8 Web services with style

This chapter covers

- Calling a web service
- Parsing an XML document
- Dynamically editing the scene graph
- Animating, with off-the-shelf transitions

In this chapter we're going to cover a range of exciting JavaFX features—and possibly the most fun project thus far. In previous chapters we were still learning the ropes, so to speak, but now that bind and triggers are becoming second nature and the scene graph is no longer a strange alien beast, we can move on to some of the power tools the JavaFX API has to offer.

We'll start by learning how to call a web service and parse its XML response. As more of our data is moving online and hidden behind web services, knowing how to exploit their power from within our own software becomes crucial. Then we'll turn our attention to taking the pain out of animation effects. As if the animation tools built into JavaFX Script weren't enough, JavaFX also includes a whole library of off-the-shelf *transition* classes. When we apply these classes to scene graph nodes, we can make them move, spin, scale, and fade with ease.

This chapter has a lot to cover, but by the time you reach its end you'll have experienced practical examples of most of the core JavaFX libraries. There are still plenty of juicy morsels in the remaining chapters. But after this we'll be focusing more on techniques and applications than on learning new API classes.

Let's get started.

8.1 Our project: a Flickr image viewer

Everyone and his dog are writing demos to exploit the Flickr web service API. It's not hard to understand why. Located at http://www.flickr.com, the site is an online photo gallery where the public can upload and arrange its digital masterpieces for others to browse. It's highly addictive trawling though random photos, some looking clearly amateur but others shockingly professional. Of course, as programmers we just want to know how cool the programming API is! As it happens, Flickr's API is pretty cool (which explains why everyone and his dog are using it).

Why did I decide to go with Flickr for this book? First, it's a well-known API with plenty of documentation and programmers who are familiar with it—important for anyone playing with the source code after the chapter has been read. Second, I wanted to show that photo gallery applications don't have to be boring (witness figure 8.1), particularly when you have a tool like JavaFX at your disposal.

The application we're going to build will run full screen. It will use a web service API to fetch details of a particular gallery then show thumbnails in a strip along the bottom of the screen, one page at a time. Selecting a thumbnail will cause the image

Figure 8.1 Our photo viewer will allow us to contact the online photo service, view thumbnails from a gallery, and then select and toss a full-sized image onto a desktop as if it were a real photo.

Thanks, Sally!

I'd like to thank Sally Lupton, who kindly allowed her gallery to be used to illustrate figures 8.1, 8.3, and 8.4 in this chapter. Her *Superlambanana* photos were certainly a lot nicer than anything your humble author could produce.

to spin onto the main desktop (the background) display, looking as if it's a printed photograph. The desktop can be dragged to move the photos, and as more pictures are dropped onto it, older ones (at the bottom of the heap) gracefully fade away.

8.1.1 The Flickr web service

A *web service* is a means of communicating between two pieces of software, typically on different networked computers. The client request is formulated using HTTP in a way that mimics a remote method invocation (RMI); the server responds with a structured document of data in either XML or JSON.

Flickr has quite a rich web service, with numerous functions covering a range of the site's capabilities. It also supports different web service data formats. In our project we'll use a lightweight ("RESTful") protocol to send the request, with the resulting data returned to us as an XML document. REST (Representational State Transfer) is becoming increasingly popular as a means of addressing web services; it generally involves less work than the heavily structured alternatives based on SOAP.

Not enough REST?

For more background information on REST take a look at its Wikipedia page. The official JavaFX site also hosts a Media Browser project that demonstrates a RESTful service.

http://en.wikipedia.org/wiki/Representational_State_Transfer
http://javafx.com/docs/tutorials/mediabrowser/

Before we can go any further, you'll need to register yourself as a Flickr developer, assuming you don't have an account already.

8.1.2 Getting registered with Flickr

You must register with Flickr so you can call its web service, which is a necessary part of this project. Signing up is relatively quick to do and totally free for nonprofessional use. Once your account is created you'll be assigned a key (a long hexadecimal string) for use in any software you write accessing the service. The necessity for a key, it seems, is primarily to stop a single developer from flooding the site with requests.

Go to http://www.flickr.com/services/api/ and click the Sign Up link at the head of the page to begin the process of creating your account. The site will walk you through what you need to do, which shouldn't take long. Once your account is created, a Your

API Keys link will appear at the head of the page whenever you're logged in. Click it to view your developer account details, including the all-important key.

The site contains plenty of API documentation and tutorials. We'll be using only a tiny subset of the full API, but once you've seen an example of one web service call, it should be clear how to apply the documentation to call another.

So, if you don't already have a Flickr developer account, put this book down and register one right now, before you read any further. You can't run the project code without one, and the very next section will throw us straight into web service coding.

8.2 Using a web service in JavaFX

At the heart of JavaFX's web service support are three classes. In the `javafx.io.http` package there's the `HttpRequest` class, used to make the HTTP request; in `javafx.data.pull` there's `PullParser` and `Event`, used to parse the reply.

Our application also uses three classes itself: `FlickrService` handles the request (using `HttpRequest`), `FlickrResult` processes the result (using `PullParser` and `Event`), and `FlickrPhoto` stores the details of the photos as they are pulled from the result.

In the sections ahead we'll examine each of these classes.

8.2.1 Calling the web service with HttpRequest

We'll start, naturally enough, with the `FlickrService`. You'll find it in listing 8.1. As in previous chapters, the listing has been broken into stages to aid explanation.

Listing 8.1 FlickrService.fx (part 1)

```
package jfxia.chapter8;

import javafx.io.http.HttpRequest;
import javafx.data.pull.PullParser;
import java.io.InputStream;
import java.lang.Exception;

def REST:String = "http://api.flickr.com/services/rest/";

function createArgList(args:String[]) : String {
    var ret="";
    var sep="";
    for(i in [0..<sizeof args step 2]) {
        ret="{ret}{sep}{args[i]}={args[i+1]}";
        sep="&";
    }
    return ret;
}
// ** Part 2 is listing 8.2; part 3 is listing 8.3
```

URL of web service

Create HTTP query string from keys/values

We begin with one variable and one function, at the script level. The variable, `REST`, is the base URL for the web service we'll be addressing. Onto this we'll add our request and its parameters. The function `createArgList()` is a useful utility for building the argument string appended to the end of `REST`. It takes a sequence of

key/value pairs and combines each into a query string using the format `key=value`, separated by ampersands.

Listing 8.2 shows the top of the `FlickrService` class itself.

Listing 8.2 FlickrService.fx (part 2)

```
// ** Part 1 is listing 8.1
public class FlickrService {
    public var apiKey:String;
    public var userId:String;
    public var photosPerPage:Integer = 10;
    public-read var page:Integer = 0;
    public var onSuccess:function(:FlickrResult);
    public var onFailure:function(:String);

    var valid:Boolean;

    init {
        valid = isInitialized(apiKey);
        if(not valid)
            println("API key required.");
    }
// ** Part 3 is listing 8.3
```

Callback functions

Missing API key?

Check for API key

At the head of the class we see several variables:

- `apiKey` holds the developer key (the one associated with your Flickr account).
- `userId` is for the account identifier of the person whose gallery we'll be viewing.
- `photosPerPage` and `page` determine the page size (how many thumbs are fetched at once) and which page was previously fetched.
- `onSuccess` and `onFailure` are function types, permitting us to run code on the success or failure of our web service request.

