Chapter 14

Hungry Burds: A Simple Android Game

In This Chapter

- Coding an Android game
- Using Android animation
- Saving data from one run to another

What started as a simple pun involving the author's last name has turned into Chapter 14 — the most self-indulgent writing in the history of technical publishing.

The scene takes place in south Philadelphia in the early part of the 20th century. My father (then a child) sees his father (my grandfather) handling an envelope. The envelope has just arrived from the old country. My grandmother grabs the envelope out of my grandfather's hands. The look on her face is one of superiority. "I open the letters around here," she says with her eyes.

While my grandmother opens the letter, my father glances at the envelope. The last name on the envelope is written in Cyrillic characters, so my father can't read it. But he notices a short last name in the envelope's address. Whatever the characters are, they're more likely to be a short name like Burd than a longer name like Burdinsky or Burdstakovich.

The Russian word for bird is *ptitsa*, so there's no etymological connection between my last name and our avian friends. But as I grew up, I would often hear kids yell "Burd is the word" or "Hey, Burdman" from across the street. Today, my one-person Burd Brain Consulting firm takes in a small amount of change every year.

Introducing the Hungry Burds Game

When the game begins, the screen is blank. Then, for a random amount of time (averaging one second), a Burd fades into view, as shown in Figure 14-1.

If the user does nothing, the Burd disappears after fading into full view. But if the user touches the Burd before it disappears, the Burd gets a cheeseburger and remains onscreen, as shown in Figure 14-2.

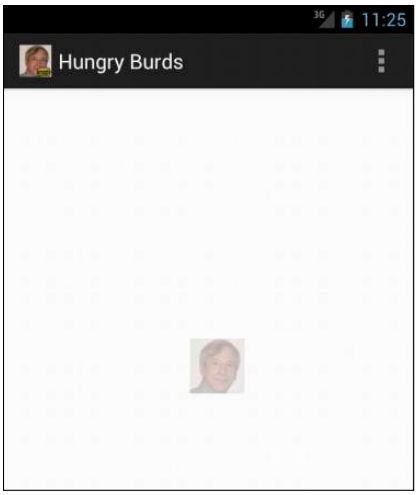

Figure 14-1: A Burd fades into view.

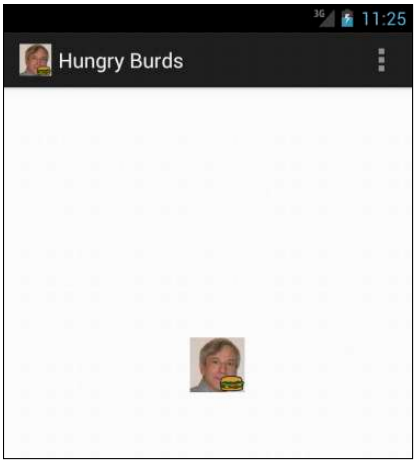

Figure 14-2: You've fed this Burd.

After ten Burds have faded in (and the unfed ones have disappeared), the screen displays a text view, showing the number of fed Burds in the current run of the game. The text view also shows the high score for all runs of the game, as shown in Figure 14-3.

For many apps, timing isn't vitally important: For them, a consistently slow response is annoying but not disabling. But for a game like Hungry Burds, timing makes a big difference. Running Hungry Burds on an emulator feels more like a waiting game than an action game. To gain a reasonable sense of how Hungry Burds works, run the app on a real-life device.

The Hungry Burds Java code is about 140 lines long. (Compare this with one of the Android game developer's books that I bought. In that book, the simplest example has 2,300 lines of Java code.) To keep the Hungry Burds code from consuming dozens of pages, I've omitted some features that you might see in a more realistically engineered game.

