

# *From app to applet*

---

## ***This chapter covers***

- Turning a desktop app into a web applet
- Explaining designer/programmer workflow
- Manipulating Adobe/SVG artwork
- Designing entire UIs in graphics programs

Previous chapters have concerned themselves with language syntax and programming APIs—the nuts and bolts of coding JavaFX. In this chapter we’re focusing more on how JavaFX fits into the wider world, both at development time and at deployment time.

One of the goals of JavaFX was to recognize the prominent role graphic artists and designers play in modern desktop software and bring them into the application development process in a more meaningful way. To this end Sun created the JavaFX Production Suite, a collection of tools for translating content from the likes of Adobe Photoshop and Illustrator into a format manipulable by a JavaFX program.

Another key goal of JavaFX was to provide a single development technology, adaptable to many different environments (as mentioned in the introductory chapter). The 1.0 release concentrated on getting the desktop and web right; phones followed with 1.1 in February 2009, and the TV platform is expected

sometime in 2009/10. The snappily titled “release 6 update 10” of the Java Runtime Environment (JRE) offers an enhanced experience for desktop applications and web applets. The new features mainly center on making it less painful for end users to install updates to the JRE, easier for developers to ensure the right version of Java is available on the end user’s computer, and considerably easier for the end user to rip their favorite applet out of the browser (literally) and plunk it down on the desktop as a standalone application.

In this chapter we’re going to explore how JavaFX makes life easier for nonprogrammers, focusing specifically on improvements in the designer and end-user experience. First we’ll have some fun developing UI widgets from SVG (Scalable Vector Graphics) images, turning them into scene graph nodes, and manipulating them in our code. Then we’ll transform the application into an applet to run inside a web browser. As with previous chapters, you’ll be learning by example. So far most of our projects have centered on visuals; it’s about time we developed something with a little more practical value. Increasingly serious applications are getting glossy UIs, and you don’t get more serious than when the output from your program could change the fate of nations.

## 9.1 The Enigma project

These days practically everyone in the developed world has used encryption. From online shopping to cell phone calls, encryption is the oil that lubricates modern digital networks. It’s interesting to consider, then, that machine-based encryption (as opposed to the pencil-and-paper variety) is a relatively recent innovation. Indeed, not until World War II did automated encryption achieve widespread adoption, with one technology in particular becoming the stuff of legend.

The Enigma machine was an electro-mechanical device, first created for commercial use in the early 1920s, but soon adopted by the German military for scrambling radio Morse code messages between commanders and their troops in the field. With its complex, ever-shifting encoding patterns, many people thought the Enigma was unbreakable, presumably the same people who thought the Titanic was unsinkable. As it happens, the Enigma cipher *was* broken, not once but twice!

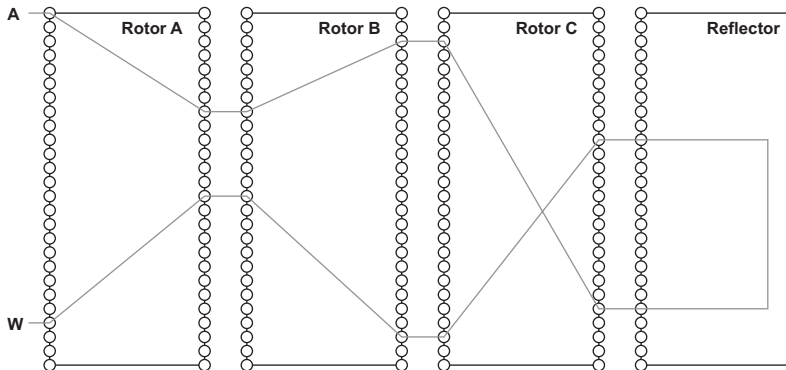
Creating our own Enigma machine gives us a practical program with a real-world purpose, albeit one 60 years old. Its UI demands may be limited but offer a neat opportunity to cover important JavaFX techniques for turbocharging UI development (plus it’s an excuse to write JFX code that isn’t strictly UI related). Having reconstructed our own little piece of computing history, we’ll be turning it from a desktop application into an applet and then letting the end user turn it back into a desktop application again with the drag and drop of a mouse.

### 9.1.1 The mechanics of the Enigma cipher

Despite its awesome power, the Enigma machine was blissfully simple by design. Pre-dating the modern electronic computer, it relied on old-fashioned electric wiring and mechanical gears. The system consisted of four parts: three rotors and a reflector.

Each rotor is a disk with 52 electrical contacts, 26 on one face and another 26 on the other. Inside the disk each contact is wired to another on the opposite face. Electric current entering at position 1 might exit on the other side in position 8, for example. The output from one rotor was passed into the next through the contacts on the faces, forming a circuit.

In figure 9.1 current entering Rotor A in position 1 leaves in position 8. Position 8 in the next rotor is wired to position 3, and 3 in the last rotor is wired to 22. The final component, the reflector, directs the current back through the rotors in the reverse direction. This ensures that pressing A will give W and (more important, when it comes time to decode) pressing W will give A.



**Figure 9.1** Tracing one path inside the Enigma, forming a circuit linking key A with lamp W and key W with lamp A. But as the rotors move, the circuit changes to provide a different link.

Each time a letter is encoded, one or more rotors turn, changing the combined circuit. This constant shifting of the circuit is what gives the Enigma part of its power; each letter is encoded differently with successive presses. To decode a message successfully, one must set the three rotor disks to the same start position as when the message was originally encoded. Failure to do so results in garbage.

## 9.2 Programmer/designer workflow: Enigma machine, version 1

The first stab we're going to have at an Enigma machine will put only the basic encryption and input and output components in place. In this initial version of the project we'll stick to familiar ground by developing a desktop application; the web applet will come later. You can see what the application will look like from figure 9.2. In the next version we'll flesh it out with a nicer interface and better features.

To create the key and lamp graphics we'll be using two SVG files, which we'll translate to JavaFX's own FXZ file format using the tools in the JavaFX Production Suite. In keeping with this book's policy of not favoring one platform or tool or IDE, I used the open source application Inkscape to draw the graphics and the SVG converter tool to



**Figure 9.2** Our initial version of the Enigma machine will provide only basic encryption, input, and output, served up with a splash of visual flare, naturally!

turn them into JavaFX scene graph-compatible files. If you are lucky enough to own Adobe Photoshop CS3 or Illustrator CS3, you can use plug-ins available in the suite to export directly into JavaFX's FXZ format from inside those programs.

The SVG/IDE-agnostic route isn't that different from working with the Adobe tools and using the NetBeans JFX plug-in. For completeness I'll give a quick rundown of the Adobe/NetBeans shortcuts at the end of the first version.

### 9.2.1 Getting ready to use the JavaFX Production Suite

To join in with this chapter you'll need to download a few things. The first is the JavaFX Production Suite, available from the official JavaFX site; download this and install it. The JavaFX Production Suite is a separate download to the SDK because the suite is intended for use by designers, not programmers.

#### JavaFX v1.0 and the Production Suite

Are you being forced to use the old 1.0 version of JavaFX? If you find yourself maintaining ancient code, you need to be aware that JavaFX 1.0 did not include the FXD library by default, as its successors do. To use the FXZ/FXD format with a JavaFX 1.0 application, you'll need to download the Production Suite and include its `javafx-fxd-1.0.jar` file on your classpath. You'll also need to ship it with your application, along with other JARs your code depends on.

The second download is the project source code, including the project's SVG and FXZ files, available from this book's web page. Without this you won't have the graphics files to use in the project.

The final download is optional: Inkscape, an open source vector graphics application, which you can use to draw SVG files yourself. Because the SVG files are already available to you in the source code download, you won't need Inkscape for this project,

but you may find it useful when developing your own graphics. As noted, you could alternatively use Photoshop CS3 (for bitmap images) or Illustrator CS3 (for vector images), if you have access to them.

### The links

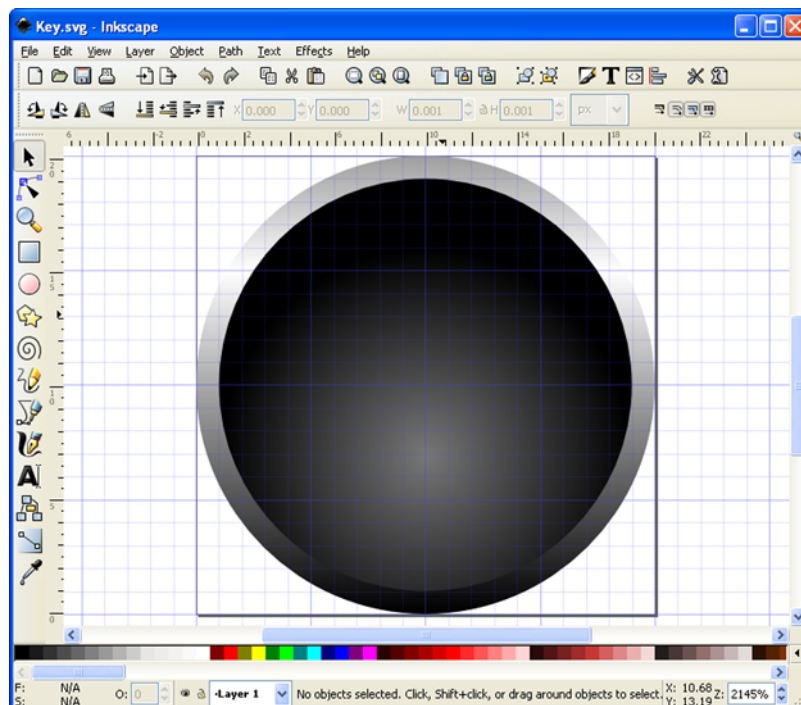
<http://javafx.com/> (follow the download link)  
<http://www.manning.com/JavaFXinAction>  
<http://www.inkscape.org/>

If you downloaded the source code, you should have access to the SVG files used to create the images in this part of the project.

## 9.2.2 Converting SVG files to FXZ

Scalable Vector Graphics is a W3C standard for vector images on the web. Vector images, unlike their bitmap cousins, are defined in terms of geometric points and shapes, making them scalable to any display resolution. This fits in nicely with the way JavaFX's scene graph works.