In the `init` block we test for `apiKey` initialization; if it's unset we print an error message. A professional application would do something more useful with the error, of course, but for our project a simple error report like this will suffice (it keeps the class free of too much off-topic detail).

We conclude the code with listing 8.3.

Listing 8.3 FlickrService.fx (part 3)

```
// ** Part 1 is listing 8.1; part 2 is listing 8.2
    public function loadPage(p:Integer) : Void {
        if(not valid) throw new Exception("API key not set.");

        page = p;

        var args = [
            "method",     "flickr.people.getPublicPhotos",
            "api_key",     apiKey,
            "user_id",     userId,
            "per_page",    photosPerPage.toString(),
            "page",        page.toString()
        ];

        def http:HttpRequest = HttpRequest {
```

Request arguments

Web call

```
            method: HttpRequest.GET;                                  Method and
            location: "{REST}?{createArgList(args)}";                 address
            onResponseCode: function(code:Integer) {                      <-- Initial
                if(code!=200 and onFailure!=null)                             response
                    onFailure("HTTP code {code}");
            }
            onException: function(ex:Exception) {      <-- I/O error
                if(onFailure!=null)
                    onFailure(ex.toString());
            }
            onInput: function(ip:InputStream) {        <-- Success!
                def fr = FlickrResult {};
                def parser = PullParser  {                            Create and call
                    documentType: PullParser.XML;                     XML parser
                    input: ip;
                    onEvent: fr.xmlEvent;
                };
                parser.parse();
                parser.input.close();
                if(onSuccess!=null)  onSuccess(fr);
            }
        };
        http.start();
    }
}
```

In the final part of our service request code `loadPage()` function is where the action is; it takes a page number and accesses the Flickr service to fetch the photo details for that page. Each request ends in a call to either `onSuccess` or `onFailure` (if populated), allowing applications to run their own code when the process ends. (We'll deal with how our photo viewer uses these functions later.)

After (double) checking the `apiKey` and storing the selected page, `loadPage()` creates a sequence of key/value pairs to act as the arguments passed to the service call. The first list argument is the function we're calling on the web service, and the following arguments are parameters we're passing in.

Flickr's `flickr.people.getPublicPhotos` function returns a list of photos for a given user account, page by page. We need to pass in our own key, the ID of the person whose gallery we want to read, the number of photos we want back (the page size to break the gallery up into), and which page we want. See the web service API documentation for more details on this function.

After the argument list we have the `HttpRequest` object itself. The HTTP request doesn't execute immediately. Web service requests are commonly instigated from inside UI event handlers; if we performed the request immediately, it would hog the current thread (the GUI thread) and cause our application's interface to become temporarily unresponsive. Instead, when `start()` is called, the network activity is pushed onto another thread, and we assign callbacks to run when there's something ready to act upon (see figure 8.2).

The `HttpRequest` request declaratively sets a number of properties. The `method` and `location` variables tell `HttpRequest` how and where to direct the HTTP call. To form the web address we use the script function `createArgList()`, turning the `args`

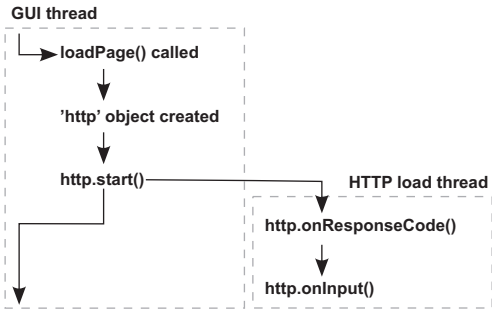

Figure 8.2 When `start()` is called on an `HttpRequest` object, a second thread takes over and communicates its progress through callback events, allowing the GUI thread to get back to its work.

sequence into a web-like query string, and append it to the `REST` base URL. The `onResponseCode`, `onException`, and `onInput` event function types will be called at different stages of the request life cycle. The `HttpRequest` class actually has a host of different functions and variables to track the request state in fine detail (check the API docs), but typically we don't need such fine-grained control.

The `onResponseCode` event is called when the initial HTTP response code is received (`200` means "ok"; other codes signify different results), `onException` is called if there's an I/O problem, while `onInput` is called when the result actually starts to arrive. The `onInput` call passes in a Java `InputStream` object, which we can assign a parser to. The JavaFX class `PullParser` is just such a parser. It reads either XML- or JSON-formatted data from the input stream and breaks it down into a series of events. To receive the events we need to register a function. But because our particular project needs to store some of the data being returned, I've written not just a single function but an entire class (the `FlickrResult` class) to interact with it. And that's what we'll look at next.

8.2.2 Parsing XML with PullParser

Because we need somewhere to store the data we are pulling from the web service, we'll create an entire class to interact with the parser. That class is `FlickrResult`, taking each XML element as it is encountered, extracting data, and populating its variables. The class also houses a `FlickrPhoto` sequence, to store details for each individual photo.

Listing 8.4 is the first part of our class to process and store the information coming back from the web service.

Listing 8.4 FlickrResult.fx (part 1)

```
package jfxia.chapter8;

import javafx.data.pull.Event;
import javafx.data.pull.PullParser;
import javafx.data.xml.QName;

public class FlickrResult {
    public-read var stat:String;
```

Status message from service

```
    public-read var total:Integer;
    public-read var perPage:Integer;
    public-read var page:Integer;
    public-read var pages:Integer;

    public-read var photos:FlickrPhoto[];

    public def valid:Boolean =  bind (stat == "ok");
// ** Part 2 is listing 8.5
```

Gallery details

Data for each photo in pages

Was request successful?

Let's have a closer look at the details:

- The `stat` variable holds the success/failure of the response, as described in the reply. If Flickr can fulfill our request, we'll get back the simple message "ok".
- The `total` variable holds the number of photos in the entire gallery, `perPage` contains how many there are per page (should match the number requested), and `pages` details the number of available pages (based on the total and number of photos per page).
- `page` is the current page (again, it should match the one we requested).
- The `valid` variable is a handy boolean for checking whether Flickr was able to respond to our request.

Listing 8.5 is the second half of our parser class. It contains the code that responds to the `PullParser` events. So we're not working blindly, the following is an example of the sort of XML the web service might reply with. Each opening element tag, closing element tag, and loose text content inside an element cause our event handler to be called.