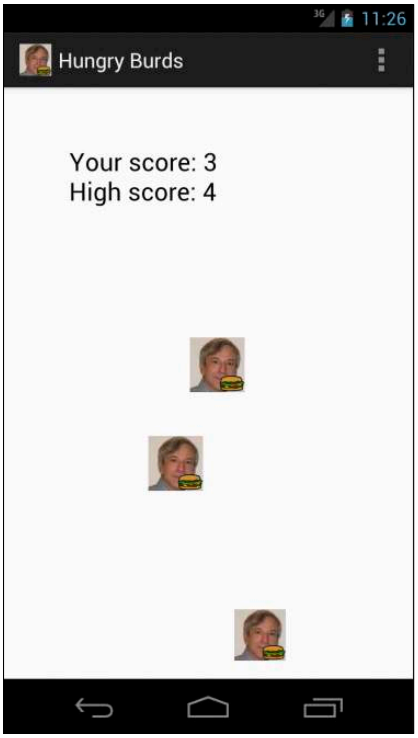

Figure 14-3: The game ends.

- **The Hungry Burds game doesn't access data over a network.**

  The game's high-score display doesn't tell you how well you did compared with your friends or with other players around the world. The high-score display applies to only one device — the one you're using to play the game.

- **The game restarts whenever you change the device's orientation.**

  If you tilt the device from Portrait mode to Landscape mode, or from Landscape mode to Portrait mode, Android calls the main activity's lifecycle methods. Android calls the activity's `onPause`, `onStop`, and `onDestroy` methods. Then it reconstitutes the activity by calling the activity's `onCreate`, `onStart`, and `onResume` methods. As a result, whatever progress you've made in the game disappears and the game starts itself over again from scratch.

  

  For an introduction to an activity's lifecycle methods, see Chapter 5.

- **The game has no Restart button.**

  To play the game a second time, you can press Android's Back button and then touch the game's launcher icon. Alternatively, you can tilt the device from Portrait mode to Landscape mode, or vice versa.

- **The screen measurements that control the game are crude.**

  Creating a visual app that involves drawing, custom images, or motion of any kind involves some math. You need math to make measurements, estimate distances, detect collisions, and complete other tasks. To do the math, you produce numbers by making Android API calls, and you use the results of your calculations in Android API library calls.

  To help me cut quickly to the chase, my Hungry Burds game does only a minimal amount of math, and it makes only the API calls I believe to be absolutely necessary. As a result, some items on the screen don't always look their best. (This happens particularly when the device is in Landscape mode.)

- **The game has no settings.**

  The number of Burds displayed, the average time of each Burd's display, and the minimal length of time for each Burd's display are all hard-coded in the game's Java file. In the code, these constants are `NUMBER_OF_BURDS`, `AVERAGE_SHOW_TIME`, and `MINIMUM_SHOW_TIME`. As a developer, you can change the values in the code and reinstall the game. But the ordinary player can't change these numbers.

- **The game isn't challenging with the default `NUMBER_OF_BURDS`, `AVERAGE_SHOW_TIME`, and `MINIMUM_SHOW_TIME` values.**

  I admit it: On this front, I'm at a distinct disadvantage. I'm a lousy game player. I remember competing in video games against my kids when they were young. I lost every time. At first it was embarrassing; in the end

it was ridiculous. I could never avoid being shot, eaten, or otherwise squashed by my young opponents' avatars.

I don't presume to know what values of `NUMBER_OF_BURDS`, `AVERAGE_SHOW_TIME`, and `MINIMUM_SHOW_TIME` are right for you. And if no values are right for you (and the game isn't fun to play no matter which values you have), don't despair. I've created Hungry Burds as a teaching tool, not as a replacement for Super Mario.

The Project's Files

The project's `AndroidManifest.xml` file is nothing special. The only element you have to watch for is `uses-sdk` — in that element, the `android:minSdkVersion` attribute has the value `13` or higher. That's because the Java code calls the `Display` class's `getSize` method, and that method isn't available in Android API levels below `13`.

If you have to get a layout's measurements in an app that runs in API Level 12 or lower, check the documentation for Android's `ViewTreeObserver.OnPreDrawListener` class.

The project's `activity_main.xml` file is almost empty, as shown in Listing 14-1. I put a `TextView` somewhere on the screen so that, at the end of each game, I can display the most recent statistics. I also add an `android:id` attribute to the `RelativeLayout` element. Using that `android:id` element, I can refer to the screen's layout in the Java code.