There are two files supplied with the project, `key.svg` and `lamp.svg`, defining the vector images we'll need. They're in the `svg` directory of the project. Figure 9.3 shows the



**Figure 9.3** SVG images are formed from a collection of shapes; the key is two circles painted with gradient fills. JavaFX also supports layered bitmaps from Photoshop and vector images from Illustrator.

key image being edited by Inkscape. To bring these images into our JavaFX project we need to translate them from their SVG format into something closer to JavaFX Script. Fortunately for us, the JavaFX Production Suite comes with a tool to do just that.

The format JavaFX uses is called FXZ (at least, that's the three-letter extension its files carry), which is a simple zip file holding one or more components. Chief among these is the FXD file, a JavaFX Script scene graph declaration. There may also be other supporting files, such as bitmap images or fonts. (If you want to poke around inside an FXZ, just change its file extension from .fxz to .zip, and it should open in your favorite zip application.)

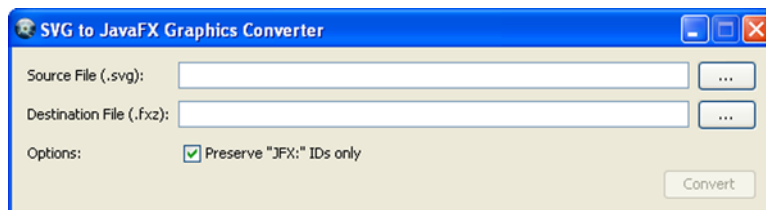
For each of our SVG files the converter parses the SVG and outputs the equivalent JavaFX Script code, which it stores in the FXD file. Any supporting files (for example, texture bitmaps) are also stored, and the whole thing is zipped up to form an FXZ file.

Once the JavaFX Production Suite is installed, the SVG to JavaFX Graphics Converter should be available as a regular desktop application (for example, via the Start menu on Windows). You can see what it looks like in figure 9.4.

To run the converter we need to provide the filenames of the source SVG and the destination FXZ. We can also choose whether to retain `jfx:` IDs during the conversion, which demands extra explanation.

While creating the original SVG file (and this is also true of Illustrator and Photoshop files), it is possible to mark particular elements and expose them to JavaFX after the conversion. We do this by assigning the prefix `jfx:` (that's `jfx` followed by a colon) to the layer or element's ID. When the conversion tool sees that prefix on an ID, it stores a reference to the element in the FXD, allowing us to later address that specific part of the scene graph in our own code. Later, when we play with the lamp graphic, we'll see an example of doing just that. In general you should ensure that the option is switched on when running the tool.

**TIP** *Check your ID* When I first attempted to use FXZ files in my JavaFX programs, the FXD reader didn't seem to find the parts of the image I'd carefully labeled with the `jfx:` prefix. After 15 minutes of frustration, I realized my schoolboy error: naming the layers of a SVG is not the same as setting their IDs. The JavaFX Production Suite relies on the ID only. So if, like me, you experience trouble finding parts of your image once it has been loaded into JavaFX, double-check the IDs on your original SVG.



**Figure 9.4** The SVG Converter takes SVG files, using the W3C's vector format, and translates them into FXZ files, using JavaFX's declarative scene graph markup.

The JavaFX Production Suite also comes with a utility to display FXZ files, called the JavaFX Graphics Viewer. After generating your FXZ, you can use it to check the output. If you haven't done so already, try running the converter tool and generating FXZ files from the project's SVGs; then use the viewer to check the results.

We'll use the resulting FXZ files when we develop the lamp and key classes. Right now you need to be aware that the FXZ files should be copied into the `jfxia.chapter9` package of the build, so they live next to the two classes that load them; otherwise the application will fail when you run it.

### 9.2.3 The Rotor class: the heart of the encryption

The Rotor class is the heart of the actual encryption code. Each instance models a single rotor in the Enigma. Its 26 positions are labeled A to Z, but they should not be confused with the actual letters being encoded or decoded. The assigning of a letter for each position is purely practical; operators needed to encode rotor start positions into each message but the machine had no digit keys, so rotors were labeled A–Z rather than 1–26. For convenience we'll also configure each rotor using the letter corresponding to each position. Since the current can pass in either direction through the rotor wiring, we'll build two lookup tables, one for left to right and one for right to left. Listing 9.1 has the code.

**Listing 9.1 Rotor.fx (version 1)**

```
package jfxia.chapter9;

package class Rotor {
    public-init var wiring:String;
    public-init var turnover:String;
    public-read var rotorPosition:Integer = 0;
    public-read var isTurnoverPosition:Boolean = bind
        (rotorPosition == turnoverPosition);

    var rightToLeft:Integer[];
    var leftToRight:Integer[];
    var turnoverPosition:Integer;

    init {
        rightToLeft = for(a in [0..<26]) { -1; }
        leftToRight = for(a in [0..<26]) { -1; }
        var i=0;
        while(i<26) {
            var j:Integer = chrToPos(wiring,i);
            rightToLeft[i]=j;
            leftToRight[j]=i;
            i++;
        }

        if(isInitialized(turnover))
            turnoverPosition = chrToPos(turnover,0);
    }

    package function encode(i:Integer,leftwards:Boolean) : Integer {
        var res = (i+rotorPosition) mod 26;
```

**Wiring  
connections  
set as string**

**When should  
next rotor move?**

**Encode an input**



```

    var r:Integer = if(leftwards) rightToLeft[res]
                    else leftToRight[res];
    return (r-rotorPosition+26) mod 26;
}
package function nextPosition() : Boolean {
    rotorPosition = if(rotorPosition==25) 0
                    else rotorPosition+1;
    return isTurnoverPosition;
}
}

package function posToChr(i:Integer) : String {
    var c:Character = (i+65) as Character;
    return c.toString();
}
package function chrToPos(s:String,i:Integer) : Integer {
    var ch:Integer = s.charAt(i);
    return ch-65;
}
}

```

↑  
**Encode an input**

|  
**Step to next  
position, returning  
turnover**

|  
**Convert  
between letter  
and position**

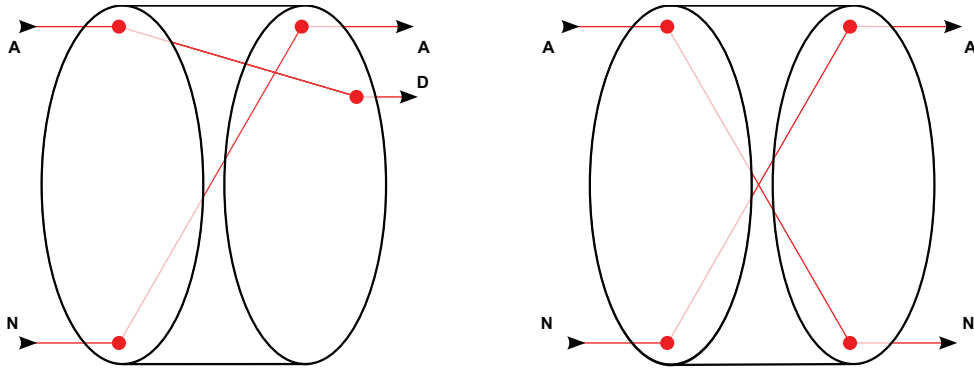
Looking at listing 9.1 we can see the wiring is set using a `String`, making it easy to declaratively create new rotor objects. Other public variables control how the encryption works and how the rotors turn.

- The variable `wiring` is the source for two lookup tables, used to model the flow of current as it passes through the rotor. The tables created from `wiring` determine how the contacts on the rotor faces are linked: `rightToLeft` gives the outputs for positions A to Z (0 to 25) on one face, and for convenience `leftToRight` does the reverse path from the other face. If the first letter in `wiring` was D, for example, the first entry of `rightToLeft` would be 3 (labeled D because A = 0, B = 1, etc.), and the fourth entry of `leftToRight` would be 0 (labeled A). Thus 0 becomes 3 right to left, and 3 becomes 0 left to right.
- The variable `turnover` is the position (as a letter) at which the next rotor should step (like when a car odometer digit moves to 0, causing the next-highest digit to also move). The private variable `turnoverPosition` is the turnover in its more useful numeric form. One might have expected a rotor to do a full A-to-Z cycle before the next rotor ticks over one position, but the Enigma designers thought this was too predictable. The `isTurnoverPosition` variable is a handy bound flag for assessing whether this rotor is currently at its turnover position.
- The `rotorPosition` property is the current rotation offset of this rotor, as an `Integer`. If this were set to 2, for example, an input for position 23 would actually enter the rotor at position 25.

So that's our Rotor class. Each rotor provides one part of the overall encryption mechanism. We can use a Rotor object to create the reflector too; we just need to ensure the wiring string models symmetrical paths. Figure 9.5 show how this might look.

The regular rotors are symmetrical only by reversing the direction of current flow. Just because N (left input) results in A (right output), does not mean A will result in N





**Figure 9.5** Two disks, with left/right faces. The rotor wiring is not symmetrical (left), but we can create a reflector from a rotor by ensuring 13 of the wires mirror the path of the other 13 (right).

when moving in the same left-to-right direction. Figure 9.5 shows this relationship in its left-hand rotor. We can, however, conspire to create a rotor in which these connections are deliberately mirrored (see figure 9.5's right-hand rotor), and this is precisely how we model the reflector. This convenience saves us from needing to create a specific reflector class.

### 9.2.4 A quick utility class

Before we proceed with the scene graph classes, we need a quick utility class to help position nodes. Listing 9.2, which does that, is up next.

#### Listing 9.2 Util.fx

```
package jfxia.chapter9;

import javafx.scene.Node;

package bound function center(a:Node,b:Node,hv:Boolean) : Number {
    var aa:Number = if(hv) a.layoutBounds.width
    else a.layoutBounds.height;
    var bb:Number = if(hv) b.layoutBounds.width
    else b.layoutBounds.height;
    return ((aa-bb) /2);
}

package bound function center(a:Number,b:Node,hv:Boolean) : Number {
    var bb:Number = if(hv) b.layoutBounds.width
    else b.layoutBounds.height;
    return ((a-bb) /2);
}
```

The two functions in listing 9.2 are used to center one node inside another. The first function centers node *b* inside node *a*; the second centers node *b* inside a given dimension. In both cases the boolean *hv* controls whether the result is based on the width (true) or the height (false) of the parameter nodes.

