example of an XML layout resource definition for a `DigitalClock` control:

```
<DigitalClock
    android:id="@+id/DigitalClock01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Using the `AnalogClock`

The `AnalogClock` control (Figure 7.9, bottom) is a dial-based clock with a basic clock face with two hands, one for the minute and one for the hour. It updates automatically as each minute passes. The image of the clock scales appropriately with the size of its `View`.

Here is an example of an XML layout resource definition for an `AnalogClock` control:

```
<AnalogClock
    android:id="@+id/AnalogClock01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

The `AnalogClock` control's clock face is simple. However you can set its minute and hour hands. You can also set the clock face to specific `drawable` resources, if you want to jazz it up. Neither of these clock controls accepts a different time or a static time to display. They can show only the current time in the current time zone of the device, so they are not particularly useful.

Providing Users with Options and Context Menus

You need to be aware of two special application menus for use within your Android applications: the options menu and the context menu.

Enabling the Options Menu

The Android SDK provides a method for users to bring up a menu by pressing the menu key from within the application (see Figure 7.10). You can use options menus within your application to bring up help, to navigate, to provide additional controls, or to configure options. The `OptionsMenu` control can contain icons, submenus, and keyboard shortcuts.

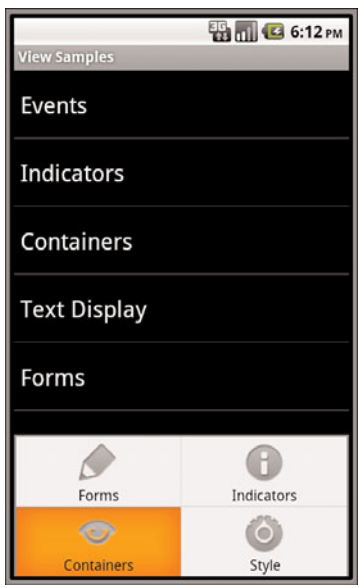

Figure 7.10 An options menu.

For an options menu to show when a user presses the `Menu` button on their device, you need to override the implementation of `onCreateOptionsMenu()` in your `Activity`. Here is a sample implementation that gives the user three menu items to choose from:

```
public boolean onCreateOptionsMenu( android.view.Menu menu) {
    super.onCreateOptionsMenu(menu);
    menu.add("Forms")
        .setIcon(android.R.drawable.ic_menu_edit)
        .setIntent(new Intent(this, FormsActivity.class));
    menu.add("Indicators")
        .setIntent(new Intent(this, IndicatorsActivity.class))
        .setIcon(android.R.drawable.ic_menu_info_details);
    menu.add("Containers")
        .setIcon(android.R.drawable.ic_menu_view)
        .setIntent(new Intent(this, ContainersActivity.class));
    return true;
}
```

For each of the items that are added, we also set a built-in icon resource and assign an `Intent` to each item. We give the item title with a regular text string, for clarity. You can use a resource identifier, as well. For this example, there is no other handling or code needed. When one of these menu items is selected, the `Activity` described by the `Intent` starts.

This type of options menu can be useful for navigating to important parts of an application, such as the help page, from anywhere within your application. Another great use for an options menu is to allow configuration options for a given screen. The user can configure these options in the form of checkable menu items. The initial menu that appears when the user presses the menu button does not support checkable menu items. Instead, you must place these menu items on a `SubMenu` control, which is a type of `Menu` that can be configured within a menu. `SubMenu` objects support checkable items but do not support icons or other `SubMenu` items. Building on the preceding example, the following is code for programmatically adding a `SubMenu` control to the previous `Menu`:

```
SubMenu style_choice = menu.addSubMenu("Style")
    .setIcon(android.R.drawable.ic_menu_preferences);
style_choice.add(style_group, light_id, 1, "Light")
    .setChecked(isLight);
style_choice.add(style_group, dark_id, 2, "Dark")
    .setChecked(!isLight);
style_choice.setGroupCheckable(style_group, true, true);
```

This code would be inserted before the return statement in the implementation of the `onCreateOptionsMenu()` method. It adds a single menu item with an icon to the previous menu, called "Style." When the "Style" option is clicked, a pop-up menu with the two items of the `SubMenu` control is displayed. These items are grouped together and the checkable icon, by default, looks like the radio button icon. The checked state is assigned during creation time.