Listing 14-1: The Main Activity's Layout File

```
<RelativeLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/relativeLayout"
  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=".MainActivity" >

  <TextView
```

(continued)

Listing 14-1 *(continued)*

```
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_alignParentTop="true"
    android:layout_marginLeft="42dp"
    android:layout_marginTop="34dp"
    android:text="@string/nothing"
    android:textAppearance=
        "?android:attr/textAppearanceLarge" />

</RelativeLayout>
```

In the `res` directory of my Hungry Burds project, I have ten `.png` files — two files for each of Android's generalized screen densities, as shown in Figure 14-4.

For a look at Android screen densities, see Chapter 8.

Each `burd.png` file is a picture of me. Each `burd_burger.png` file is a picture of me with a cheeseburger. When Android runs the game, Android checks the device's specs and decides, on the spot, which of the five screen densities to use. (You don't need an `if` statement like the one in Chapter 8.)

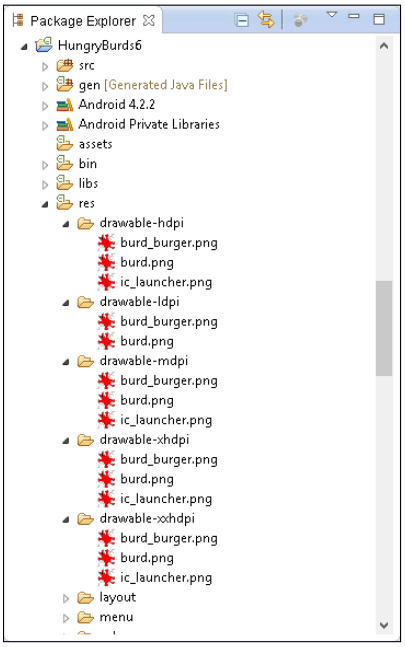

Figure 14-4: Images in the project's `res` directory.

The Main Activity

The Hungry Burds game has only one activity: the app's main activity. So you can digest the game's Java code in its entirety in one big gulp. To make this gulp palatable, I start with an outline of the activity's code. The outline is in Listing 14-2. (If outlines don't work for you, and you want to see the code in its entirety, refer to Listing 14-3.)

Listing 14-2: An Outline of the App's Java Code

```
package com.allmycode.hungryburds;

public class MainActivity extends Activity
    implements OnClickListener, AnimationListener {

  // Declare fields

  /* Activity methods */

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

    // Find layout elements

    // Get the size of the device's screen

    // Set up SharedPreferences to record high scores
  }

  @Override
  public void onResume() {
    showABurd();
  }

  /* Game methods */

  void showABurd() {
    // Add a Burd in some random place
    // At first, the Burd is invisible

    // Create an AlphaAnimation to make the Burd
    // fade in (from invisible to fully visible).
    burd.startAnimation(animation);
  }

  private void showScores() {
    // Get high score from SharedPreferences
```

(continued)

Listing 14-2 ***(continued)***

```
    // Display high score and this run's score
  }

  /* OnClickListener method */

  public void onClick(View view) {
    countClicked++;
    // Change the image to a Burd with a cheeseburger
  }

  /* AnimationListener methods */

  public void onAnimationEnd(Animation animation) {
    if (++countShown < NUMBER_OF_BURDS) {
      showABurd(); // Again!
    } else {
      showScores();
    }
  }

}
```

The heart of the Hungry Burds code is the code's game loop, as shown in the following example:

```
public void onResume() {
  showABurd();
}

void showABurd() {
  // Add a Burd in some random place.
  // At first, the Burd is invisible ...

  burd.setVisibility(View.INVISIBLE);

  // ... but the animation will make the
  // Burd visible.

  AlphaAnimation animation =
      new AlphaAnimation(0.0F, 1.0F);
  animation.setDuration(duration);
  animation.setAnimationListener(this);
  burd.startAnimation(animation);
}

public void onAnimationEnd(Animation animation) {
  if (++countShown < NUMBER_OF_BURDS) {
    showABurd(); // Again!
```

```
    } else {
      showScores();
    }
  }
```

When Android executes the `onResume` method, the code calls the `showA-Burd` method. The `showABurd` method does what its name suggests, by animating an image from alpha level 0 to alpha level 1. (Alpha level 0 is fully transparent; alpha level 1 is fully opaque.)