### 9.2.5 The Key class: input to the machine

The real Enigma machine used keys and lamps to capture input and show output. To remain faithful to the original we'll create our own Key and Lamp custom nodes. The Key is first; see listing 9.3.

#### Listing 9.3 Key.fx

```
package jfxia.chapter9;

import javafx.fxml.FXMLNode;
import javafx.scene.Node;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.input.MouseEvent;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class Key extends CustomNode {
    package def diameter:Number = 40;
    package def fontSize:Integer = 24;

    public-init var letter:String on replace {
        letterValue = letter.charAt(0)-65;
    }
    package var action:function(:Integer,:Boolean):Void;

    def scale:Number = bind if(pressed) 0.9 else 1.0;
    def letterFont:Font = Font.font(
        "Courier", FontWeight.BOLD, fontSize
    );
    var letterValue:Integer;

    override function create() : Node {
        def keyNode = FXDNode {
            url: "{__DIR__}key.fxz";
        }

        keyNode.onMousePressed = function(ev:MouseEvent) {
            if(action!=null)
                action(letterValue,true);
        };
        keyNode.onMouseReleased = function(ev:MouseEvent) {
            if(action!=null)
                action(letterValue,false);
        };

        Group {
            var k:Node;
            var t:Node;
            content: [
                k = keyNode ,
                t = Text {
                    layoutX: bind Util.center(k,t,true);
                }
            ]
        }
    }
}
```

Key's  
letter

← Animation  
scale

Load FXZ file  
into node

Assign  
mouse  
handlers

FXD  
node

↓ Key letter,  
centered

```

        layoutY: bind Util.center(k,t,false);
        fill: Color.WHITE;
        content: letter;
        font: letterFont;
        textOrigin: TextOrigin.TOP;
    }
    ];
    scaleX: bind scale;
    scaleY: bind scale;
}
}
}

```

↑  
**Key letter,  
centered**

The Key class is used to display an old-fashioned-looking manual typewriter key on the screen. Its variables are as follows:

- diameter and fontSize set the size of the key and its key font.
- letter is the character to display on this key. In order to convert the ASCII characters A to Z into the values 0 to 25, the `String.toChar()` function is called, and 65 (the value of ASCII letter A) is subtracted.
- action is the event callback by which the outside world can learn when our key is pressed or released.
- scale is bound to the inherited variable pressed. It resizes our key whenever the mouse button is down.
- letterFont is the font we use for the key symbol.

The overridden `create()` function is, as always, where the scene graph is assembled. It starts with an unfamiliar bit of code, reproduced here:

```

def keyNode = FXDNode {
    url: "{__DIR__}key.fxz";
}

```

The `FXDNode` class creates a scene graph node from an `FXZ` file. This isn't actually as complex as sounds, given the SVG to JavaFX Graphics Converter tool has already done all the heavy lifting of converting the SVG format into declarative JavaFX Script code. The class also has options to load the `FXZ` in the background (rather than tie up the GUI thread) and provide a placeholder node while the file is loaded and processed. But the `FXDNode` created from our file doesn't have any event handlers.

```

keyNode.onMousePressed = function(ev:MouseEvent) {
    if(action!=null)
        action(letterValue,true);
};
keyNode.onMouseReleased = function(ev:MouseEvent) {
    if(action!=null)
        action(letterValue,false);
};

```

Once our key image has been loaded as a node, we need to assign two mouse event handlers to it. Because the object is already defined, we can't do this declaratively, so

we must revert to plain-old procedural code (à la Java) to hook up two anonymous functions. Both call the `action()` function type (if set) to inform the outside world a key has been pressed or released.

The rest of the scene graph code should be fairly clear. We need to overlay a letter onto the key, and that's what the `Text` node does. It uses the utility functions we created earlier to center itself inside the key. (There is actually a layout node called `Stack` that can center its contents; we saw it in listing 6.11.) At the foot of the scene graph the containing `Group` is bound to the `scale` variable, which in turn is bound to the inherited `pressed` state of the node. Whenever the mouse button goes down, the whole key shrinks slightly, as if being pressed.

**NOTE** *Don't forget to copy the FXZ files* The FXZ files for this project should be inside the directory representing the `jfxia.chapter9` package, so a reference to `__DIR__` can be used to find them. Once you've built the project code, make sure the FXZs are alongside the project class files.

Next we need to create the `Lamp` class to display our output.

## 9.2.6 The Lamp class: output from the machine

We've just developed a stylized input for our emulator; now we need a similar retro-looking output. In the original Enigma machine the encoded letters were displayed on 26 lamps, one of which would light up to display the output as a key was pressed, so that's what we'll develop next, in listing 9.4.

### Listing 9.4 Lamp.fx

```
package jfxia.chapter9;

import javafx.fxd.FXDLoader;
import javafx.fxd.FXDContent;
import javafx.scene.Node;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class Lamp extends CustomNode {
    package def diameter: Number = 40;
    package def fontSize: Integer = 20;

    public-init var letter: String;
    package var lit: Boolean = false on replace {
        if (lampOn != null) lampOn.visible = lit;
    }

    def letterFont: Font = Font.font(
        "Helvetica", FontWeight.REGULAR, fontSize
    );
}
```

**Manipulate  
lamp FXD**

```

var lampOn:Node;                                     ← Our FXD node

override function create() : Node {
    def lampContent:FXDContent = FXDLoader
        .loadContent("{__DIR__}lamp.fxz");           | Load as
                                                    | FXDContent type

    def lampNode = lampContent.getRoot();             ← Top level

    lampOn = lampContent.getNode("lampOn");           ← Specific
                                                    | layer

    var c:Node;
    var t:Node;
    Group {
        content: [
            c = lampNode ,                             ← Use in scene
            t = Text {                                  graph
                content: letter;
                font: letterFont;
                textOrigin: TextOrigin.TOP;
                layoutX: bind Util.center(c,t,true);
                layoutY: bind Util.center(c,t,false);
            }
        ]
    }
}

```

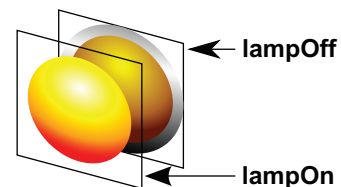
Letter text,  
centered

The Lamp class is a very simple binary display node. It has no mouse or keyboard input, just two states: lit or unlit. Once again we load an FXZ file and bring its content into our code. But this time, instead of using the quick-'n'-easy FXDNode class, we go the *long route* via FXDLoader and FXDContent. (There's no great advantage to this; I just thought a bit of variety in the example code wouldn't hurt!) Once the file is loaded, we extract references to specific parts of the scene graph described by the FXD.

If you load up the original SVG files, you'll find inside the lamp a layer with an alternative version of the center part of the image, making it appear lit up (see figure 9.6). The layer is set to be invisible, so it doesn't show by default. It has been given the ID `jfx:lampOn`, causing the converter to record a reference to it in the FXD.

In the code we extract that reference by calling `FXDContent.getNode()` with an ID of `lampOn`. We don't need the `jfx:` prefix with the FXZ/FXD; it was used in the original SVG only to tag the parts of the image we wanted to ID in the FXD. If the ID is found, we'll get back a scene graph node, which we'll store in a variable called `lampOn` (it doesn't *have* to share the same name). We can now manipulate `lampOn` in the code, by switching its visibility on or off whenever the status of `lit` changes.

This ability to design images in third-party applications, bring them into our JavaFX programs, and then reach inside and manipulate their constituent



**Figure 9.6** The lamp image is constructed from two layers. The lower layer shows the rim of the lamp and its dormant (off) graphic; the upper layer, invisible by default, shows the active (on) graphic.

parts is very powerful. Imagine, for example, a chess game where the board and all the playing pieces are stored in one big SVG (or Photoshop or Illustrator) image, tagged with individual JFX IDs. To update the game's graphics the designer merely supplies a replacement FXZ file. Providing the IDs are the same, it should drop straight into the project as a direct replacement, without the need for a rebuild of the code.

### 9.2.7 The Enigma class: binding the encryption engine to the interface

Ignoring the utility class, we've seen three application classes so far: the Rotor, which provides the basis of our Enigma cipher emulation; the Key, which provides the input; and the Lamp, which displays the output. Now it's time to pull these classes together with an actual application class, the first part of which is listing 9.5.

**Listing 9.5 Enigma.fx (version 1, part 1)**

```
package jfxia.chapter9;

import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.Tile;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

def rotors:Rotor[] = [
  Rotor { wiring: "JGDQOXUSCAMIFRVTPNEWKBLZYH";
    turnover: "R" } ,
  Rotor { wiring: "NTZPSFBOKMWRCJDIVLAEYUXHGQ";
    turnover: "F" } ,
  Rotor { wiring: "JVIUBHTCDYAKEQZPOSGXNRMWFL";
    turnover: "W" }
];

def reflector =
  Rotor { wiring: "QYHOGNECVPUZTFDJAXWMKISRBL"; }

def row1:String[] = [ "Q", "W", "E", "R", "T", "Z", "U", "I", "O" ];
def row2:String[] = [ "A", "S", "D", "F", "G", "H", "J", "K" ];
def row3:String[] = [ "P", "Y", "X", "C", "V", "B", "N", "M", "L" ];

def dummyLamp = Lamp {};
var lamps:Lamp[] = for(i in [0..<26]) dummyLamp;

def innerWidth:Integer = 450;
def innerHeight:Integer = 320;
// Part 2 is listing 9.6; part 3, listing 9.7
```

**Declare the  
rotors and  
reflector**

**Key and  
lamp layout**

This is just the top part of our application class. The code mainly concerns itself with setting up the three rotors and single reflector, plus defining the keyboard and lamp layout. We'll use the Enigma's authentic keyboard layout, which is similar to but not quite the same as QWERTY. Because we need to manipulate the lamps, we create a sequence to hold the nodes alphabetically. We won't be creating the lamp node in ABC order, so 26 dummy entries pad the sequence to allow `lamps[letter]=node` later on.

Listing 9.6 is the second part of our project and shows the main scene graph code that builds the window and its contents.