To handle the event when a menu option item is selected, we also implement the `onOptionsItemSelected()` method, as shown here:

```
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == light_id) {
        item.setChecked(true);
        isLight = true;
        return true;
    } else if (item.getItemId() == dark_id) {
        item.setChecked(true);
        isLight = false;
        return true;
    }

    return super.onOptionsItemSelected(item);
}
```

This method must call the super class's `onOptionsItemSelected()` method for basic behavior to work. The actual `MenuItem` object is passed in, and we can use that to retrieve the identifier that we previously assigned to see which one was selected and perform an appropriate action. Here, we switch the values and return. By default, a `Menu` control goes away when any item is selected, including checkable items. This means it's useful for quick settings but not as useful for extensive settings where the user might want to change more than one item at a time.

As you add more menu items to your options menu, you might notice that a "More" item automatically appears. This happens whenever more than six items are visible. If the user selects this, the full menu appears. The full, expanded menu doesn't show menu icons and although checkable items are possible, you should not use them here. Additionally, the full title of an item doesn't display. The initial menu, also known as the icon menu, shows only a portion of the title for each item. You can assign each item a `condensedTitle` attribute, which shows instead of a truncated version of the regular title. For example, instead of the title Instant Message, you can set the `condensedTitle` attribute to "IM."

Enabling the `ContextMenu`

The `ContextMenu` is a subtype of `Menu` that you can configure to display when a long press is performed on a `View`. As the name implies, the `ContextMenu` provides for contextual menus to display to the user for performing additional actions on selected items.

`ContextMenu` objects are slightly more complex than `OptionsMenu` objects. You need to implement the `onCreateContextMenu()` method of your `Activity` for one to display. However, before that is called, you must call the `registerForContextMenu()` method and pass in the `View` for which you want to have a context menu. This means each `View` on your screen can have a different context menu, which is appropriate as the menus are designed to be highly contextual.

Here we have an example of a `Chronometer` timer, which responds to a long click with a context menu:

```
registerForContextMenu(timer);
```

After the call to the `registerForContextMenu()` method has been executed, the user can then long click on the `View` to open the context menu. Each time this happens, your `Activity` gets a call to the `onCreateContextMenu()` method, and your code creates the menu each time the user performs the long click.

The following is an example of a context menu for the `Chronometer` control, as previously used:

```
public void onCreateContextMenu(
    ContextMenu menu, View v, ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);

    if (v.getId() == R.id.Chronometer01) {
        getMenuInflater().inflate(R.menu.timer_context, menu);
        menu.setHeaderIcon(android.R.drawable.ic_media_play)
            .setHeaderTitle("Timer controls");
    }
}
```

Recall that any `View` control can register to trigger a call to the `onCreateContextMenu()` method when the user performs a long press. That means we have to check which `View` control it was for which the user tried to get a context menu. Next, we inflate the appropriate menu from a menu resource that we defined with XML. Because we can't define header information in the menu resource file, we set a stock Android SDK resource to it and add a title. Here is the menu resource that is inflated:

```
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/start_timer"
        android:title="Start" />
    <item
        android:id="@+id/stop_timer"
        android:title="Stop" />
    <item
        android:id="@+id/reset_timer"
        android:title="Reset" />
</menu>
```

This defines three menu items. If this weren't a context menu, we could have assigned icons. However, context menus do not support icons, submenus, or shortcuts. For more information on setting `Menu` resources in XML, see Chapter 6.