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
  <photos page="1" pages="20" perpage="10" total="195">
    <photo id="3188821292" owner="12345678@N09" secret="cafebabe"
        server="3095" farm="4" title="Hello"
        ispublic="1" isfriend="0" isfamily="0" />
    <!-- Another nine photo elements appear here -->
  </photos>
</rsp>
```

And now, here is the code to parse this data.

Listing 8.5 FlickrResult.fx (part 2)

```
// ** Part 1 is listing 8.4
    public function xmlEvent(ev:Event) : Void {
        if(not (ev.type == PullParser.START_ELEMENT)) {
            return;
        }

        if(ev.level==0 and ev.qname.name == "rsp") {
            stat = readAttrS(ev,"stat");
        }
        else if(ev.level==1 and ev.qname.name == "photos") {
            total = readAttrI(ev,"total");
```

Not a start element? Exit!

Top level, <rsp>

2nd level, <photos>

```
            perPage = readAttrI(ev,"perpage");
            page = readAttrI(ev,"page");
            pages = readAttrI(ev,"pages");
        }
        else if(ev.level==2 and ev.qname.name == "photo") {
            def photo = FlickrPhoto {
                id: readAttrS(ev,"id");
                farm: readAttrS(ev,"farm");
                owner: readAttrS(ev,"owner");
                secret: readAttrS(ev,"secret");
                server: readAttrS(ev,"server");
                title: readAttrS(ev,"title");
                isFamily: readAttrB(ev,"isfamily");
                isFriend: readAttrB(ev,"isfriend");
                isPublic: readAttrB(ev,"ispublic");
            };
            insert photo into photos;
        }
        else {
            println("{ev}");
        }
    }

    function readAttrS(ev:Event,attr:String) : String {
        def qn = QName{name:attr};
        return ev.getAttributeValue(qn) as String;
    }
    function readAttrI(ev:Event,attr:String) : Integer {
        return java.lang.Integer.parseInt(readAttrS(ev,attr));
    }
    function readAttrB(ev:Event,attr:String) : Boolean {
        return (readAttrI(ev,attr)!=0);
    }
}
```

3rd level, <photo>

Create and store photo object

Didn't recognize element

Read string attribute

Read integer attribute

Read boolean attribute

The function `xmlEvent()` is the callback invoked whenever a node in the XML document is encountered (note: *node* in this context does *not* refer to a scene graph node). Both XML and JSON documents are nested structures, forming a tree of nodes. JavaFX's parser walks this tree, firing an event for each node it encounters, with an `Event` object to describe the type of node (text or tag, for example), its name, its level in the tree, and so on.

Our XML handler is interested only in starting tags; that's why we exit if the node type isn't an element start. The large `if/else` block parses specific elements. At level 0 we're interested in the `<rsp>` element, to get the status message (which we hope will be "ok"). At level 1 we're interested in the `<photos>` element, with attributes describing the gallery, page size, and so on. At level 2, we're interested in the `<photo>` element, holding details of a specific photo on the page we're reading. For any other type of element, we simply print to the console (handy for debugging) and then ignore.

The `<photo>` element is where we create each new `FlickrPhoto` object, with the help of three private functions for extracting named attributes from the tag in given data formats. Let's look at the `FlickrPhoto` class, in listing 8.6.

Listing 8.6 FlickrPhoto.fx

```
package jfxia.chapter8;

public def SQUARE:Number = 75;
public def THUMB:Number = 100;
public def SMALL:Number = 240;
public def MEDIUM:Number = 500;
public def LARGE:Number = 1024;

public class FlickrPhoto {
    public-init var id:String;
    public-init var farm:String;
    public-init var owner:String;
    public-init var secret:String;
    public-init var server:String;
    public-init var title:String;
    public-init var isFamily:Boolean;
    public-init var isFriend:Boolean;
    public-init var isPublic:Boolean;

    def urlBase:String = bind
        "http://farm{farm}.static.flickr.com/"
        "{server}/{id}_{secret}";
    public def urlSquare:String = bind "{urlBase}_s.jpg";
    public def urlThumb:String = bind "{urlBase}_t.jpg";
    public def urlSmall:String = bind "{urlBase}_m.jpg";
    public def urlMedium:String = bind "{urlBase}.jpg";
    //public def urlLarge:String = bind "{urlBase}_b.jpg";
    //public def urlOriginal:String = bind "{urlBase}_o.jpg";
}
```

Image pixel sizes

Photo data, provided by the XML

Base image address

Actual image URLs

Each Flickr photo comes prescaled to various sizes, accessible via slightly different filenames. You'll note that script-level constants are used to describe the sizes of these images.

- A square thumbnail is 75 x 75 pixels.
- A regular thumbnail is 100 pixels on its longest side.
- A small image is 240 pixels on its longest side.
- A medium image is 500 pixels on its longest side.
- A large image is 1024 pixels on its longest side.
- The original image has no size restrictions.

Inside the class proper we find a host of `public-init` properties that store the details supplied via the XML response. The `farm`, `secret`, and `server` variables are all used to construct the web address of a given image. The other variables should be self-explanatory.

At the foot of the class we have the web addresses of each scaled image. The different sizes of image all share the same basic address, with a minor addition to the filename for each size (except for medium). We can use these addresses to load our thumbnails and full-size images. In our project we'll be working with the thumbnail and medium-size images only. The class can load any of the images, but since extra

steps and permissions may be required to load the larger-size images using the web service API, I've commented out the last two web addresses. The web service documentation explains how to get access to them.

That's all we require to make, and consume, a web service request. Now all that's needed is code to test it, but before we go there, let's recap the process, to ensure you understand what's happening.

8.2.3 A recap

The process of calling a web service may seem a bit convoluted. A lot of classes, function calls, and event callbacks are involved, so here's a blow-by-blow recap of how our project code works:

1. We formulate a web service request in `FlickrService`, using the service function we want to call plus its parameters.
2. Our `FlickrService` has two function types (event callbacks), `onSuccess` and `onFailure`, called upon the outcome of a web service request. We implement functions for these to deal with the data once it has loaded or handle any errors.
3. Now that everything is in place, we use JavaFX's `HttpRequest` to execute the request itself. It begins running *in the background*, allowing the current GUI thread to continue running unblocked.
4. If the `HttpRequest` fails, `onFailure` will be called. If we get as far as an `InputStream`, we create a parser (JavaFX's `PullParser`) to deal with the XML returned by the web service. The parser invokes an event callback function as incoming nodes are received, which we assign to a function, `xmlEvent()` in `FlickrResult`, a class designed to store data received from the web service.
5. The callback function in `FlickrResult` parses each start tag in the XML. For each photo it creates and stores a new `FlickrPhoto` object.
6. Once the parsing is finished, execution returns to `onInput()` in `FlickrService`, which calls `onSuccess` with the resulting data.
7. In the `onSuccess` function we can now do whatever we want with the data loaded from the service.

Now, at last, we need to actually see our web service in action.

8.2.4 Testing our web service code

Having spent the last few pages creating a set of classes to extract data from our chosen web service, we'll round off this part of the project with listing 8.7, showing how easy it is to use.

Listing 8.7 TestWS.fx

```
package jfxia.chapter8;

FlickrService {
    apiKey: "-";    // <== Your key goes here
    userId: "29803026@N08";
```

User's gallery to view