### Listing 9.6 Enigma.fx (version 1, part 2)

```
// Part 1 is listing 9.5
Stage {
  scene: Scene {
    var l:Node;
    content: Group {
      content: [
        l = VBox {
          layoutX: bind
            Util.center(innerWidth,l,true);
          layoutY: 20;
          spacing: 4;
          content: [
            createRow(row1,false,createLamp) ,
            createRow(row2,true,createLamp) ,
            createRow(row3,false,createLamp)
          ];
        } ,
        VBox {
          layoutX: bind l.layoutX;
          layoutY: bind l.boundsInParent.maxY+10;
          spacing: 4;
          content: [
            createRow(row1,false,createKey) ,
            createRow(row2,true,createKey) ,
            createRow(row3,false,createKey)
          ];
          effect: DropShadow {
            offsetX: 0;  offsetY: 5;
          };
        }
      ];
    };
    width: innerWidth; height: innerHeight;
  };
  title: "Enigma";
  resizable: false;
  onClose: function() { FX.exit(); }
}

// Part 3 is listing 9.7
```

**Lamp, centered**

**Three rows of lamps**

**Keys, positioned using lamps**

**Three rows of keys**

**Drop-shadow effect**

**Window inner content**

The structure is fairly simple: we have two VBox containers (they stack their contents in a vertical alignment, recall), each populated with three calls to `createRow()`. The top VBox is aligned centrally using the utility functions we developed earlier and 20 pixels from the top of the window's inner bounds. The second VBox has its X layout bound to its sibling and its Y layout set to 10 pixels below the bottom of its sibling.

The second VBox has an effect attached to it, `javafx.scene.effect.DropShadow`. We touched on effects briefly when we used a reflection in our video player project. Effects manipulate the look of a node, anything from a blur or a color tint to a reflection or even

a pseudo 3D distortion. The DropShadow effect merely adds a shadow underneath the node tree it is connected to. This gives our keys a floating 3D effect.

The `createRow()` function manufactures a row of nodes from a sequence of letters, using a function passed into it (`createLamp` or `createKey`). Its code begins the final part of this source file, as shown in listing 9.7.

### Listing 9.7 Enigma.fx (version 1, part 3)

```
// Part 1 is listing 9.5; part 2, listing 9.6
function createRow(row:String[] , indent:Boolean ,
  func:function(l:String):Node) : Tile {
  def h = Tile {
    hgap: 4;
    columns: sizeof row;
    content: for(l in [row]) { func(l); };
  };
  if(indent) {
    h.translateX =
      h.content[0].layoutBounds.width/2;
  }
  return h;
}
function createKey(l:String) : Node {
  Key {
    letter: l;
    action: handleKeyPress;
  };
}
function createLamp(l:String) : Node {
  def i:Integer = l.charAt(0)-65;
  def lamp = Lamp { letter: l; };
  lamps[i]=lamp;
  return lamp;
}
function handleKeyPress(l:Integer,down:Boolean) : Void {
  def res = encodePosition(l);
  lamps[res].lit = down;
  if(not down) {
    var b:Boolean;
    b=rotors[2].nextPosition();
    if(b) { b=rotors[1].nextPosition(); }
    if(b) { rotors[0].nextPosition(); }
  }
}
function encodePosition(i:Integer) : Integer {
  var res=i;
  for(r in rotors) { res=r.encode(res,false); }
  res=reflector.encode(res,false);
  for(r in reverse rotors) { res=r.encode(res,true); }
  return res;
}
```

← Create rows of lamps or keys

Intent row, if required

← Function used to create key

← Function used to create lamp

← Handle key up or down

Key release? Turn rotors

← Encode key position

Forward then back through rotors



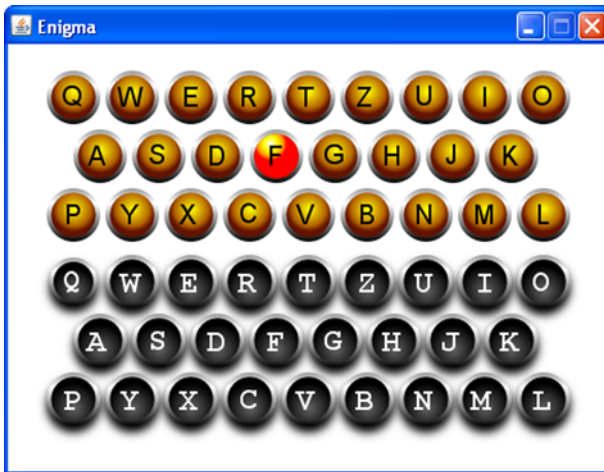
Here we see the `createRow()` function encountered in the previous part. Since the layout for both the keyboard and lamp board is the same, a single utility function is employed to position the nodes in each row. The `createKey()` and `createLamp()` functions are passed to `createRow()` to manufacture the nodes to be laid out. For that important authentic keyboard look, rows can be indented slightly, using the `indent` boolean.

Finally, we need to consider the `handleKeyPress()` and `encodePosition()` functions. The former uses the latter to walk forward, then backward across the rotors, with the deflector in the middle, performing each stage of the encryption. Once the input letter is encoded, it switches the associated lamp on or off (depending on the direction the key just moved) and turns any rotors that need turning (the `nextPosition()` function turns a rotor and returns `true` if it hits its *turnover* position).

And so, with the Enigma class itself now ready, we have version 1 of our simulated encryption machine.

### 9.2.8 Running version 1

What we have with version 1 is a functional, if not very practical, Enigma emulator. Figure 9.7 shows what to expect when the application is fired up.



**Figure 9.7** This is what we should see when compiling and running version 1. It works, but not in a very practical way. The stylized button and lamp nodes help lend the application an authentic feel.

Pressing a key runs its position through the rotors and reflector, with the result displayed on a lamp. There are two clear problems with the current version: first, we cannot see or adjust the settings of the rotors, and second (in a departure from the real Enigma), we cannot capture the output in a more permanent form.

In the second version of our Enigma we'll be fixing those problems, making the application good enough to put on the web for all to see.

### 9.2.9 Shortcuts using NetBeans, Photoshop, or Illustrator

The details in this chapter attempt to be as inclusive as possible. Inkscape was chosen as an example of an SVG editor precisely because it was free and did not favor a partic-

ular operating system. As mentioned previously, the JavaFX Production Suite does hold some benefits for users of certain well-known products. In this section we'll list those benefits.

If you are lucky enough to own Adobe Photoshop CS3 or Adobe Illustrator CS3, or you're working with someone who uses them, you'll be happy to know that the JavaFX Production Suite comes with plug-ins for these applications to save directly to FXZ (no need for external tools, like the SVG Converter). Obviously you need to install the Production Suite on the computer running Photoshop or Illustrator, and make sure there's an up-to-date JRE on there too, but you *don't* need to install the full JavaFX SDK.

Just as we saw with our SVG files, layers and sublayers in graphics created with these products can be prefixed with `jfx:` to preserve them in the FXD definition written into the FXZ file. Fonts, bitmaps, and other supporting data will also be included in the FXZ.

The good news isn't confined to designers; programmers using NetBeans also have an extra little tool included in the Production Suite. The *UI Stub Generator* is a convenience tool for NetBeans users to create JavaFX Script wrappers from FXZ files. Pointing the tool at any FXZ file results in a source code file being generated that extends `javafx.fxml.FXMLNode`. Each exposed node in the FXZ (the ones with `jfx:` prefixes in the original image) is given a variable in the class, so it can be accessed easily. An `update()` method is written to do all the heavy lifting of populating the variables with successive calls to `getNode()`. Once the file is generated, NetBeans users can compile this class into their project and use it in preference to explicit `getNode()` calls.

The UI Stub Generator helps to keep your source files clean from the mechanics of reading FXD data; however, once the source code has been generated you may feel the need to edit it. The tool assumes everything is a `Node`, so if you plan on addressing a given part of the scene graph by its true form (for example, a `Text` node), you'll need to change the variable's type and add a cast to the relevant `getNode()` line.

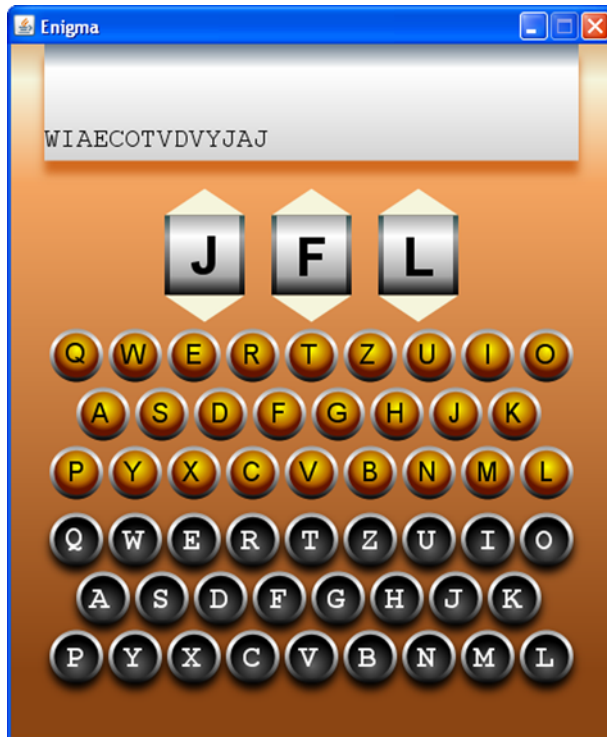
All of these tools are fully documented in the help files that come bundled with the JavaFX Production Suite.

### 9.3 **More cryptic: Enigma machine, version 2**

Version 1 of the project has a couple of issues that need addressing. First, the user needs to be able to view and change the state of the rotors. Second, we need to introduce some way to capture the encoded output.

To fix the first problem we're going to turn the `Rotor` class into a node that can be used not only to encode a message but also to set the encryption parameters. To fix the second we'll develop a simple printout display, which looks like a teletype page, that records our output along with the lamps. You can see what it looks like at the top of figure 9.8.

As a glance at figure 9.8 reveals, each rotor will show its letter in a gradient-filled display, with arrow buttons constructed from scene graph polygons. The paper node we'll develop will be shaded at the top, giving the impression its text is vanishing over a curved surface. Changes to the `Enigma` class will tie the features into the application's interface, giving a more polished look.



**Figure 9.8** Quite an improvement: the Enigma emulator acquires a printout display and rotors, as well as an attractive shaded backdrop.