In the `onCreate` method, you put code that runs when the activity comes into existence. In contrast, in the `onResume` method, you put code that runs when the user begins interacting with the activity. The user isn't aware of the difference because the app starts running so quickly. But for you, the developer, the distinction between an app's coming into existence and starting to interact is important. In Listings 14-2 and 14-3, the `onCreate` method contains code to set the layout of the activity, assign variable names to screen widgets, measure the screen size, and prepare for storing high scores. The `onResume` method is different. With the `onResume` method, the user is about to touch the device's screen. So in Listings 14-2 and 14-3, the `onResume` method displays something for the user to touch: the first of several hungry Burds.

When the animation ends, the `onAnimationEnd` method checks the number of Burds that have already been displayed. If the number is less than ten, the `onAnimationEnd` method calls `showABurd` again, and the game loop continues.

By default, a Burd returns to being invisible when the animation ends. But the main activity implements `OnClickListener`, and when the user touches a Burd, the class's `onClick` method makes the Burd permanently visible, as shown in the following snippet:

```
public void onClick(View view) {
  countClicked++;
  ((ImageView) view).setImageResource
                            (R.drawable.burd_burger);
  view.setVisibility(View.VISIBLE);
}
```

The code, all the code, and nothing but the code

Following the basic outline of the game's code in the previous section, Listing 14-3 contains the entire text of the game's `MainActivity.java` file.

Listing 14-3: The App's Java Code

```
package com.allmycode.hungryburds;

import java.util.Random;

import android.app.Activity;
import android.content.SharedPreferences;
import android.graphics.Point;
import android.os.Bundle;
import android.view.Display;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.animation.AlphaAnimation;
import android.view.animation.Animation;
import android.view.animation.Animation.AnimationListener;
import android.widget.ImageView;
import android.widget.RelativeLayout;
import android.widget.RelativeLayout.LayoutParams;
import android.widget.TextView;

public class MainActivity extends Activity
    implements OnClickListener, AnimationListener {

  final int NUMBER_OF_BURDS = 10;
  final long AVERAGE_SHOW_TIME = 1000L;
  final long MINIMUM_SHOW_TIME = 500L;
  TextView textView;
  int countShown = 0, countClicked = 0;
  Random random = new Random();

  RelativeLayout relativeLayout;
  int displayWidth, displayHeight;

  SharedPreferences prefs;
  SharedPreferences.Editor editor;

  /* Activity methods */

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

    textView = (TextView) findViewById(R.id.textView1);
    relativeLayout = (RelativeLayout)
        findViewById(R.id.relativeLayout);

    Display display =
        getWindowManager().getDefaultDisplay();
    Point size = new Point();
```

```
    display.getSize(size);
    displayWidth = size.x;
    displayHeight = size.y;

    prefs = getPreferences(MODE_PRIVATE);
    editor = prefs.edit();
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
  }

  @Override
  public void onResume() {
    super.onResume();
    countClicked = countShown = 0;
    textView.setText(R.string.nothing);
    showABurd();
  }

  /* Game methods */

  void showABurd() {
    long duration =
        random.nextInt((int) AVERAGE_SHOW_TIME)
        + MINIMUM_SHOW_TIME;

    LayoutParams params = new LayoutParams
                    (LayoutParams.WRAP_CONTENT,
                     LayoutParams.WRAP_CONTENT);

    params.leftMargin =
        random.nextInt(displayWidth) * 7 / 8;
    params.topMargin =
        random.nextInt(displayHeight) * 4 / 5;

    ImageView burd = new ImageView(this);
    burd.setOnClickListener(this);
    burd.setLayoutParams(params);
    burd.setImageResource(R.drawable.burd);
    burd.setVisibility(View.INVISIBLE);

    relativeLayout.addView(burd);

    AlphaAnimation animation =
        new AlphaAnimation(0.0F, 1.0F);
    animation.setDuration(duration);
    animation.setAnimationListener(this);
    burd.startAnimation(animation);
```