Now we need to handle the `ContextMenu` clicks by implementing the `onContextItemSelected()` method in our `Activity`. Here's an example:

```
public boolean onContextItemSelected(MenuItem item) {
    super.onContextItemSelected(item);
    boolean result = false;
    Chronometer timer = (Chronometer)findViewById(R.id.Chronometer01);
    switch (item.getItemId()){
        case R.id.stop_timer:
            timer.stop();
            result = true;
            break;
        case R.id.start_timer:
            timer.start();
            result = true;
            break;
        case R.id.reset_timer:
            timer.setBase(SystemClock.elapsedRealtime());
            result = true;
            break;
    }
    return result;
}
```

Because we have only one context menu in this example, we find the `Chronometer` view for use in this method. This method is called regardless of which context menu the selected item is on, though, so you should take care to have unique resource identifiers or keep track of which menu is shown. This can be accomplished because the context menu is created each time it's shown.

Handling User Events

You've seen how to do basic event handling in some of the previous control examples. For instance, you know how to handle when a user clicks on a button. There are a number of other events generated by various actions the user might take. This section briefly introduces you to some of these events. First, though, we need to talk about the input states within Android.

Listening for Touch Mode Changes

The Android screen can be in one of two states. The state determines how the focus on `View` controls is handled. When touch mode is on, typically only objects such as `EditText` get focus when selected. Other objects, because they can be selected directly by the user tapping on the screen, won't take focus but instead trigger their action, if any. When not in touch mode, however, the user can change focus between even more object types. These include buttons and other views that normally need only a click to trigger their action. In

this case, the user uses the arrow keys, trackball, or wheel to navigate between items and select them with the Enter or select keys.

Knowing what mode the screen is in is useful if you want to handle certain events. If, for instance, your application relies on the focus or lack of focus on a particular control, your application might need to know if the phone is in touch mode because the focus behavior is likely different.

Your application can register to find out when the touch mode changes by using the `addOnTouchModeChangeListener()` method within the `android.view.ViewTreeObserver` class. Your application needs to implement the `ViewTreeObserver.OnTouchModeChangeListener` class to listen for these events. Here is a sample implementation:

```
View all = findViewById(R.id.events_screen);
ViewTreeObserver vto = all.getViewTreeObserver();
vto.addOnTouchModeChangeListener(
    new ViewTreeObserver.OnTouchModeChangeListener() {
        public void onTouchModeChanged(
            boolean isInTouchMode) {
            events.setText("Touch mode: " + isInTouchMode);
        }
});
```

In this example, the top-level `View` in the layout is retrieved. A `ViewTreeObserver` listens to a `View` and all its child `View` objects. Using the top-level `View` of the layout means the `ViewTreeObserver` listens to events within the entire layout. An implementation of the `onTouchModeChanged()` method provides the `ViewTreeObserver` with a method to call when the touch mode changes. It merely passes in which mode the `View` is now in.

In this example, the mode is written to a `TextView` named `events`. We use this same `TextView` in further event handling examples to visually show on the screen which events our application has been told about. The `ViewTreeObserver` can enable applications to listen to a few other events on an entire screen.

By running this sample code, we can demonstrate the touch mode changing to true immediately when the user taps on the touch screen. Conversely, when the user chooses to use any other input method, the application reports that touch mode is false immediately after the input event, such as a key being pressed or the trackball or scroll wheel moving.

Listening for Events on the Entire Screen

You saw in the last section how your application can watch for changes to the touch mode state for the screen using the `ViewTreeObserver` class. The `ViewTreeObserver` also provides three other events that can be watched for on a full screen or an entire `View` and all of its children. These are

- **`PreDraw`**: Get notified before the `View` and its children are drawn
- **`GlobalLayout`**: Get notified when the layout of the `View` and its children might change, including visibility changes

- **`GlobalFocusChange`**: Get notified when the focus within the `View` and its children changes

Your application might want to perform some actions before the screen is drawn. You can do this by calling the method `addOnPreDrawListener()` with an implementation of the `ViewTreeObserver.OnPreDrawListener` class interface.

Similarly, your application can find out when the layout or visibility of a `View` has changed. This might be useful if your application is dynamically changing the display contents of a view and you want to check to see if a `View` still fits on the screen. Your application needs to provide an implementation of the `ViewTreeObserver.OnGlobalLayoutListener` class interface to the `addGlobalLayoutListener()` method of the `ViewTreeObserver` object.