### 9.3.1 The Rotor class, version 2: giving the cipher a visual presence

The Rotor class needs a makeover to turn it into a fully fledged custom node, allowing users to interact with it. Because this code is being integrated into the existing class, I've taken the liberty of snipping (omitting for the sake of brevity) the bulk of the code from the previous version, as you'll see from listing 9.8. (It may not save many trees, but it could help save a few branches.) The snipped parts have been marked with bold comments to show what has been dropped and where. Refer to listing 9.1 for a reminder of what's missing.

#### Listing 9.8 Rotor.fx (version 2—changes only)

```
package jfxia.chapter9;

import javafx.scene.Node;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
```

```

import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class Rotor extends CustomNode {
    // [Snipped variables: see previous version]

    def fontSize:Integer = 40;
    def width:Number = 60;
    def height:Number = 60;
    def buttonHeight:Number = 20.0;
    def letterFont:Font = Font.font(
        "Helvetica" , FontWeight.BOLD, fontSize
    );

    // [Snipped init block: see previous version]

    override function create() : Node {
        var r:Node;
        var t:Node;
        Group {
            content: [
                Polygon {
                    points: [
                        width/2 , 0.0 ,
                        0.0 , buttonHeight ,
                        width , buttonHeight
                    ];
                    fill: Color.BEIGE;
                    onMouseClicked: function(ev:MouseEvent) {
                        rotorPosition = if(rotorPosition==0) 25
                        else rotorPosition-1;
                    }
                } ,
                r = Rectangle {
                    layoutY: buttonHeight;
                    width: width;
                    height: height;
                    fill: LinearGradient {
                        proportional: false;
                        endX:0; endY:height;
                        stops: [
                            Stop { offset: 0.0;
                                color: Color.DARKSLATEGRAY; } ,
                            Stop { offset: 0.25;
                                color: Color.DARKGRAY; } ,
                            Stop { offset: 1.0;
                                color: Color.BLACK; }
                        ]
                    }
                } ,
                Rectangle {
                    layoutX: 4;
                    layoutY: buttonHeight;
                    width: width-8;
                    height: height;
                    fill: LinearGradient {

```

**Sizing and font constants**

**Up arrow polygon**

**Move rotor back one**

**Rotor display rim**

**Rotor display body**

```

    proportional: false;
    endX:0; endY:height;
    stops: [
        Stop { offset: 0.0;
              color: Color.DARKGRAY; } ,
        Stop { offset: 0.25;
              color: Color.WHITE; } ,
        Stop { offset: 0.75;
              color: Color.DARKGRAY; } ,
        Stop { offset: 1.0;
              color: Color.BLACK; }
    ]
}
},
t = Text {
    layoutX: bind Util.center(r,t,true);
    layoutY: bind buttonHeight +
              Util.center(r,t,false);
    content: bind posToChr(rotorPosition);
    font: letterFont;
    textOrigin: TextOrigin.TOP;
},
Polygon {
    layoutY: buttonHeight+height;
    points: [
        0.0 , 0.0 ,
        width/2 , buttonHeight ,
        width , 0.0
    ];
    fill: Color.BEIGE;
    onMouseClicked: function(ev:MouseEvent) {
        rotorPosition = if(rotorPosition==25) 0
                        else rotorPosition+1;
    }
}
];
}
}

// [Snipped encode(), nextPosition(): see previous version]
}
// [Snipped posToChr(), chrToPos(): see previous version]

```

Rotor current letter

Down arrow polygon

Move rotor forward one

The new Rotor class contains a hefty set of additions. The scene graph, as assembled by `create()`, brings together several layered elements to form the final image. The graph is structured around a `Group`, at the top and tail of which are `Polygon` shapes. As its name suggests, the `Polygon` is a node that allows its silhouette to be custom defined from a sequence of coordinate pairs. The values in the `points` sequence are alternate x and y positions. They form the outline of a shape, with the last coordinate connecting back to the first to seal the perimeter. Both of our polygons are simple triangles, with points at the extreme left, extreme right, and in the middle, forming up and down arrows.

The rest of the scene graph consists of familiar components: two gradient-filled rectangles and a text node, forming the rotor body between the arrows.

The arrow polygons have mouse handlers attached to them to change the current `rotorPosition`. As rotor positions form a circle, condition logic wraps `rotorPosition` around when it overshoots the start or end of its range. The turnover position is ignored, you'll note, because we don't want rotors *clocking* each other as we're manually adjusting them.

So that's the finished Rotor. Now for the Paper class.

### 9.3.2 The Paper class: making a permanent output record

The Paper class is a neat little display node, with five lines of text that are scaled vertically to create the effect of the lines vanishing over a curved surface. We're employing a fixed-width font, for that authentic manual typewriter look. Check out figure 9.9. You can almost hear the clack-clack-clack of those levers, hammering out each letter as the keys are pressed.



**Figure 9.9** Each line of our Paper node is scaled to create the optical effect of a surface curving away, to accompany the shading of the background Rectangle.

The real Enigma didn't have a paper output. The machines were designed to be carried at the frontline of a battle, and their rotor mechanics and battery were bulky enough without adding a stationery cupboard full of paper and ink ribbons. But we've moved on a little in the 80-plus years since the Enigma was first developed, and while a 1200 dpi laser printer might be stretching credibility a little *too* far, we can at least give our emulator a period teletype display. The code is in listing 9.9.

#### Listing 9.9 Paper.fx

```
package jfxia.chapter9;

import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.effect.DropShadow;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
```

```

import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class Paper extends CustomNode {
    package var width:Number = 300.0;
    public-read var height:Number;

    def lines:Integer= 5;
    def font:Font = Font.font(
        "Courier" , FontWeight.REGULAR , 20
    );

    def paper:Text[] = for(i in [0..

Sequence of text lines



Scale and position to create curve



Shaded paper backdrop



Text nodes expanded in place



Scroll up when clicked



```

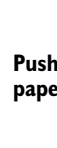
```


    }
}

package function add(l:String) : Void {
    def z:Integer = lines-1;
    if(l.equals("\n")) {
        var i:Integer = 1;
        while(i<lines) {
            paper[i-1].content = paper[i].content;
            i++;
        }
        paper[z].content="";
    }
    else {
        paper[z].content = "{paper[z].content}{l}";
    }
}
}

```

 **Add to bottom text line**

 **Push paper up**

 **Append to last line**

The most interesting thing about listing 9.9 is the creation of the sequence paper. What this code does is to stack several Text nodes using layoutY, scaling each in turn from a restricted height at the top to full height at the bottom. The rather fearsome-looking code `1.0-((lines-1-i)*0.15)` subtracts multiples of 15 percent from the height of each line, so the first line is scaled the most and the last line not at all. The result is a curved look to the printout, just what we wanted!

The rest of the listing is unremarkable scene graph code, with the exception of the `add()` function at the end. This is what the outside world uses to push new letters onto the bottom line of the teletype display. Normally the character is appended to the end of the last Text node, but if the character is a carriage return, each Text node in content is copied to its previous sibling, creating the effect of the paper scrolling up a line. The last Text node is set to an empty string, ready for fresh output.

We now have all the classes we need for our finished Enigma machine; all that's left is to refine the application class itself to include our new graphical rotors and paper.

### 9.3.3 The Enigma class, version 2: at last our code is ready to encode

Having upgraded the Rotor class and introduced a new Paper class, we need to integrate these into the display. Listing 9.10 does just that.

#### Listing 9.10 Enigma.fx (version 2, part 1 – changes only)

```

package jfxia.chapter9;

import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.Tile;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;

```



```

import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

// [Snipped definitions for rotors, reflector, row1, row2,
//   row3 and lamps: see previous version]

def innerWidth:Integer = 450;
def innerHeight:Integer = 520;

var paper:Paper;
// Part 2 is listing 9.11; part 3, listing 9.12

```

Listing 9.10 is the top part of the updated application class, adding little to what was already there in the previous version. That's why I've snipped parts of the content again, indicated (as before) with comments in bold. You can refresh your memory by glancing back at listing 9.5. As you can see, the window has been made bigger to accommodate the new elements we're about to add, and we've created a variable to hold one of them, paper.

The main changes come in the next part, listing 9.11, the scene graph.

#### Listing 9.11 Enigma.fx (version 2, part 2)

```

// Part 1 is listing 9.10
Stage {
  scene: Scene {
    content: Group {
      var p:Node;
      var r:Node;
      var l:Node;
      content: [
        p = paper = Paper {
          width: innerWidth-50;
          layoutX: 25;
        } ,
        r = Tile {
          columns: 3;
          layoutX: bind Util.center(innerWidth,r,true);
          layoutY: bind p.boundsInParent.maxY;
          hgap: 20;
          content: [
            rotors[0], rotors[1], rotors[2]
          ];
        } ,
        l = VBox {
          layoutX: bind Util.center(innerWidth,l,true);
          layoutY: bind r.boundsInParent.maxY+10;
          spacing: 4;
          content: [
            createRow(row1,false,createLamp) ,
            createRow(row2,true,createLamp) ,
            createRow(row3,false,createLamp)
          ];
        } ,
        VBox {

```

Our new  
Paper class

Rotors in  
horizontal layout

```

        layoutX: bind l.layoutX;
        layoutY: bind l.boundsInParent.maxY+10;
        spacing: 4;
        content: [
            createRow(row1,false,createKey) ,
            createRow(row2,true,createKey) ,
            createRow(row3,false,createKey)
        ];
        effect: DropShadow {
            offsetX: 0;  offsetY: 5;
        };
    }
};

fill: LinearGradient {
    proportional: true;  endX: 0;  endY: 1;
    stops: [
        Stop { offset: 0.0;  color:  Color.BURLYWOOD; } ,
        Stop { offset: 0.05; color:  Color.BEIGE; } ,
        Stop { offset: 0.2;  color:  Color.SANDYBROWN; } ,
        Stop { offset: 0.9;  color:  Color.SADDLEBROWN; }
    ];
    width: innerWidth; height: innerHeight;
}
title: "Enigma";
resizable: false;
onClose: function() { FX.exit(); }
}
// Part 3 is listing 9.12

```

Gradient backdrop  
to window

Listing 9.11 is the meat of the new changes. I've preserved the full listing this time, without any snips, to better demonstrate how the changes integrate into what's already there. At the top of the scene graph we add our new Paper class, sized to be 50 pixels smaller than the window width. Directly below that we create a horizontal row of Rotor objects; recall that the new Rotor is now a CustomNode and can be used directly in the scene graph. The lamps have now been pushed down to be 10 pixels below the rotors. At the very bottom of the Scene we install a LinearGradient fill to create a pleasing shaded background for the window contents.