(continued)

Listing 14-3 ***(continued)***

```
  }

  private void showScores() {
    int highScore = prefs.getInt("highScore", 0);

    if (countClicked > highScore) {
      highScore = countClicked;
      editor.putInt("highScore", highScore);
      editor.commit();
    }

    textView.setText("Your score: " + countClicked +
                     "\nHigh score: " + highScore);
  }

  /* OnClickListener method */

  public void onClick(View view) {
    countClicked++;
    ((ImageView) view).setImageResource
                              (R.drawable.burd_burger);
    view.setVisibility(View.VISIBLE);
  }

  /* AnimationListener methods */

  public void onAnimationEnd(Animation animation) {
    if (++countShown < NUMBER_OF_BURDS) {
      showABurd();
    } else {
      showScores();
    }
  }

  public void onAnimationRepeat(Animation arg0) {
  }

  public void onAnimationStart(Animation arg0) {
  }
}
```

Random

A typical game involves random choices. (You don't want Burds to appear in the same places every time you play the game.) Truly random values are difficult to generate. But an instance of Java's `Random` class creates what appear to be random values (*pseudorandom* values) in ways that the programmer can help determine.

For example, a `Random` object's `nextDouble` method returns a `double` value between 0.0 and 1.0 (with 0.0 being possible but 1.0 being impossible). The Hungry Burds code uses a `Random` object's `nextInt` method. A call to `nextInt(10)` returns an `int` value from `0` to `9`.

If `displayWidth` is 720 (which stands for 720 pixels), the call to `random.nextInt(displayWidth)` in Listing 14-3 returns a value from `0` to `719`. And because `AVERAGE_SHOW_TIME` is the `long` value 1000L, the expression `random.nextInt((int) AVERAGE_SHOW_TIME)` stands for a value from `0` to `999`. (The casting to `int` helps fulfill the promise that the `nextInt` method's parameter is an `int`, not a `long` value.) By adding back `MINIMUM_SHOW_TIME` (refer to Listing 14-3), I make `duration` be a number between 500 and 1499. A Burd takes between 500 and 1499 milliseconds to fade into view.

Measuring the display

Android's `Display` object stores information about a device's display. How complicated can that be? You can measure the screen size with a ruler, and you can determine a device's resolution by reading the specs in the user manual.

Of course, Android programs don't have opposable thumbs, so they can't use plastic rulers. And a layout's characteristics can change depending on several runtime factors, including the device's orientation (portrait or landscape) and the amount of screen space reserved for Android's notification bar and buttons. If you don't play your cards right, you can easily call methods that prematurely report a display's width and height as zero values.

Fortunately, the `getSize` method in Android API level 13 and higher gives you some correct answers in an activity's `onCreate` method. So, here and there in Listing 14-3, you find the following code:

```
public class MainActivity extends Activity {

  int displayWidth, displayHeight;

  public void onCreate(Bundle savedInstanceState) {

    Display display =
        getWindowManager().getDefaultDisplay();
    Point size = new Point();
    display.getSize(size);
    displayWidth = size.x;
    displayHeight = size.y;

  }

  void showABurd() {
```

```
    LayoutParams params;
    params = new LayoutParams(LayoutParams.WRAP_CONTENT,
                              LayoutParams.WRAP_CONTENT);
    params.leftMargin =
        random.nextInt(displayWidth) * 7 / 8;
    params.topMargin =
        random.nextInt(displayHeight) * 4 / 5;

  }
```

An instance of Android's `Point` class is basically an object with two components: an `x` component and a `y` component. In the Hungry Burds code, a call to `getWindowManager().getDefaultDisplay()` retrieves the device's display. The resulting display's `getSize` method takes an instance of the `Point` class and fills its `x` and `y` fields. The `x` field's value is the display's width, and the `y` field's value is the display's height, as shown in Figure 14-5.