Finally, your application can register to find out when the focus changes between a `View` control and any of its child `View` controls. Your application might want to do this to monitor how a user moves about on the screen. When in touch mode, though, there might be fewer focus changes than when the touch mode is not set. In this case, your application needs to provide an implementation of the `ViewTreeObserver.OnGlobalFocusChangeListener` class interface to the `addGlobalFocusChangeListener()` method. Here is a sample implementation of this:

```
vto.addOnGlobalFocusChangeListener(new
    ViewTreeObserver.OnGlobalFocusChangeListener() {
        public void onGlobalFocusChanged(
            View oldFocus, View newFocus) {
                if (oldFocus != null && newFocus != null) {
                    events.setText("Focus \nfrom: " +
                        oldFocus.toString() + " \nto: " +
                        newFocus.toString());
                }
            }
    });
```

This example uses the same `ViewTreeObserver`, `vto`, and `TextView` events as in the previous example. This shows that both the currently focused `View` and the previously focused `View` pass to the listener. From here, your application can perform needed actions.

If your application merely wants to check values after the user has modified a particular `View`, though, you might need to only register to listen for focus changes of that particular `View`. This is discussed later in this chapter.

Listening for Long Clicks

In a previous section discussing the `ContextMenu` control, you learned that you can add a context menu to a `View` that is activated when the user performs a long click on that view. A long click is typically when a user presses on the touch screen and holds his finger there until an action is performed. However, a long press event can also be triggered if the user navigates there with a non-touch method, such as via a keyboard or trackball, and

then holds the Enter or Select key for a while. This action is also often called a press-and-hold action.

Although the context menu is a great typical use case for the long-click event, you can listen for the long-click event and perform any action you want. However, this is the same event that triggers the context menu. If you've already added a context menu to a `View`, you might not want to listen for the long-click event as other actions or side effects might confuse the user or even prevent the context menu from showing. As always with good user interface design, try to be consistent for usability sake.

Tip

Usually a long click is an alternative action to a standard click. If a left-click on a computer is the standard click, a long click can be compared to a right-click.

Your application can listen to the long-click event on any `View`. The following example demonstrates how to listen for a long-click event on a `Button` control:

```
Button long_press = (Button)findViewById(R.id.long_press);
long_press.setOnLongClickListener(new View.OnLongClickListener() {
    public boolean onLongClick(View v) {
        events.setText("Long click: " + v.toString());
        return true;
    }
});
```

First, the `Button` object is requested by providing its identifier. Then the `setOnLongClickListener()` method is called with our implementation of the `View.OnLongClickListener` class interface. The `View` that the user long-clicked on is passed in to the `onLongClick()` event handler. Here again we use the same `TextView` as before to display text saying that a long click occurred.

Listening for Focus Changes

We already discussed focus changes for listening for them on an entire screen. All `View` objects, though, can also trigger a call to listeners when their particular focus state changes. You do this by providing an implementation of the `View.OnFocusChangeListener` class to the `setOnFocusChangeListener()` method. The following is an example of how to listen for focus change events with an `EditText` control:

```
TextView focus = (TextView)findViewById(R.id.text_focus_change);
focus.setOnFocusChangeListener(new View.OnFocusChangeListener() {
    public void onFocusChange(View v, boolean hasFocus) {
        if (hasFocus) {
            if (mSaveText != null) {
                ((TextView)v).setText(mSaveText);
            }
        } else {
            mSaveText = ((TextView)v).getText().toString();
```

```
            ((TextView)v).setText("");
        }
    }
```

In this implementation, we also use a private member variable of type `String` for `mSaveText`. After retrieving the `EditText` view as a `TextView`, we do one of two things. If the user moves focus away from the control, we store off the text in `mSaveText` and set the text to empty. If the user changes focus to the control, though, we restore this text. This has the amusing effect of hiding the text the user entered when the control is not active. This can be useful on a form on which a user needs to make multiple, lengthy text entries but you want to provide the user with an easy way to see which one they edit. It is also useful for demonstrating a purpose for the focus listeners on a text entry. Other uses might include validating text a user enters after a user navigates away or prefilling the text entry the first time they navigate to it with something else entered.