Just one more change to this class is left, and that's in the (on screen) keyboard handler, as shown in listing 9.12.

#### Listing 9.12 Enigma.fx (version 2, part 3 – changes only)

```

// Part 1 is listing 9.10; part 2 is listing 9.11
// [Snipped createRow(), createKey(),
//  createLamp(): see previous version]

function handleKeyPress(l:Integer,down:Boolean) : Void {
    def res = encodePosition(l);
    lamps[res].lit = down;
    if(not down) {

```

```

    var b:Boolean;
    b=rotors[2].nextPosition();
    if(b) { b=rotors[1].nextPosition(); }
    if(b) { rotors[0].nextPosition(); }
  }
  else {
    paper.add(Rotor.posToChr(res));
  }
}

// [Snipped encodePosition(): see previous version]

```

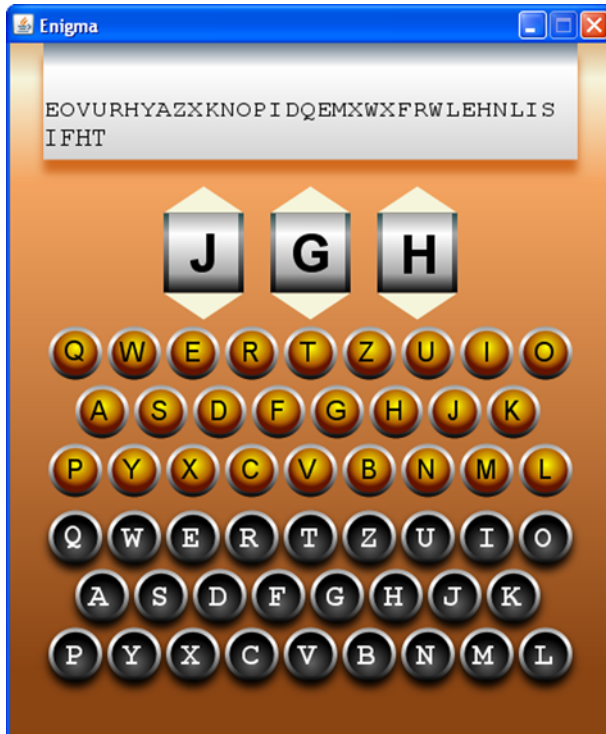
**Key down?  
Add to paper**

At last we have the final part of the Enigma class, and once again the unchanged parts have been snipped. Check out listing 9.7 if you want to refresh your memory. Only one change to mention, but it's an important one: the `handleKeyPress()` now has some plumbing to feed its key into the new paper node. This is what makes the encoded letter appear on the printout when a key is clicked.

### 9.3.4 Running version 2

Running the Enigma class gives us the display shown in figure 9.10. Our keys and lamps have been joined by a printout/teletype affair at the head of the window and three rotors just below.

Clicking the arrows surrounding the rotors causes them to change position, while clicking the paper display causes it to jump up a line. As we stab away at the keys, our



**Figure 9.10** Our Enigma machine in action, ready to keep all our most intimate secrets safe from prying eyes (providing they don't have access to any computing hardware made after 1940).

input is encoded through the rotors, flashed up on the lamps, and appended to the printout.

Is our Enigma machine finished then? Well, for now, yes, but there's plenty of room for new features. The Enigma we developed is a simplified version of the original, with two important missing elements. The genuine Enigma had rotors that could be removed and interchanged, to further hinder code breaking. It also featured a *plug board*: 26 sockets linked by 13 pluggable cables, effectively creating a configurable second reflector. That said, unless anyone reading this book is planning on invading a small country, the simplified emulator we've developed should be more than enough.

In the final part of this chapter we'll turn our new application into an applet, so we can allow the world and his dog to start sending us secret messages.

## 9.4 From application to applet

One of the banes of deploying Java code in the *bad old days* was myriad options and misconfigurations one might encounter. Traditionally Java relied on the end user (the customer) to keep the installed Java implementation up to date, and naturally this led to a vast range of runtime versions being encountered in the wild. The larger download size of the Java runtime, compared to the Adobe Flash Player, also put off many web developers. Java fans might note this is an apples-and-oranges comparison; the JRE is more akin to Microsoft's .NET Framework, and .NET is many times larger than the JRE. Yet still the perception of Java as large and slow persisted.

Starting with Java 6 update 10 at the end of 2008, a huge amount of effort was put behind trying to smarten up the whole Java deployment experience, for both the end user and the developer. In the final part of this chapter we'll be exploring some of these new features, through JavaFX. Although none of them are specifically JavaFX changes, they go hand in glove with the effort to brighten up Java's prospects on the desktop, something that JavaFX *is* a key part of.

### 9.4.1 The Enigma class: from application to applet

Having successfully created an Enigma application, we need to port it over to be an applet. If you've ever written any Java applets, this might send a chill down your spine. Although Java applications and applets have a lot in common, translating one to the other still demands a modest amount of reworking. But this is JavaFX, not Java, and our expectations are different. JavaFX was, after all, intended to ease the process of moving code between different types of user-facing environments.

Listing 9.13 shows the extent of the changes. Astute readers (those who actually read code comments) will note that once again the unchanged parts of the listing have been omitted, showing only the new material and its context.

#### Listing 9.13 Enigma.fx (version 3 – changes only)

```
package jfxia.chapter9;
import javafx.scene.Group;
```

```

import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.Tile;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Rectangle;
import javafx.stage.AppletStageExtension;
import javafx.stage.Stage;

// [Snipped definitions for rotors, reflector, row1, row2, row3,
//   lamps, innerWidth, innerHeight and paper: see previous version]

Stage {
    // [Snipped Scene, title, resizable and
    //   onClose: see previous version]

    extensions: [
        AppletStageExtension {
            shouldDragStart: function(e: MouseEvent): Boolean {
                return e.shiftDown and e.primaryButtonDown;
            }
            useDefaultClose: true;
        }
    ];
}

// [Snipped createRow(), createKey(), createLamp(),
//   handleKeyPress() and encodePosition(): see first version]

```

Trap mouse events

Import applet extension

Plug extension into Stage

Condition to begin drag action

Closing X button

Your eyes do not deceive you: the entire extent of the modifications amount to nothing more than eight lines of new code and two extra class imports (only one of which specifically relates to applets). The modifications center on an enhancement to the Stage object. To accommodate specific needs of individual platforms and environments, JavaFX supports an extension mechanism, using subclasses of `javafx.stage.StageExtension`. The specific extension we're concerned with is the `AppletStageExtension`, which adds functionality for using the JavaFX program from inside a web page.

In truth, we're not obliged to use this extension. Most JavaFX desktop programs can be run as applets without any specific modifications, but the extension allows us to specify applet-specific behavior, thus making our application that little bit more professional.

As you can see from the code, Stage has an `extensions` property that accepts a sequence of `StageExtension` objects. We've used an instance of `AppletStageExtension`, which does two things. First, it specifies when a mouse event should be considered the start of a drag action to rip the applet out of the web page and onto the desktop. Second, it adds a close box to the applet when on the desktop. But simply changing our application isn't enough; for effective deployment on the web we need to package it into a JAR file. And that's just what we'll do next.

### 9.4.2 The JavaFX Packager utility

The JavaFX SDK comes with a neat little utility to help deploy our JavaFX programs. It can be used to package up our applications for all manner of different environments. For our project we want to use it to target the desktop platform and make the resulting program capable of running as an applet.

#### NetBeans users, take note

As always, I'm going to show you how the packaging process works under the hood, sans IDE. If you're using NetBeans, however, you may want to look over the following step-by-step tutorial for how to access the same functionality inside your IDE:

<http://javafx.com/docs/tutorials/deployment/>

The `javafxpackager` utility can be called from the command line and has a multitude of options. Table 9.1 is a list of what's available.

**Table 9.1** JavaFXPackager options

Option(s)	Function
-src or -sourcepath	The location of your JavaFX Script source code (mandatory).
-cp or -classpath or -librarypath	The location of dependencies, extra JAR files your code relies on.
-res or -resourcepath	The location of resources, extra data, and image files your code relies on.
-d or -destination	The directory to write the output.
-workDir	The temporary directory to use when building the output.
-v or -verbose	Print useful (debugging) information during the packaging process.
-p or -profile	Which environment to target. Either <code>DESKTOP</code> or <code>MOBILE</code> (desktop includes applets).
-appName and -appVendor and -appVersion	Meta information about the application: its name, creator's name, and version.
-appClass	The startup class name, including package prefix (mandatory).
-appWidth and -appHeight	Application width and height (particularly useful for applets).
-appCodebase	Codebase of where the application is hosted, if available on the web. This is the base address of where the JNLP, JAR, and other files are located.

**Table 9.1** JavaFXPackager options (continued)

Option(s)	Function
-draggable	Make applet draggable from the browser and onto the desktop.
-sign and -keystore and -keystorePassword and -keyalias and -keyaliasPassword	Used to sign an applet to grant it extra permissions.
-pack200	Use Java-specific compression (usually much tighter than plain JAR).
-help and -version	Useful information about packager utility.

Now that you've familiarized yourself with the options, it's time to see the packager in action.

### 9.4.3 Packaging up the applet

Before we can run the packager for ourselves, we need to make one change. In previous versions of this project we simply copied the FXZ files into the package directory created by our compiler. For the `javafxpackager` to work, we need to put the files in their own resource directory. (Of course, if you're using an IDE you probably already have set up this directory, so your IDE could copy the necessary resources alongside the code during the build process.)

- 1 Next to the `src` directory that holds the project source code, create a new directory called `res`.
- 2 Inside the new `res` directory create one called `jfxia` and then one inside that called `chapter9`. You should end up with a directory structure of `res/jfxia/chapter9` (or `res\jfxia\chapter9` if you prefer DOS backslashes) inside the project directory.
- 3 Copy the two FXZ files we're using for our project into the `chapter9` directory.