A `LayoutParams` object stores information about the way a widget should appear as part of an activity's layout. (Each kind of layout has its own `LayoutParams` inner class, and the code in Listing 14-3 imports the `RelativeLayout.LayoutParams` inner class.) A `LayoutParams` instance has a life of its own, apart from any widget whose appearance the instance describes. In Listing 14-3, I construct a new `LayoutParams` instance before applying the instance to any particular widget. Later in the code, I call

```
burd.setLayoutParams(params);
```

to apply the new `LayoutParams` instance to one of the Burds.

Constructing a new `LayoutParams` instance with a double dose of `LayoutParams.WRAP_CONTENT` (one `LayoutParams.WRAP_CONTENT` for width and one `LayoutParams.WRAP_CONTENT` for height) indicates that a widget should shrink-wrap itself around whatever content is drawn inside it. Because the code eventually applies this `LayoutParams` instance to a Burd, the Burd will be only wide enough and only tall enough to contain a picture of yours truly from one of the project's `res/drawable` directories.

The alternative to `WRAP_CONTENT` is `MATCH_PARENT`. With two `MATCH_PARENT` parameters in the `LayoutParams` constructor, a Burd's width and height would expand to fill the activity's entire relative layout.

A `LayoutParams` instance's `leftMargin` field stores the number of pixels between the left edge of the display and the left edge of the widget. Similarly, a `LayoutParams` instance's `topMargin` field stores the number of pixels between the top edge of the display and the top edge of the widget. (Refer to Figure 14-5.)

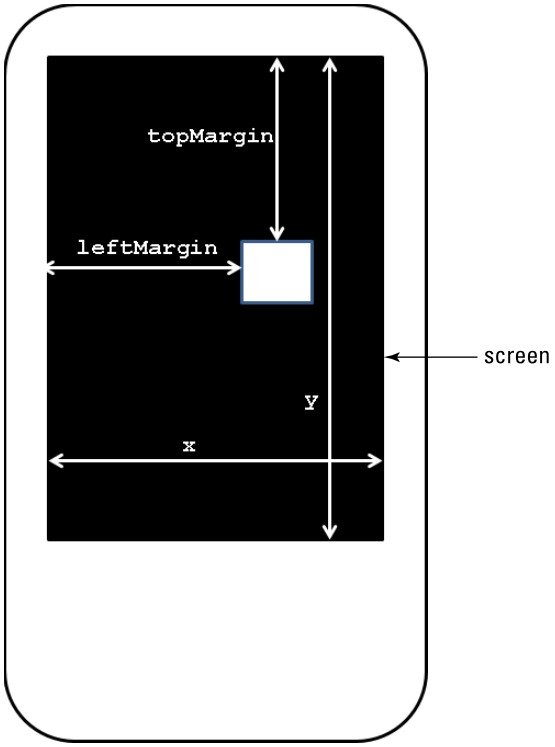

Figure 14-5: Measuring distances on the screen.

In Listing 14-3, I use random values to position a new Burd. A Burd's left edge is no farther than ⅞ths of the way across the screen, and the Burd's top edge is no lower than ⅘ths of the way down the screen. If you don't multiply the screen's width by ⅞ (or some such fraction), an entire Burd can be positioned beyond the right edge of the screen. The user sees nothing while the Burd comes and goes. The same kind of thing can happen if you don't multiply the screen's height by ⅘.

The fractions ⅞ and ⅘, which I use to determine each widget's position, are crude guesstimates of a portrait screen's requirements. A more refined app would carefully measure the available turf and calculate the optimally sized region for positioning new Burds.

Constructing a Burd

Android's `ImageView` class represents objects that contain images. Normally, you put an image file (a `.png` file, a `.jpg` file, or a `.gif` file) in one of your project's `res/drawable` directories, and a call to the `ImageView`

object's `setImageResource` method associates the `ImageView` object with the image file. In Listing 14-3, the following lines fulfill this role:

```
ImageView burd = new ImageView(this);

burd.setImageResource(R.drawable.burd);
```

Because of the `R.drawable.burd` parameter, Android looks in the project's `res/drawable` directories for files named `burd.png`, `burd.jpg`, or `burd.gif`. (Refer to Figure 14-4.) Android selects the file whose resolution best suits the device and displays that file's image on the `ImageView` object.