Working with Dialogs

An `Activity` can use dialogs to organize information and react to user-driven events. For example, an activity might display a dialog informing the user of a problem or ask the user to confirm an action such as deleting a data record. Using dialogs for simple tasks helps keep the number of application activities manageable.

Tip

Many of the code examples provided in this section are taken from the SimpleDialogs application. This source code for the SimpleDialogs application is provided for download on the book website.

Exploring the Different Types of Dialogs

There are a number of different dialog types available within the Android SDK. Each has a special function that most users should be somewhat familiar with. The dialog types available include

- **`Dialog`**: The basic class for all `Dialog` types. A basic `Dialog` is shown in the top left of Figure 7.11.
- **`AlertDialog`**: A `Dialog` with one, two, or three `Button` controls. An `AlertDialog` is shown in the top center of Figure 7.11.
- **`CharacterPickerDialog`**: A `Dialog` for choosing an accented character associated with a base character. A `CharacterPickerDialog` is shown in the top right of Figure 7.11.
- **`DatePickerDialog`**: A `Dialog` with a `DatePicker` control. A `DatePickerDialog` is shown in the bottom left of Figure 7.11.
- **`ProgressDialog`**: A `Dialog` with a determinate or indeterminate `ProgressBar` control. An indeterminate `ProgressDialog` is shown in the bottom center of Figure 7.11.

- **`TimePickerDialog`**: A `Dialog` with a `TimePicker` control. A `TimePickerDialog` is shown in the bottom right of Figure 7.11.

Figure 7.11 The different dialog types available in Android.

If none of the existing `Dialog` types is adequate, you can also create custom `Dialog` windows, with your specific layout requirements.

Tracing the Lifecycle of a Dialog

Each `Dialog` must be defined within the `Activity` in which it is used. A `Dialog` may be launched once, or used repeatedly. Understanding how an `Activity` manages the `Dialog` lifecycle is important to implementing a `Dialog` correctly. Let's look at the key methods that an `Activity` must use to manage a `Dialog`:

- The `showDialog()` method is used to display a `Dialog`.
- The `dismissDialog()` method is used to stop showing a `Dialog`. The `Dialog` is kept around in the `Activity`'s `Dialog` pool. If the `Dialog` is shown again using `showDialog()`, the cached version is displayed once more.
- The `removeDialog()` method is used to remove a `Dialog` from the `Activity` objects `Dialog` pool. The `Dialog` is no longer kept around for future use. If you call `showDialog()` again, the `Dialog` must be re-created.

Adding the `Dialog` to an `Activity` involves several steps:

1. Define a unique identifier for the `Dialog` within the `Activity`.
2. Implement the `onCreateDialog()` method of the `Activity` to return a `Dialog` of the appropriate type, when supplied the unique identifier.

3. Implement the `onPrepareDialog()` method of the `Activity` to initialize the `Dialog` as appropriate.
4. Launch the `Dialog` using the `showDialog()` method with the unique identifier.

Defining a Dialog

A `Dialog` used by an `Activity` must be defined in advance. Each `Dialog` has a special identifier (an integer). When the `showDialog()` method is called, you pass in this identifier. At this point, the `onCreateDialog()` method is called and must return a `Dialog` of the appropriate type.

It is up to the developer to override the `onCreateDialog()` method of the `Activity` and return the appropriate `Dialog` for a given identifier. If an `Activity` has multiple `Dialog` windows, the `onCreateDialog()` method generally contains a switch statement to return the appropriate `Dialog` based on the incoming parameter—the `Dialog` identifier.

Initializing a Dialog

Because a `Dialog` is often kept around by the `Activity` in its `Dialog` pool, it might be important to re-initialize a `Dialog` each time it is shown, instead of just when it is created the first time. For this purpose, you can override the `onPrepareDialog()` method of the `Activity`.