```
    photosPerPage: 10;

    onSuccess: function(res:FlickrResult) {
        for(photo in res.photos) {
            println("{photo.urlMedium}");
        }
    }
    onFailure: function(s:String) {
        println("{s}");
    }
}.loadPage(1);

javafx.stage.Stage { visible: true; }
```

Success, print photo URLs

Failure, print message

Prevent termination

Listing 8.7 is a small test program to create a web service request for photo details and print the URL of each medium-size image. To make the code work you'll need to supply the key you got when you signed up for a Flickr developer account.

As you can see, all that hard work paid off in the form of a nice and simple class we can use to get at our photo data.

Because the network activity takes place in the background, away from the main GUI thread, we need to stop the application from immediately terminating before Flickr has time to send back any details. We do this by creating a dummy window; it's a crude solution but effective. If all goes well, the code should spit out onto the console a list of 10 web addresses, one for each medium-size image in the first page of the gallery we accessed.

Now that our network code is complete, we can get back to our usual habit of writing cool GUI code.

8.3 Picture this: the PhotoViewer application

In this, the second part of the project, we're going to use the web service classes we developed earlier in an application to throw photos on screen. The application will be full screen—that is to say, it will not run in a window on the desktop but will take over the whole display. It will also use transitions to perform its movement and other animated effects.

The application has a bar of thumbnails along its foot, combined with three buttons, demonstrated in figure 8.3. One button moves the thumbnails forward by a page, another moves them back, and the final button clears the main display area (or *desktop*, as I'm calling it) of photos. As we move over the thumbnails in the bar, the associated title text, which the web service gave us, is displayed.

To get a photo onto the desktop, we merely click its thumbnail to see it spin dramatically onto the display scaled to full size. Initially we scale the tiny thumbnail up to the size of the photo, while we wait for the higher-resolution image to be loaded. As we wait, a progress bar appears in the corner of the photo, showing how much of the image has arrived over the network. When the high-resolution image finally finishes loading, it replaces the scaled-up thumbnail, and the progress bar vanishes.

We can click and drag individual images to move them around the desktop, or we can click on an empty part of the desktop and drag to move all the images at once.

Figure 8.3 Photos selected from the thumbnail bar fly onto the desktop.

The application itself is constructed from two further classes, weighing in at over 200 lines apiece. But don't worry, they still contain plenty of fresh JavaFX goodness for us to explore. As usual, they've been broken up into parts to aid explanation. We begin with the class that handles the thumbnail bar.

8.3.1 Displaying thumbnails from the web service: the GalleryView class

The `GalleryView` class is the visual component that deals with the Flickr web service, and it presents a horizontal list of thumbnails based on the data it extracts from the service. Figure 8.4 shows the specific part of the application we're building.

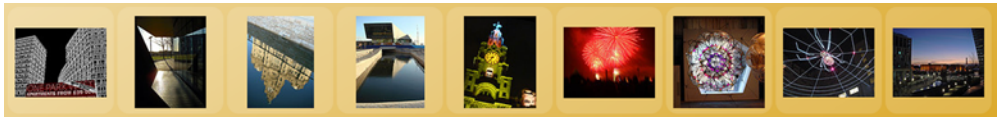

Figure 8.4 The custom scene graph node we are creating

That's what we want it to look like; let's dive straight into the source code with listing 8.8. GalleryView.fx is presented in four parts: listings 8.8, 8.9, 8.10, and 8.11. Here we see the variables in the top part of our `GalleryView` class.

Listing 8.8 GalleryView.fx (part 1)

```
package jfxia.chapter8;

import javafx.animation.Interpolator;
```

```
import javafx.animation.transition.TranslateTransition;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;

package class GalleryView extends CustomNode {
    def thumbWidth:Number = FlickrPhoto.THUMB;
    def thumbHeight:Number = FlickrPhoto.THUMB;
    def thumbBorder:Number =  10;

    public-init var apiKey:String;
    public-init var userId:String;
    package var width:Number = 0;
    package def height:Number = thumbHeight + thumbBorder*2;
    package var action:function(:FlickrPhoto,
        :Image,:Number,:Number);

    public-read var page:Integer = 1
        on replace {
            loadPage();
        };
    public-read var pageSize:Integer =
        bind {
            def aw:Number = width;
            def pw:Number = thumbWidth + thumbBorder*2;
            (aw/pw).intValue();
        }
        on replace {
            if(pageSize>0)  createUI();
            loadPage();

        };
    var service:FlickrService;
    var result:FlickrResult;
    var thumbImages:Image[];

    var topGroup:Group = Group{};
    var thumbGroup:Group;
    var textDisplay:Text;
// ** Part 2 is listing 8.9; part 3, listing 8.10; part 4 is listing 8.11
```

Handy constants

Flickr details

Thumbnail clicked event

New page means thumbs reload

Width determines thumbnail count

Rebuild scene graph on change

Flickr classes and fetched thumbs

Handy scene graph stuff

At its head we see a few handy constants being defined: the maximum dimensions of a thumbnail (brought in from the `FlickrPhoto` class for convenience) and the gap between each thumbnail. The other variables are:

- `apiKey` and `userId` should be familiar from the first part of the project. We set these when we create a `GalleryView`, so it can access the Flickr service.
- The `height` of the bar we can calculate from the size of the thumbnails and their surrounding frames, but the `width` is set externally by using the size of the screen.

- The `action` function type holds the callback we use to tell the outside application that a thumbnail has been clicked. The parameters are the `FlickrPhoto` associated with this thumbnail, the thumb `Image` already downloaded, and the x/y location of the thumbnail in the bar.
- The `page` and `pageSize` variables control which page of thumbnails is loaded and how many thumbnails are on that page. Changing either causes an access to the web service to fetch fresh data, which is why both have an `on replace` block. A change to `pageSize` will also cause the contents of the scene graph to be rebuilt, using the function we created for this very purpose. The page size is determined by the number of thumbnails we can fit inside the `width`, which explains the bind.
- The private variables `service`, `results`, and `thumbImages` all relate directly to the web service. The first is the interface we use to load each page of thumbnails, the second is the result of the last page load, and finally we have the actual thumbnail images themselves.
- Private variables `topGroup`, `thumbGroup`, and `textDisplay` are all parts of the scene graph that need manipulating in response to events.

Now we'll turn to the actual code that initializes those variables. Listing 8.9 sets up the web service and returns the top-level node of our bit of the scene graph.

Listing 8.9 GalleryView.fx (part 2)

```
// ** Part 1 is listing 8.8
    init {
        service = FlickrService {
            apiKey: bind apiKey;
            userId: bind userId;
            photosPerPage: bind pageSize;
            onSuccess: function(res:FlickrResult)
            {   result = res;
                assignThumbs(result);
            }
            onFailure: function(s:String)
            {   println("{s}");
            }
        };
    }

    override function create() : Node {
         topGroup;
    }
// ** Part 3 is listing 8.10; part 4, listing 8.11
```

Flickr web service class

Do this on success

Do this on failure

Return scene graph node

The code shouldn't need too much explanation. We register two functions with the `FlickrService` class: `onSuccess` will run when data has been successfully fetched, and `onFailure` will run if it hits a snag. In the case of a successful load, we store the `result` object so we can use its data later, and we call a private function (see later for the code) to copy the URLs of the new thumbnails out of the `result` and into the `thumbImage` sequence, causing them to start loading.

Listing 8.10, the third part of the code, takes us to the scene graph for this class.

Listing 8.10 GalleryView.fx (part 3)