The statement

```
burd.setVisibility(View.INVISIBLE);
```

makes the Burd be completely transparent. The next statement

```
relativeLayout.addView(burd);
```

normally makes a widget appear on the user's screen. But with the `View.INVISIBLE` property, the Burd doesn't show up. It's not until I start the code's fade-in animation that the user begins seeing a Burd on the screen.

Android has two kinds of animation: view animation and property animation. The Hungry Burds code uses view animation. An object's `visibility` property doesn't change when a view animation makes the object fade in or fade out. In this chapter's example, a Burd starts off with `View.INVISIBLE`. A fade-in animation makes the Burd appear slowly on the screen. But when the animation finishes, the Burd's `visibility` field still contains the original `View.INVISIBLE` value. So normally, when the animation ends, the Burd simply disappears.

When the user clicks on a Burd, Android calls the `onClick` method in Listing 14-3. The `onClick` method's `view` parameter represents the `ImageView` object that the user clicked. In the body of the `onClick` method, the statement

```
((ImageView) view).setImageResource
                          (R.drawable.burd_burger);
```

assures Java that `view` is indeed an `ImageView` instance and changes the picture on the face of that instance from a hungry author to a well-fed author. The `onClick` method also sets the `ImageView` instance's visibility to `View.VISIBLE`. That way, when this Burd's animation ends, the happy Burd remains visible on the user's screen.

Android animation

Android has two types of animation:

- **View animation:** An older system in which you animate with either tweening or frame-by-frame animation, as described in this list:
  - *Tweening:* You tell Android how an object should look initially and how the object should look eventually. You also tell Android how to change from the initial appearance to the eventual appearance. (Is the change gradual or sudden? If the object moves, does it move in a straight line or in a curve of some sort? Will it bounce a bit when it reaches the end of its path?)

    With tweening, Android considers all your requirements and figures out exactly how the object looks *between* the start and the finish of the object's animation.
  - *Frame-by-frame animation:* You provide several snapshots of the object along its path. Android displays these snapshots in rapid succession, one after another, giving the appearance of movement or of another change in the object's appearance.

    Movie cartoons are the classic example of frame-by-frame animation even though, in modern moviemaking, graphics specialists use tweening to create sequences of frames.
- **Property animation:** A newer system (introduced in Android 3.0, API Level 11) in which you can modify any property of an object over a period of time.

  With property animation, you can change anything about any kind of object, whether the object appears on the device's screen or not. For example, you can increase an `earth` object's average temperature from 15° Celsius to 18° Celsius over a period of ten minutes. Rather than display the `earth` object, you can watch the way average temperature affects water levels and plant life, for example.

  Unlike view animation, the use of property animation changes the value stored in an object's field. For example, you can use property animation to change a widget from being invisible to being visible. When the property animation finishes, the widget remains visible.

The Hungry Burds code uses view animation, which includes these specialized animation classes:

- `AlphaAnimation`: Fades into view or fades out of view
- `RotateAnimation`: Turns around

- `ScaleAnimation`: Changes size
- `TranslateAnimation`: Moves from one place to another

In particular, the Hungry Burds code uses `AlphaAnimation`.

The statement

```
AlphaAnimation animation =
    new AlphaAnimation(0.0F, 1.0F);
```

creates a fade-in/fade-out animation. An alpha level of 0.0 indicates complete transparence, and an alpha level of 1.0 indicates complete opaqueness. (The `AlphaAnimation` constructor expects its parameters to be float values, so I plug the `float` values `0.0F` and `1.0F` into the constructor call.)

The call

```
animation.setAnimationListener(this);
```

tells Java that the code to respond to the animation's progress is in this main activity class. Indeed, the class header at the top of Listing 14-3 informs Java that the `HungryBurds` class implements the `AnimationListener` interface. And to make good on the implementation promise, Listing 14-3 contains bodies for the methods `onAnimationEnd`, `onAnimationRepeat`, and `onAnimationStart`. (Nothing happens in the `onAnimationRepeat` and `onAnimationStart` methods. That's okay.)