Although the `onCreateDialog()` method may only be called once for initial `Dialog` creation, the `onPrepareDialog()` method is called each time the `showDialog()` method is called, giving the `Activity` a chance to modify the `Dialog` before it is shown to the user.

Launching a Dialog

You can display any `Dialog` defined within an `Activity` by calling its `showDialog()` method and passing it a valid `Dialog` identifier—one that will be recognized by the `onCreateDialog()` method.

Dismissing a Dialog

Most types of dialogs have automatic dismissal circumstances. However, if you want to force a `Dialog` to be dismissed, simply call the `dismissDialog()` method and pass in the `Dialog` identifier.

Removing a Dialog from Use

Dismissing a `Dialog` does not destroy it. If the `Dialog` is shown again, its cached contents are redisplayed. If you want to force an `Activity` to remove a `Dialog` from its pool and not use it again, you can call the `removeDialog()` method, passing in the valid `Dialog` identifier.

Working with Custom Dialogs

When the dialog types do not suit your purpose exactly, you can create a custom `Dialog`. One easy way to create a custom `Dialog` is to begin with an `AlertDialog` and use an `AlertDialog.Builder` class to override its default layout. In order to create a custom `Dialog` this way, the following steps must be performed:

1. Design a custom layout resource to display in the `AlertDialog`.
2. Define the custom `Dialog` identifier in the `Activity`.
3. Update the `Activity`'s `onCreateDialog()` method to build and return the appropriate custom `AlertDialog`. You should use a `LayoutInflater` to inflate the custom layout resource for the `Dialog`.
4. Launch the `Dialog` using the `showDialog()` method.

Working with Styles

A style is a group of common `View` attribute values. You can apply the style to individual `View` controls. Styles can include such settings as the font to draw with or the color of text. The specific attributes depend on the `View` drawn. In essence, though, each style attribute can change the look and feel of the particular object drawn.

In the previous examples of this chapter, you have seen how XML layout resource files can contain many references to attributes that control the look of `TextView` objects. You can use a style to define your application's standard `TextView` attributes once and then reference to the style either in an XML layout file or programmatically from within Java. In Chapter 6, we see how you can use one style to indicate mandatory form fields and another to indicate optional fields. Styles are typically defined within the resource file `res/values/styles.xml`. The XML file consists of a `resources` tag with any number of `style` tags, which contain an `item` tag for each attribute and its value that is applied with the style.

The following is an example with two different styles:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="padded_small">
        <item name="android:padding">2px</item>
        <item name="android:textSize">8px</item>
    </style>
    <style name="padded_large">
        <item name="android:padding">4px</item>
        <item name="android:textSize">16px</item>
    </style>
</resources>
```

When applied, this style sets the padding to two pixels and the `textSize` to eight pixels. The following is an example of how it is applied to a `TextView` from within a layout resource file:

```
<TextView
    style="@style/padded_small"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Small Padded" />
```

Styles support inheritance; therefore, styles can also reference another style as a parent. This way, they pick up the attributes of the parent style. The following is an example of how you might use this:

```
<style name="red_padded">
    <item name="android:textColor">#F00</item>
    <item name="android:padding">3px</item>
</style>

<style name="padded_normal" parent="red_padded">
    <item name="android:textSize">12px</item>
</style>

<style name="padded_italics" parent="red_padded">
    <item name="android:textSize">14px</item>
    <item name="android:textStyle">italic</item>
</style>
```

Here you find two common attributes in a single style and a reference to them from the other two styles that have different attributes. You can reference any style as a parent style; however, you can set only one style as the style attribute of a `View`. Applying the `padded_italics` style that is already defined makes the text 14 pixels in size, italic, red, and padded. The following is an example of applying this style:

```
<TextView
    style="@style/padded_italics"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Italic w/parent color" />
```

As you can see from this example, applying a style with a parent is no different than applying a regular style. In fact, a regular style can be used for applying to `Views` and used as a parent in a different style.