```
// ** Part 1 is listing 8.8; part 2, listing 8.9
    function createUI() : Void {                          // Create actual scene graph
        def sz:Number = thumbWidth + thumbBorder*2;
        var thumbImageViews:ImageView[] = [];

        textDisplay = Text {                              // Photo title banner text
            font: Font { size: 30; }
            fill: Color.BROWN;
            layoutX: bind
                (width-textDisplay.layoutBounds.width)/2;
        }

        thumbGroup = Group {                              // Sequence of thumbnail images
            content: for(i in [0..<pageSize]) {
                var iv:ImageView = ImageView {
                    layoutX: bind i*sz + thumbBorder +    // Center align thumb
                        (if(iv.image==null) 0
                        else (thumbWidth-iv.image.width)/2);
                    layoutY: bind thumbBorder +
                        (if(iv.image==null) 0
                        else (thumbHeight-iv.image.height)/2);
                    fitWidth: thumbWidth;                 // Resize image
                    fitHeight: thumbHeight;
                    preserveRatio: true;

                    image: bind if(thumbImages!=null)     // Use image, if available
                        thumbImages[i] else null;
                };
                insert iv into thumbImageViews;
                iv;
            };
        };

        def frameGroup:Group = Group {                    // Background rectangle
            content: for(i in [0..<pageSize]) {
                def r:Rectangle = Rectangle {
                    layoutX: i*sz + 2;
                    width: sz-4;  height: sz - 4;
                    arcWidth: 25;  arcHeight: 25;
                    opacity: 0.15;
                    fill: Color.WHITE;

                    onMouseEntered: function(ev:MouseEvent) {   // Change opacity, show title text
                        r.opacity = 0.35;
                        if(result!=null and
                            i<(sizeof result.photos)) {
                            textDisplay.content =
                                result.photos[i].title;
                        }
                    }
                    onMouseExited: function(ev:MouseEvent) {    // Change opacity, clear title text
                        r.opacity = 0.15;
                        textDisplay.content = "";
                    }
```

```
                    onMouseClicked: function(ev:MouseEvent) {
                        if(action!=null) {
                            def f = result.photos[i];
                            def t = thumbImages[i];
                            def v = thumbImageViews[i];
                            def x:Number = v.layoutX;
                            def y:Number = v.layoutY;
                            action(f, t,x,y);
                        }
                    }
                };
            };
        };

        topGroup.layoutX = (width - pageSize*sz) / 2;
        topGroup.content =  [
             frameGroup , thumbGroup , textDisplay
        ];
    }
// ** Part 4 is listing 8.11
```

Fire action event

Center gallery view

Create top-level node

Listing 8.10 is the real meat of the scene graph. Whenever the `pageSize` variable is changed (assuming it's not the initial zero assignment), the `createUI()` function runs to repopulate the `topGroup` node, which is the root of our scene graph fragment.

The code constructs three layers of node, as shown in figure 8.5. We group the images separately from their background frames, so they can be animated independently. Whenever a new page of thumbnails is loaded, the current thumbnails will fall out of view; we can animate them separately only if they are grouped apart from their framing rectangles.

The `textDisplay` is used to show the title of each image as the mouse passes over it. We build the `thumbGroup` by adding an image at a time, scaled to fit the space available (which shouldn't be necessary if Flickr deliver the thumbs in the size we expect, but just in case this changes). We store each `Image` as we create it, because we'll need access to this thumbnail in the `onMouseClicked()` event handler.

The `frameGroup` is where we put all the mouse handling code. A rollover causes the photo's text to be displayed (entered) or cleared (exited), and the frame's opacity is changed. When the mouse is clicked it triggers an `action` event, with the appropriate `FlickrPhoto` object, its thumbnail image (we stored it earlier), and the coordinates within the bar, all bundled up and passed to the function assigned to `action`.

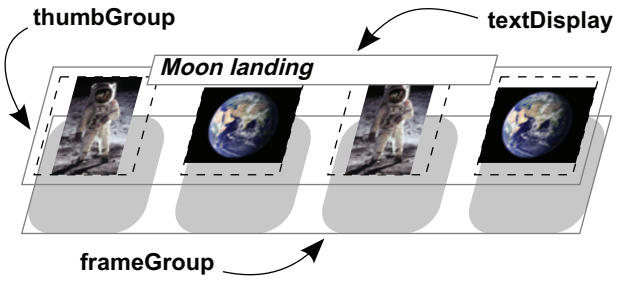

Figure 8.5 The thumbnail bar scene graph is made up on three core components: a group of frame rectangles in the background, a group of images over it, and a text node to display the thumb titles.

Almost there! Only one block of code left, listing 8.11, and it is has a surprise up its sleeve.

Listing 8.11 GalleryView.fx (part 4)

```
// ** Part 1 is listing 8.8; part 2, listing 8.9; part 3, listing 8.10
    package function next() : Void { setPage(page+1); }
    package function previous() : Void { setPage(page-1); }
    function setPage(p:Integer) : Void {                    <- Make sure page is within range
        if(result!=null) {
            page =
                if(p<=0)  result.pages
                else if(p>result.pages)  1                  <- Wrap around on over/underflow
                else  p;
        }
    }
    function loadPage() : Void {                            <- Load page of thumbs
        if(service!=null and pageSize>0 and page>0) {
            TranslateTransition {                           <- Move...
                node: thumbGroup;                           <- ...this node...
                byX: 0;  byY: height;                       <- ...by this amount.
                interpolator: Interpolator.EASEIN;
                duration: 0.5s;
                action: function() {                        <- Do this, when finished
                    unassignThumbs();
                    thumbGroup.translateY = 0;
                    service.loadPage(page);
                }
            }.play();                                       <- Run transition
        }
    }
    function assignThumbs(res:FlickrResult) : Void {        <- URLs from result into ImageViews
        thumbImages = for(i in [0..<sizeof res.photos]) {
            var im = Image {
                url: res.photos[i].urlThumb;
                backgroundLoading: true;
            };
        };
    }
    function unassignThumbs() : Void {                      <- Clear thumb image
        thumbImages =  [];
    }
}
```

The final part of our mammoth `GalleryView` class begins with three functions for manipulating the page variable, ensuring it never strays outside the acceptable range (Flickr starts its page numbering at 1, in case you were wondering). The `next()` and `previous()` functions will be called by outside classes to cause the gallery to advance or retreat.

We'll skip over `loadPage()` for now (we'll come back to it in a moment). The `assignThumbs()` and `unassignThumbs()` functions do what their name suggests. The first takes a `FlickrResult`, as retrieved from the web service, and populates the `ImageView` nodes in the thumbnail bar with fresh `Image` content. The second clears the `thumbImages` sequence, to remove the thumbnails from the bar.

The `loadPage()` function is the code ultimately responsible for responding to each request to fetch a fresh page of thumbnails from the web service. The entire function is based on a strange new type of operation called a *transition*. We've yet to see a transition in any of the projects so far, so let's stop and examine it in detail.

8.3.2 The easy way to animate: transitions

So far, whenever we wanted to animate something we had to build a `Timeline` and use it to manipulate the variables we wanted to change. It doesn't take a rocket scientist to realize a handful of common node properties (location, rotation, opacity, etc.) are frequent targets for animation. We could save a lot of unnecessary boilerplate if we created a library of prebuilt animation classes, pointed them at the nodes we wanted to animate, and then let them get on with the job.

If you haven't guessed by now, this is what *transitions* are. Let's have another look at the code in the last section:

```
TranslateTransition {
    node: thumbGroup;
    byX: 0;  byY: height;
    interpolator: Interpolator.EASEIN;
    duration: 0.5s;
    action: function() {
        unassignThumbs();
        thumbGroup.translateY = 0;
        service.loadPage(page);
    }
}.play();
```

The `TranslateTransition` is all about movement. It has a host of different configuration options that allow us to move a node *from* a given point, *to* a given point, or *by* a given distance. In our example we're moving the entire thumbnail group downward by the height of the thumbnail bar, which (one assumes) would have the effect of sending the thumbnails off the bottom of the screen.

When the transition is over, we have the opportunity to run some code. That's the purpose of the `action` function type. When this code runs, we know the thumbnails will be off screen, so we unassign them to get rid of the current images. Then we move the group we just animated back to its starting position, by resetting `translateY` (note: `translateY`, not `layoutY`, because `TranslateTransition` doesn't change its target node's layout position). And finally, we ask the web service to load a fresh page of thumbnails. This call will return immediately, as the web service interaction is carried out on another thread. In listing 8.9 we saw that the web service invokes `assignThumbs()` when its data has successfully loaded, so the images we just deleted with `unassignThumbs()` should start to be repopulated once the web service code is finished.

The call to `play()`, in case you haven't realized, fires the transition into action.

8.3.3 The main photo desktop: the PhotoViewer class

To round off our project we're going to look at clicking and dragging bits of the scene graph, and we'll play with even more types of transition. It all takes place in the `PhotoViewer` class, of which listing 8.12 is the first part. PhotoViewer.fx is divided into four parts; the final one in listing 8.15.

Listing 8.12 PhotoViewer.fx (part 1)

```
package jfxia.chapter8;

import javafx.animation.Interpolator;
import javafx.animation.transition.FadeTransition;
import javafx.animation.transition.ParallelTransition;
import javafx.animation.transition.ScaleTransition;
import javafx.animation.transition.RotateTransition;
import javafx.animation.transition.TranslateTransition;
import javafx.geometry.Bounds;
import javafx.scene.*;
import javafx.scene.control.Button;
import javafx.scene.control.ProgressBar;
import javafx.scene.image.*;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.*;
import javafx.scene.layout.LayoutInfo;
import javafx.scene.layout.Tile;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Math;

import java.lang.System;
import java.util.Random;

def buttonW:Number = 90;
def buttonH:Number = 35;
def maxImages:Integer = 6;
def rand:Random = new Random(System.currentTimeMillis());

var sc:Scene;

def exitButton = Button  {
    text: "Exit";
    layoutX: bind sc.width - buttonW;
    width: buttonW;  height: buttonH;
    action: function() { FX.exit(); }
};

def mouseHandler:Rectangle = Rectangle {
    var dragX:Number;
    var dragY:Number;
    width: bind sc.width;
    height: bind sc.height;
    opacity: 0;
    onMousePressed: function(ev:MouseEvent) {
        dragX=ev.x;  dragY=ev.y;
    }
```

Constants and utility objects

Exit button

Capture desktop click events

Remember initial position

```
    onMouseDragged: function(ev:MouseEvent) {
        var idx:Integer = 0;
        for(i in desktopGroup.content) {
            def v:Number = 1.0 + idx/3.0;
            i.layoutX += (ev.x-dragX) * v;
            i.layoutY += (ev.y-dragY) * v;
            idx++;
        }
        dragX=ev.x;  dragY=ev.y;
    }
}
def desktopGroup:Group = Group {}
```

Move all photos with parallax

Photos added here

```
// ** Part 2 is listing 8.13; part 3, listing 8.14; part 4, listing 8.15
```

At the head of the source we define a few constants, such as the button size and maximum number of photos on the desktop. We also create an instance of Java's random number class, which we'll use to add a degree of variety to the animations. JFX's `javafx.util.Math` class has a `random()` function, but it only returns a `Double` between 0 and 1. (Although it has fewer options, our code would be able to run outside the desktop, where the full Java SE libraries aren't necessarily available.)

To get the dimensions of the screen we reference the application's `Scene`, which is what the `sc` variable is for. JavaFX has a class called `javafx.stage.Screen`, detailing all the available screens on devices that accommodate multiple monitors. We could tie the layout to the data in `Screen`, but referencing `Scene` instead makes it easier to adapt the code so it no longer runs full screen.

The exit button, imaginatively named `exitButton`, comes next. Then we define an invisible `Rectangle`, for capturing desktop mouse clicks and drag events. When the user clicks an empty part of the desktop, the event will be directed to this shape. On button down it stores the x/y position, and as drag events come in it moves all the nodes inside a `Group` called `desktopGroup`. This group is where all the desktop photos are stored. As each is moved, the code adds a small scaling factor, so more recent (higher) photos move farther than earlier (lower) ones, creating a cool parallax effect. (My friend Adam wasn't so keen on the parallax effect when I showed it to him, but what does he know?)

Moving on, listing 8.13 shows us our `GalleryView` class in action.

Listing 8.13 PhotoViewer.fx (part 2)

```
// ** Part 1 is listing 8.12
def galView:GalleryView = GalleryView
{   layoutY: bind (sc.height-galView.height);
    width: bind sc.width-100;

    apiKey: "-";  // <== Your key goes here
    userId: "29803026@N08";

    action: function(ph:FlickrPhoto ,
        im:Image,x:Number,y:Number) {
        def pv:Node = createPhoto(ph,im);
```

Thumbnail bar

Position and size

Flickr details

Thumbnail clicked

```
        def pvSz:Bounds = pv.layoutBounds;
        def endX:Number = (sc.width-pvSz.width) / 2;
        def endY:Number = (sc.height-pvSz.height) / 2;
        ParallelTransition {                                    <- Perform transitions together
            node: pv;
            content: [
                TranslateTransition {                           <- Move: onto desktop from thumb bar
                    fromX: galView.layoutX + x;
                    fromY: galView.layoutY + y;
                    toX: endX;
                    toY: endY;
                    interpolator: Interpolator.EASEIN;
                    duration: 0.5s;
                } ,
                ScaleTransition {                               <- Scale: quarter to full
                    fromX: 0.25;  fromY: 0.25;
                    toX: 1.0;  toY: 1.0;
                    duration: 0.5s;
                },
                RotateTransition {                              <- Rotate: Full 360, plus/minus random
                    fromAngle: 0;
                    byAngle: 360 + rand.nextInt(60)-30;
                    duration: 0.5s;
                }
            ];
        }.play();

        insert pv into desktopGroup.content;                    <- Add photo to desktopGroup