Now we're ready to run the packager. Open a new command console (such as an MS-DOS console on Windows) and change into the project directory, such that the `src` and `res` directories live off your current directory. Assuming the JavaFX tools are on your current command path, type (all on one line):

```
javafxpackager -src .\src -res .\res
  -appClass "jfxia.chapter9.Enigma"
  -appWidth 450 -appHeight 520
  -draggable -pack200 -sign
```

Alternatively, if you're running a Unix-flavored machine, try this:

```
javafxpackager -src ./src -res ./res
  -appClass "jfxia.chapter9.Enigma"
  -appWidth 450 -appHeight 520
  -draggable -pack200 -sign
```

I've broken the command over several lines; you might want to turn it into a script (or batch file) if you plan to run the packager from outside an IDE often. Let's look at the options, chunk by chunk:

- `-src` is the location of our source code files. The packager attempts to build the source code, so it needs to know where it lives.
- `-res` is the location of any resource, which in our case is the `res` directory housing copies of our FXZ files.
- `-appClass` is the startup class for our application.
- `-appWidth` and `-appHeight` are the size of the applet.
- `-draggable` activates the ability to drag the applet out of the browser (but our code controls what key/mouse events will trigger this).
- `-pack200` uses supertight compression on the resulting application archive.
- `-sign` signs the output so we can ask to be granted extra permissions when it runs on the end user's machine. Even though our applet doesn't do anything dangerous, we may trigger a security problem when running it directly from the computer's file system (using a file: URL location). Since we don't provide any keystore details, the packager will create a short-term self-signed certificate for us, which will be fine for testing purposes.
- `-cp` (classpath) informs the packager about any extra JAR dependencies, so they can be included in the build process and copied into the distribution output. We don't need any extra JARs, so I've not used this option (although remember, if you're building code under JavaFX 1.0, you may need to reference the Production Suite's `javafx-fxd-1.0.jar` file, because it wasn't shipped as standard).

If all went well, you should end up with a new directory, called `dist`. This name is the default when we don't specify a `-destination` option. Inside, we have all the files we need to run the Enigma applet.

- `Enigma.jar` and `Enigma.jar.pack.gz`, which contain our project's classes and resources in both plain JAR and Pack200 format.
- `Enigma.jnlp` and `Enigma_browser.jnlp`, which hold the Java Web Start (JWS) details for our Enigma machine in regular desktop and applet variants.
- `Enigma.html`, an HTML file we can use to launch our applet. We can copy the core markup from this file into any web page hosting the applet.
- If you package a project using the `-cp` option to list dependent JARs, a `lib` directory will be created to contain copies of them.

From the desktop, enter the `dist` directory and double-click (or otherwise open) the HTML file. Your favorite web browser should start, and eventually the Enigma applet should appear in all its glory. It may take some time initially, as the JavaFX libraries are not bundled in the distribution the package creates but loaded over the web from the `javafx.com` site. This is to ensure maximum efficiency—why should



### Java Web Start, JNLP, and applet security

JWS is a way of packaging and distributing Java desktop applications, deployable from a single click. Although usually started from a web link in the first instance, a JWS program can install desktop icon shortcuts and uninstall options in appropriate places for the host OS, resembling any other locally installed application. Cached copies of the application JARs may be held locally for speed, but the application is tied firmly to its origin web address, with updates fetched as required.

The JNLP file format (Java Network Launch(ing) Protocol) is at the heart of JWS. It provides metadata about an application the JRE needs to install and manage it. In Java 6 update 10 the reach of JNLP was extended to include applets, as this web page explains:

<http://java.sun.com/developer/technicalArticles/javase/newapplets/>

JavaFX doesn't change the underlying security model of Java, either for applets or JWS applications. Unsigned applets (those not requiring user permission to run) are typically restricted in their network access and ability to interact with local computer resources. Signed applets can use the JNLP file to request extra privileges, which the user can grant or deny. Expert end users can also edit a special policy file to automatically grant or deny permissions to Java programs, based on their origin.

For an overview of JNLP and its security permissions, plus a quick tutorial on applet security, see the following web pages (split over two lines):

<http://java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/syntax.html>

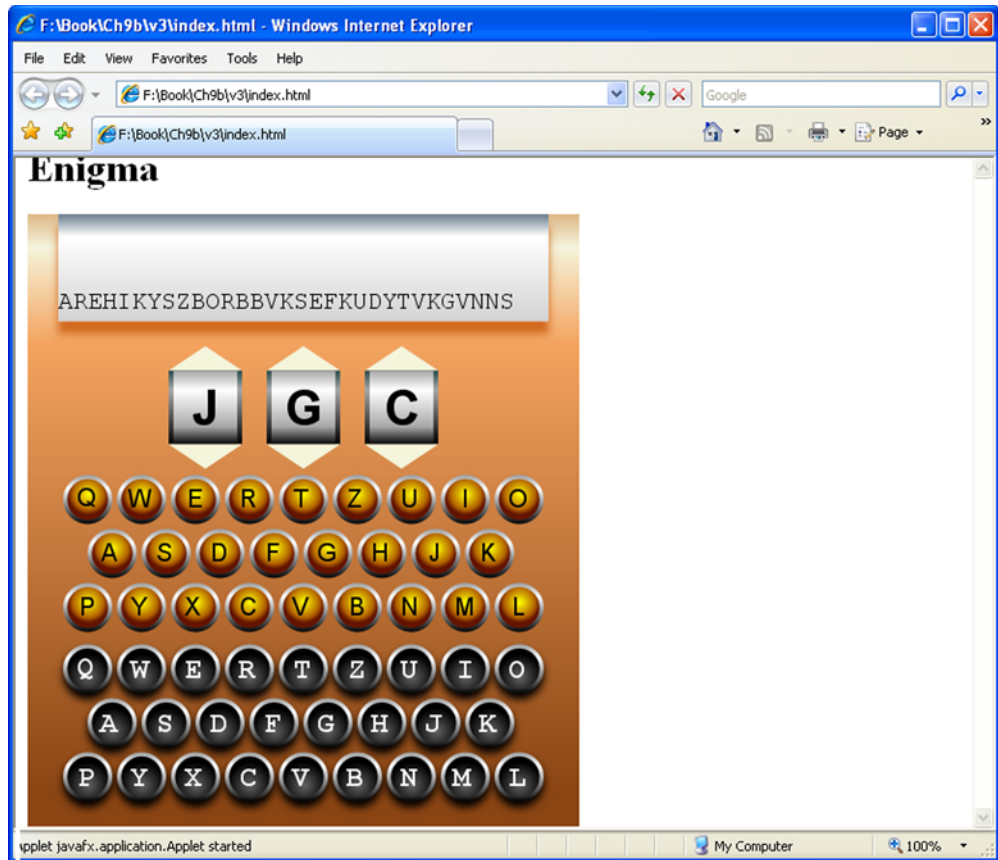
<http://java.sun.com/docs/books/tutorial/security/tour1/>

every individual JFX application burden itself with local copies of the core API libraries, when they could be loaded (and maybe cached) from a single location across all JavaFX programs?

The `javafx.com` site holds independent copies of the JARs for each JavaFX release, and the `javafxpackager` utility writes files that target its own release, so code packaged under JavaFX 1.2 will continue to run once 1.3 is available. This enables the JavaFX team to make breaking changes to the APIs without upsetting currently deployed code.

Figure 9.11 shows the applet in action.

The Enigma machine looks just as it did when we ran it directly on the desktop, except now it's inside a browser window. Before we drag it back onto the desktop, let's look at the JNLP file and make a small modification.



**Figure 9.11** Our applet running inside Microsoft's Internet Explorer

#### 9.4.4 Dragging the applet onto the desktop

Dragging the application out of the browser gives us the opportunity to put an icon on the desktop. By default JavaFX uses a boring generic logo, but fortunately we can change that. To do so we need to create icon files and link them into the JNLP file.

If you download the source code from the book's website, you'll find, inside this project's directory, a couple of GIF files: `Enigma_32.gif` and `Enigma_64.gif`. We'll use these as our icons (unless you fancy creating your own). Java Web Start allows us to specify icons of different sizes, for use in different situations. For the record: the first of the supplied icons is 32x32 pixels in size, and the second is 64x64 pixels in size.

Copy these two images into the `dist` directory, and we're ready to change the JNLP. Take a look at listing 9.14.

#### Listing 9.14 `Enigma_browser.jnlp`

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="file:/JavaFX_in_Action/dist/"
```