```
<style name="padded_xlarge">
    <item name="android:padding">10px</item>
    <item name="android:textSize">100px</item>
</style>
<style name="green_glow" parent="padded_xlarge">
```

```
    <item name="android:shadowColor">#0F0</item>
    <item name="android:shadowDx">0</item>
    <item name="android:shadowDy">0</item>
    <item name="android:shadowRadius">10</item>
</style>
```

Here the `padded_xlarge` style is set as the parent for the `green_glow` style. All six attributes are then applied to any view that this style is set to.

Working with Themes

A theme is a collection of one or more styles (as defined in the resources) but instead of applying the style to a specific control, the style is applied to all `View` objects within a specified `Activity`. Applying a theme to a set of `View` objects all at once simplifies making the user interface look consistent and can be a great way to define color schemes and other common control attribute settings.

An Android theme is essentially a style that is applied to an entire screen. You can specify the theme programmatically by calling the `Activity` method `setTheme()` with the style resource identifier. Each attribute of the style is applied to each `View` within that `Activity`, as applicable. Styles and attributes defined in the layout files explicitly override those in the theme.

For instance, consider the following style:

```
<style name="right">
    <item name="android:gravity">right</item>
</style>
```

You can apply this as a theme to the whole screen, which causes any view displayed within that `Activity` to have its gravity attribute to be right-justified. Applying this theme is as simple as making the method call to the `setTheme()` method from within the `Activity`, as shown here:

```
setTheme(R.style.right);
```

You can also apply themes to specific `Activity` instances by specifying them as an attribute within the `<activity>` element in the `AndroidManifest.xml` file, as follows:

```
<activity android:name=".myactivityname"
   android:label="@string/app_name"
   android:theme="@style/myAppIsStyling">
```

Unlike applying a style in an XML layout file, multiple themes can be applied to a screen. This gives you flexibility in defining style attributes in advance while applying different configurations of the attributes based on what might be displayed on the screen. This is demonstrated in the following code:

```
setTheme(R.style.right);
setTheme(R.style.green_glow);
setContentView(R.layout.style_samples);
```

In this example, both the `right` style and the `green_glow` style are applied as a theme to the entire screen. You can see the results of green glow and right-aligned gravity, applied to a variety of `TextView` controls on a screen, as shown in Figure 7.12. Finally, we set the layout to the `Activity`. You must do this after setting the themes. That is, you must apply all themes before calling the method `setContentView()` or the `inflate()` method so that the themes' attributes can take effect.

Figure 7.12 Packaging styles for glowing text, padding, and alignment into a theme.

A combination of well-designed and thought-out themes and styles can make the look of your application consistent and easy to maintain. Android comes with a number of built-in themes that can be a good starting point. These include such themes as `Theme_Black`, `Theme_Light`, and `Theme_NoTitleBar_Fullscreen`, as defined in the `android.R.style` class. They are all variations on the system theme, `Theme`, which built-in apps use.

Summary

The Android SDK provides many useful user interface components, which developers can use to create compelling and easy-to-use applications. This chapter introduced you to many of the most useful controls, discussed how each behaves, how to style them, and how to handle events from the user.

You learned how controls can be combined to create user entry forms. Important controls for forms include `EditText`, `Button`, `RadioButton`, `CheckBox`, and `Spinner`. You also learned about controls that can indicate progress or the passage of time to users. You mastered a variety of useful user interface constructs Android applications can take advantage of, including context and options menus, as well as various types of dialogs. In addition to drawing controls on the screen, you learned how to detect user actions, such as clicks and focus changes, and how to handle these events. Finally, you learned how to style individual controls and how to apply themes to entire screens (or more specifically, a single `Activity`) so that your application is styled consistently and thoroughly.

We talked about many common user interface controls in this chapter; however, there are many others. In Chapter 9, "Drawing and Working with Animation," and Chapter 15, "Using Android Multimedia APIs," we use graphics controls such as `ImageView` and `VideoView` to display drawable graphics and videos. In the next chapter, you learn how to use various layout and container controls to organize a variety of controls on the screen easily and accurately.