        if((sizeof desktopGroup.content) > maxImages) {         <- Too many photos on desktop?
            FadeTransition {
                node: desktopGroup.content[0];
                fromValue:1;  toValue:0;
                duration: 2s;
                action: function() {
                    delete desktopGroup.content[0];
                }
            }.play();
        }
    };
};
// ** Part 3 is listing 8.14; part 4, listing 8.15
```

Here's a really meaty piece of code for us to chew on. The `action` event does all manner of clever things with transitions to spin the photo from the thumbnail bar onto the desktop. But let's look at the other definitions first. We need the thumbnail bar to sit along the bottom of the screen, so the `width` and `layoutY` are bound to the `Scene` object's size. We then plug in the familiar Flickr `apiKey` and `userID` (remember to provide your own key).

The `action` function runs when someone clicks a thumbnail in the `GalleryView`. It passes us the `FlickrPhoto` object, a copy of the thumbnail-size image (we'll use this as a stand-in while the full-size image loads), and the coordinates of the image within the bar so we can calculate where to start the movement transition.

We need a photo to work from, and the `createPhoto()` is a handy private function that creates one for us, complete with white border, shadow, loading progress bar, and the ability to be dragged around the desktop. We'll examine its code at the end of the listing; for now accept that `pv` is a new photo to be added to the desktop. The `endX` and `endY` variables are the destination coordinates on the desktop where the image will land. We use the photo's dimensions, from `layoutBounds`, to center it.

The `ParallelTransition` is another type of animation transition, except it doesn't have any direct effect itself. It acts as a group, playing several other transitions at the same time. In our code we use the `ParallelTransition` with three other transitions, represented in figure 8.6.

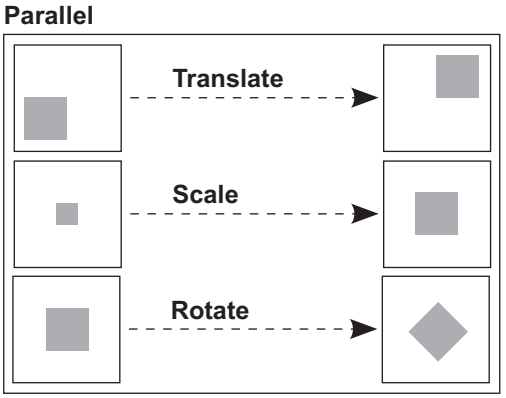

Figure 8.6 All three transitions, translate (movement), scale, and rotate, are performed at the same time to the same scene graph node.

The `TranslateTransition` you've already seen; it moves the image from the thumbnail bar into the center of the desktop. At the same time the `ScaleTransition` makes it grow from a quarter of its size to full size, over the same period of time. The `RotateTransition` spins the node by 360 degrees, plus or minus a random angle, no more than 30 degrees either way. Note how we only have to specify the actual target node at the `ParallelTransition` level, not at every transition contained within it. We fire off the transition to run in the background and add the new photo to the `desktopGroup`, to ensure it's displayed on the screen.

The final block of code uses yet another transition, this time a `FadeTransition`. We don't want too many photos on the desktop at a time, so once the maximum has been exceeded we kick off a fade to make the oldest photo gracefully vanish, and then (using an `action`) we delete it from the `desktopGroup`.

In the third part of the `PhotoViewer` code (listing 8.14) we'll tie off the loose ends, getting ready to build the scene graph.

Listing 8.14 PhotoViewer.fx (part 3)

```
// ** Part 1 is listing 8.12; part 2, listing 8.13
def controls:Tile = Tile {
    def li = LayoutInfo { width: buttonW;  height: buttonH }
    layoutX: bind sc.width - buttonW;
    layoutY: bind sc.height - controls.layoutBounds.height;
    hgap: 5;
    columns: 1;
    content: [
        Button {
            text: "Clear";
            layoutInfo: li;
```

Clear/prev/next buttons

Clear: empty desktopGroup

```
            action: function() { desktopGroup.content = []; }
        } ,
        Button {
            text: "Next";
            layoutInfo: li;
            action: function() { galView.next(); }
        } ,
        Button {
            text: "Previous";
            layoutInfo: li;
            action: function() { galView.previous(); }
        }
    ],
}

Stage {
    scene: sc = Scene {
        content: [
            mouseHandler, desktopGroup,
            galView, controls, exitButton
        ];
        fill: LinearGradient {
            endX: 1;  endY: 1;  proportional: true;
            stops: [
                Stop { offset: 0;  color: Color.WHITE; } ,
                Stop { offset: 1;  color: Color.GOLDENROD; }
            ];
        };
    }
    fullScreen: true;
};
// ** Part 4 is listing 8.15
```

Clear: empty desktopGroup

Next: next page

Previous: last page

Add to application scene

Full screen, please

Listing 8.14 is quite tame compared to the previous two parts of `PhotoViewer`. Three JavaFX buttons allow the user to navigate the gallery using the exposed functions in `GalleryView` and to clear the desktop by emptying `desktopGroup.contents`. Then we build the actual `Stage` itself by combining the screenwide mouse hander, photo group, gallery, and buttons. The `Scene` background is set to a pleasant yellow/gold tint, a suitable backdrop for our photos. And speaking of photos, listing 8.15 has the final piece of code in this source file, a function to build our photo nodes.

Listing 8.15 PhotoViewer.fx (part 4)

```
// ** Part 1 is listing 8.12; part 2, listing 8.13; part 3, listing 8.14
function createPhoto(photo:FlickrPhoto, image:Image) : Node {
    var im:Image;
    var iv:ImageView;
    var pr:ProgressBar;

    def w:Number = bind iv.layoutBounds.width + 10;
    def h:Number = bind iv.layoutBounds.height + 10;

    def n:Group = Group {
        var dragOriginX:Number;
        var dragOriginY:Number;
```

Photo size (without shadow)

Drag variables

```
        content: [
            Rectangle {
                layoutX: 15;  layoutY: 15;
                width: bind w;  height: bind h;
                opacity: 0.25;
            } ,
            Rectangle {
                width: bind w;  height: bind h;
                fill: Color.WHITE;
            } ,

            iv = ImageView {
                layoutX: 5;  layoutY: 5;
                fitWidth: FlickrPhoto.MEDIUM;
                fitHeight: FlickrPhoto.MEDIUM;
                preserveRatio: true;
                //smooth:true;
                image: im = Image {
                    url: photo.urlMedium;
                    backgroundLoading: true;
                    placeholder: image;
                };
            } ,
            pr = ProgressBar {
                layoutX: 10;  layoutY: 10;
                progress: bind im.progress/100.0;
                visible: bind (pr.progress<1.0);
            }
        ];

        blocksMouse: true;
        onMousePressed: function(ev:MouseEvent) {
            dragOriginX=ev.x;  dragOriginY=ev.y;
        }
        onMouseDragged: function(ev:MouseEvent) {
            n.layoutX += ev.x-dragOriginX;
            n.layoutY += ev.y-dragOriginY;
        }
    };
}
```

Shadow rectangle

White border rectangle

ImageView displays thumb or photo

Progress bar bound to loading

Mouse events stop here

Record click start coords

Update coordinates from drag

We've saved the best for last; this is real heavy-duty JavaFX Script coding! The `createPhoto()` function constructs a scene graph node to act as our full-size desktop photo, but it does more than just that. It:

- Creates a white border and shadow to fit around the photo, matched to the correct dimensions of the image.
- Displays a scaled thumbnail and kicks off the loading of the full-size image.
- Displays a progress meter while we wait for the full image to load.
- Allows itself to be dragged by the mouse, within its parent (the desktop).

At the top of the function we define some variables to reference parts of the scene graph we're creating. The variables `w` and `h` are the size of the photo, including its white border. The first `Rectangle` inside our `Group` is the shadow, using the default

color (black) set to a quarter opacity. I experimented with using the `DropShadow` effect class inside `javafx.scene.effect` but found it had too much of a detrimental impact on the application's frame rate, so this `Rectangle` is a kind of poor man's alternative. The shadow is followed by the white photo border, and this in turn is followed by the `ImageView` to display our photo.

We make sure the image is sized to fit the proportions of Flickr's medium-size photo, and we point to the medium photo's URL from the `FlickrPhoto` object. Here's the clever part: we assign the thumbnail as the placeholder while we wait for the full-size image to load. This ensures we get *something* to display immediately, even if it's blocky.

The progress bar completes the scene graph contents. It will be visible only so long as there's still some loading left to be done.

That's the scene graph; now it's time to look at the event handlers.