```

href="Enigma_browser.jnlp">
<information>
  <title>Enigma</title>
  <vendor>Simon</vendor>
  <homepage href="" />
  <description>Enigma</description>
  <offline-allowed/>
  <shortcut>
    <desktop/>
  </shortcut>
  <icon href="Enigma_32.gif" width="32" height="32" />
  <icon href="Enigma_64.gif" width="64" height="64" />
</information>
<security>
  <all-permissions/>
</security>
<resources>
  <j2se version="1.5+" />
  <property name="jnlp.packEnabled" value="true"/>
  <property name="jnlp.versionEnabled" value="true"/>
  <extension name="JavaFX Runtime"
    href="http://dl.javafx.com/1.2/javafx-rt.jnlp"/>
  <jar href="Enigma.jar" main="true"/>
</resources>
<applet-desc name="Enigma"
  main-class="com.sun.javafx.runtime.adapter.Applet"
  width="450" height="520">
  <param name="MainJavaFXScript" value="jfxia.chapter9.Enigma">
</applet-desc>
</jnlp>

```

We have the `Enigma_browser.jnlp` file created by the packager for our application, complete with two extra lines (highlighted in bold) to hook up our icons. By studying the XML contents you can see how the details we passed into the packager were used to populate the JNLP configuration.

### More JNLP information

Java Web Start supports all manner of options for controlling how Java applications behave as applets and on the desktop. Some of them are exposed by options on the JavaFX Packager, while others may require some post-packaging edits to the JNLP files. Unfortunately an in-depth discussion of JNLP is beyond the scope of this chapter. The lengthy web address below (split over two lines) points to a page with details and examples of the various JNLP options.

[http://java.sun.com/javase/6/docs/technotes/  
guides/javaws/developersguide/syntax.html](http://java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/syntax.html)

So much for the icons, now it's time to try dragging our applet out of the browser and turning it back into a desktop application.

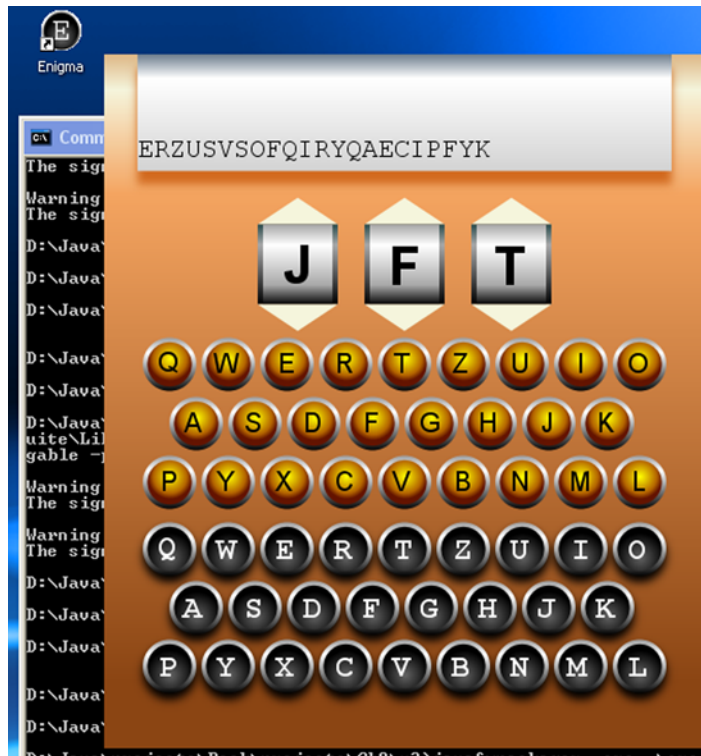
Double-click the Enigma.html file inside dist to start it up again in the browser. Now hold down the Shift key on your computer, click and hold inside the applet with your mouse, and drag away from the browser. The applet should become unstuck from the browser page and float over the desktop. Remember, we specified the criteria for the start of a drag operation in the `shouldDragStart` function of the `AppletStageExtension` we plugged into our application's `Stage`.

Once it's floating free, you can let the applet go, and it will remain on the desktop. Figure 9.12 shows how it might look. In place of the applet on the web page there should appear a default Java logo, showing where the applet once lived. In the top-right corner of our floating applet there should be a tiny close button, a result of us specifying `useDefaultClose` in the extension.

We now have two courses of action:

- Clicking the applet's close button while the applet's web page is still open returns the applet to its original home, inside the web page.
- Clicking the web page's close button while the applet is outside the browser (on the desktop) causes the browser window (or tab) to close and the applet to be installed onto our desktop, including a desktop icon.

Even though the applet's web page may have vanished, the applet continues to run without a hitch. It is now truly independent of the browser. This means getting the



**Figure 9.12** Starting life in a web page, our Enigma emulator was then dragged onto the desktop to become an application (note the desktop icon) and finally relaunched from the desktop.

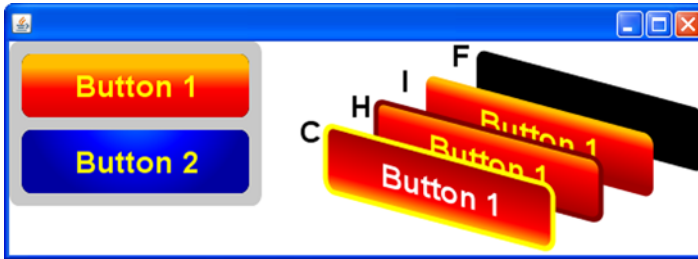
Enigma emulator onto the desktop is as simple as pulling it from the browser window and closing the page. The applet will automatically transform into a desktop application, complete with startup icon (although in reality it remains a JWS application, it merely has a new home).

Removing the application should be as simple as following the regular uninstall process for software on your operating system. For example, Windows users can use the Add or Remove Programs feature inside Control Panel. Enigma will be listed along with other installed applications, with a button to remove it.

At last we have a fully functional JavaFX applet.

## 9.5 Bonus: Building the UI in an art tool

Before we get to the summing up, there's just enough time for another quick end-of-chapter detour. At the end of section 9.2.6, when we coded the Enigma's lamps, I mentioned the possibility of building an entire UI inside a design/art tool. In this section we're going to do just that. Figure 9.13 shows a UI constructed in Inkscape. The left-hand side of the window contains two buttons, each constructed from four layers stacked atop each other. The right-hand side indicates how those layers are formed.



**Figure 9.13** Two buttons (left), each formed using four carefully labeled layers (demonstrated right), which are manipulated by JavaFX code to create functioning buttons

Each button layer has an ID consisting of a base ID for that button, followed by a letter denoting the layer's function: either F (footprint), I (idle), H (hover), or C (clicked). The *idle* layer is shown when the button is idle (not hovered over or pressed), the *hover* layer is shown when the mouse is inside the button, and the *clicked* layer is shown when the button is pressed. Only one of these three layers is visible at any one time. The final layer, *footprint*, is used as a target for event handlers and to define the shape of the button; it is never actually displayed.

Listing 9.15 is sample code that loads the FXD created from the Inkscape file and manipulates the two buttons inside it. The first button has a base ID of `button1`, and the second has a base ID of `button2`.

### Listing 9.15 UI.fx

```
import javafx.fxd.FXDNode;
import javafx.scene.Node;
import javafx.scene.input.MouseEvent;
import javafx.scene.Scene;
```

```
import javafx.stage.Stage;

def ui:FXDNode = FXDNode { url: "{__DIR__}ui.fxz" };
FXDButton {
  fxd: ui;
  id: "button1";
  action: function() { println("Button 1 clicked"); }
}
FXDButton {
  fxd: ui;
  id: "button2";
  action: function() { println("Button 2 clicked"); }
}

Stage {
  scene: Scene {
    content: ui;
  }
}

class FXDButton {
  var footprintNode:Node;
  var idleNode:Node;
  var hoverNode:Node;
  var clickNode:Node;

  public-init var fxd:FXDNode;
  public-init var id:String;
  public-init var action:function():Void;

  init {
    footprintNode = fxd.getNode("{id}F");
    idleNode = fxd.getNode("{id}I");
    hoverNode = fxd.getNode("{id}H");
    clickNode = fxd.getNode("{id}C");
    makeVisible(idleNode);

    footprintNode.onMouseEntered = function(ev:MouseEvent) {
      makeVisible(
        if(footprintNode.pressed) clickNode
        else hoverNode
      );
    }
    footprintNode.onMouseExited = function(ev:MouseEvent) {
      makeVisible(idleNode);
    }
    footprintNode.onMousePressed = function(ev:MouseEvent) {
      makeVisible(clickNode);
      if(action!=null) action();
    }
    footprintNode.onMouseReleased = function(ev:MouseEvent) {
      makeVisible(
        if(footprintNode.hover) hoverNode
        else idleNode
      );
    }
  }
}
```

Root scene graph  
node of FXD

```

function makeVisible(n:Node) : Void {
    for(i:Node in [idleNode,hoverNode,clickNode])
        i.visible = (i==n);
}

```

The `FXDButton` class is the hub of the action, turning parts of the FXD scene graph into a button. It accepts an `FXDNode` object, a base ID, and an action function to run when the button is clicked. During initialization it locates the four required layer nodes inside the FXD structure and adds the necessary event code to make them function as a button. For example, a base ID of `button1` extracts nodes with IDs of `button1F`, `button1I`, `button1H`, and `button1C` and then adds event handlers to the footprint node to control the visibility of the other three according to mouse events. (In the original Inkscape file the layers were labeled with IDs of `jfx:button1F`, `jfx:button1I`, etc.)

The beauty of this scheme is that a designer can create whole UI scene graphs inside Inkscape, Photoshop, or Illustrator, and then a programmer can turn them into functional user interfaces without having to recompile the code each time the design is tweaked. This brings us perilously close to the Holy Grail of desktop software programming: no more tedious and inflexible reconstructions of a designer's artwork using UI widgets and fiddly layout code; designers draw what they want, and programmers breathe life directly into their art.

I'm certainly not suggesting that in the future every UI will be constructed this way. This technique suits UIs resembling interactive photos or animations, an informal style previously used only in video games and children's software but now becoming increasingly trendy with other types of software too. I expect most UIs will become a fusion of the formal and informal—an audio production tool, for example, may have a main window looking just like a picture of a studio mixing desk, but *secondary* windows will still use familiar widgets (albeit styled).

As JavaFX (and its supporting tool set) continues to grow, we can expect more and more animation and transition capabilities to be shifted away from the source code and exposed directly to the designer. While highly sophisticated UIs will probably always need to be developed predominantly by a programmer, the ultimate goal of JFX is to allow simple (bold, fun, colorful) UIs to be developed in design tools and then plugged into the application at runtime. Updating the UI becomes as simple as exporting a new FXD/FXZ file.

## 9.6 Summary

In this chapter we looked at writing a practical application, with a swish front end, focusing on a couple of important JavaFX skills. First of all we looked at how to take the hard work of a friendly neighborhood graphic designer, bring it directly into our JavaFX project, and manipulate it as part of the JFX scene graph. Then we examined how to take our own hard work and magically transform it into an applet capable of

being dragged from the browser and turned into a JWS application. (Okay, okay, it's not *actually* magic!)

I hope this chapter, as well as being a fun little project, has demonstrated how easy it is to move JavaFX software from one environment to another and how simple it is to move multimedia elements from artist to programmer. The addition of a Stage extension was all it took to add applet-specific capabilities to our Enigma machine, and conversion into FXZ files was all it took to turn our vector images into programmable elements in our project.

There are plans to bring JavaFX to many different types of device in the future. The ability to leap across environments in a single bound would be a welcome change to the current drudgery of moving applications between platforms. The bonus section revealed how entire UIs could be drawn by a designer and then hooked up directly into JavaFX code. Imagine if we could switch the whole design of our application for desktop, mobile, or TV by merely choosing which FXZ file was loaded on startup! It's this sense of freedom, in both *how* we work and *where* our code can run, that will be central to JavaFX as it evolves in years to come.

So much for the future. For now, I'll just set the Enigma rotors to an appropriate three letters (I'll let you guess what they might be) and leave you with the simple departing message "NIAA CHZ ZBPS DVD AWOBHC RKNAHI."