The `onAnimationEnd` method does what I describe earlier in this chapter: The method checks the number of Burds that have already been displayed. If the number is less than ten, the `onAnimationEnd` method calls `showABurd` again, and the game loop continues.

Shared preferences

When a user finishes a game of Hungry Burds, the app displays the score for the current game and the high score for all games. (Refer to Figure 14-3.) The high score display applies to only one device — the device that's running the current game. To remember the high score from one run to another, I use Android's *shared preferences* feature.

Android provides several ways to store information from one run of an app to the next. In addition to using shared preferences, you can store information in the device's SQLite database. (Every Android device has SQLite database software.) You can also store information in an ordinary Linux file or on a network host of some kind.

Here's how you wield a set of shared preferences:

- **To create shared preferences, you call the activity's `getSharedPreferences` method.**

  In fact, the `getSharedPreferences` method belongs to Android's `Context` class, and the `Activity` class is a subclass of the `Context` class.

  In Listing 14-3, I call `getSharedPreferences` in the activity's `onCreate` method. The call's parameter, `MODE_PRIVATE`, tells Android that no other app can read from or write to this app's shared preferences. (I know — there's nothing "shared" about something that no other app can use. But that's the way Android's terminology works.)

  Aside from `MODE_PRIVATE`, the alternatives are described in this list:

  - `MODE_WORLD_READABLE`: Other apps can read from these preferences.
  - `MODE_WORLD_WRITEABLE`: Other apps can write to these preferences.
  - `MODE_MULTI_PROCESS`: Other apps can write to these preferences even while an app is in the middle of a read operation. Weird things can happen with this much concurrency. If you use `MODE_MULTI_PROCESS`, watch out!

  You can combine modes with Java's bitwise *or* operator (`|`). A call such as

  ```
  getSharedPreferences(
      MODE_WORLD_READABLE | MODE_WORLD_WRITEABLE);
  ```

  makes your preferences both readable and writable for all other processes.

- **To start adding values to a set of shared preferences, you use an instance of the `SharedPreferences.Editor` class.**

  In Listing 14-3, the `onCreate` method makes a new editor object. Then, in the `showScores` method, I use the editor to add `("highScore", highScore)` to the shared preferences. Taken together, `("highScore", highScore)` is a *key/value pair*. The *value* (whatever number my `highscore` variable holds) is the actual information. The *key* (the string `"highScore"`) identifies that particular piece of information. (Every value has to have a key. Otherwise, if you've stored several different values in your app's shared preferences, you have no way to retrieve any particular value.)

  In Listing 14-3, I call `putInt` to store an `int` value in shared preferences. Android's `Editor` class (an inner class of the `SharedPreferences` class) has methods such as `putInt`, `putFloat`, `putString`, and `putStringSet`.

- **To finish adding values to a set of shared preferences, you call the editor's `commit` method.**

  In the `showScores` method in Listing 14-3, the statement `editor.commit()` does the job.

- **To read values from an existing set of shared preferences, you call `getBoolean`, `getInt`, `getFloat`, or one of the other get methods belonging to the `SharedPreferences` class.**

  In the `showScores` method in Listing 14-3, the call to `getInt` takes two parameters. The first parameter (the string `"highscore"`) is the key that identifies a particular piece of information. The second parameter (the `int` value `0`) is a default value. So when you call `prefs.getInt("highScore", 0)`, the following applies:

  - If `prefs` has no pair with key `"highscore"`, the method call returns `0`.
  - If `prefs` has a previously stored `"highscore"` value, the method returns that value.

It's Been Fun

This chapter has been fun, and this book has been fun! I love writing about Android and Java. And I love hearing from readers. Remember that you can send e-mail to me at `java4android @allmycode.com`, and you can reach me on Twitter (`@allmycode`) and on Facebook (`/allmycode`).

Occasionally, I hear from a reader who says something like this: "If I read your whole book, will I know everything I have to know about Java?" The answer is always "No, no, no!" (That's not only one "no." It's "no" times three.) No matter what topic you study, there's always more to learn. So keep reading, keep practicing, keep learning, and, by all means, keep in touch.