```
blocksMouse: true;
onMousePressed: function(ev:MouseEvent) {
    dragOriginX=ev.x;  dragOriginY=ev.y;
}
onMouseDragged: function(ev:MouseEvent) {
    n.layoutX += ev.x-dragOriginX;
    n.layoutY += ev.y-dragOriginY;
}
```

The example is the chunk of code responsible for allowing us to drag the node around the desktop display. The `blocksMouse` setting is very important; without it we'd get all manner of bizarre effects whenever we moved a photo. When the mouse button goes down on a given part of the screen, there may be numerous scene graph nodes layered beneath it, so what gets the event? The highest node? The lowest node? All of them?

The answer is "all of them," working from front to back, unless we take action to stop it. If we allowed mouse events to be applied to both a photo and the underlying desktop, we'd get two sets of actions at once. With some applications this might be desirable, but in our case it is not; the event should go to either an individual photo or the desktop, but not both! The handy `blocksMouse` variable can be set to prevent mouse events from traveling any farther down the stack of nodes.

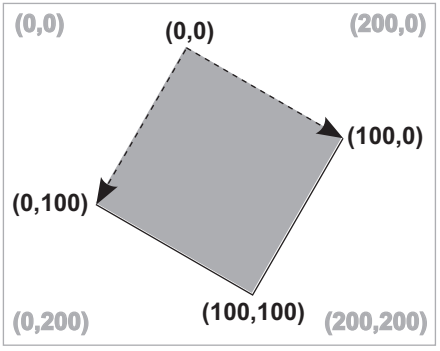

Figure 8.7 The coordinate system of a node always matches the rotation of the node itself. The gray rectangle has been rotated 30 degrees clockwise, yet its local coordinate system is unaffected.

The remainder of the event code should be fairly obvious. When the mouse goes down we record its start position, relative to the top-left corner of the node. Even though the node will be rotated on the desktop, the mouse coordinates still work in sync with the rectangular shape of the photo, because the coordinates are local to the interior of the node (see figure 8.7). Drag events then

update the layout position of the node within its parent, causing it to move inside the desktop.

Now that our JavaFX Script code is done and dusted, we can try running the application.

8.3.4 Running the application

Before running the application, make sure you've set your own Flickr API key (the one you got after registering) into the `apiKey` variable in listing 8.13. Failure to do so will mean the code will exit with an exception.

Once the application is running, the screen should look like figures 8.1 and 8.3. A paged thumbnail image display should appear at the foot of the screen, with clicked images flying onto the main desktop. Initially the full-size image will lack detail, as a copy of the thumbnail is scaled up to act as a placeholder while the high-res version loads. After a second or two, however, the scaled thumbnail should be replaced by the real image (depending on how fast your internet connection is).

Clicking and dragging the images on the desktop will move them around, while clicking and dragging an empty part of the desktop will move all the desktop images currently displayed. The desktop will show only a handful of images at a time; older images will fade away as new ones are added.

The application demonstrates a lot of manipulation of nodes, including scaling, rotation, and movement. This seems an ideal time to discuss how JavaFX calculates the size (bounds) of each part of the scene graph when various transformations are applied. So that's what we'll do next.

8.4 Size matters: node bounds in different contexts

In this project we had a lot of rotating and scaling of nodes, and we saw prominent use of `layoutBounds` and other node properties. If you browsed the JavaFX API documentation, you may have noticed that every node has several `javafx.geometry.Bounds` properties, publicizing its location and size. Each defines two x and y coordinates: top-left corner and bottom-right corner. But why so many `Bounds`—wouldn't one be enough, and how do they relate to each other?

The truth is they detail the location of the node at different points during its journey from abstract shape in a scene graph, to pixels on the video screen. The journey looks like this:

1. We start with the basic node.
2. At this point `layoutBounds` is calculated.
3. Any effects are performed (reflections, drop shadow, etc.), as specified by the node's `effect` property.
4. If the `cache` property is set to true, the node may be cached as a bitmap to speed future updates.
5. Opacity is applied next, as per the `opacity` property.
6. The node is clipped according to its `clip` property, if set.
7. At this point `boundsInLocal` is calculated.

8. Any `transforms` are then applied, translating, rotating, and scaling the node.
9. The node is scaled, by `scaleX` and `scaleY`.
10. The `rotate` property is now applied.
11. The node origin is moved within its parent, using `layoutX/translateX` and `layoutY/translateY`.
12. At this point `boundsInParent` is calculated.

In layperson's terms `layoutBounds` is the size the node thinks it is. Most of the time these are the bounds you'll use to read the node's size or lay it out. Next, `boundsInLocal` accommodates any nontransforming alterations (drop shadow, clip, etc.) applied to the node. Usually effects are ignored during layout (we usually want two nodes placed side by side to touch, even if they have surrounding drop shadows), so you'll probably find `boundsInLocal` is only infrequently needed in your code. Finally, `boundsInParent` exposes the true pixel footprint of the node inside its parent, once all effects, clips, and transformation operations have been applied. Since the scene graph translates coordinate spaces automatically, including for events, you'll likely find there are very few circumstances where you actually need to reference `boundsInParent` yourself.

8.5 Summary

This has been quite a long project, and along the way we've dealt with a lot of important topics. We started by looking at addressing a web service and parsing the XML data it gave back. This is a vital skill, as the promise of *cloud computing* is ever more reliance on web services as a means of linking software components. Although we addressed only one Flickr method, the technique is the same for all web service calls. You should be able to extend the classes in the first part of our project to talk to other bits of the Flickr API with ease. Why not give it a try?

In the main application we threw nodes around the screen with reckless abandon, thanks to our new friend the transition. Transitions take the sting out of creating beautiful UI animation, and as such they deserve prime place in any JavaFX programmer's toolbox. Why not experiment with the other types of transition `javafx.animation.transition` has to offer?

You'll be glad to know that not only do we now have a nice little application to view our own (or someone else's) photos with, but we've passed an important milestone on our journey to master JavaFX. With this chapter we've now touched on most of the important scene graph topics. Sure, we didn't get to use *every* transition, or try out *every* different effect, and we didn't even get to play with *every* different type of shape in the `javafx.scene.shape` package, but we've covered enough of a representative sample that you should now be able to find your way around the JavaFX API documentation without getting hopelessly lost.

In the next chapter we'll move away from purely language and API concerns to look at how designers and programmers can work together and how to turn our applications into applets. Until then, I encourage you make this project's code your own—add a text field for the ID of the gallery to view, and have a lot more fun with transitions.