4 Swing by numbers

This chapter covers

- Practicing core JavaFX Script skills
- Using model/view/controller, without reams of boilerplate
- Building a user interface that swings
- Validating a simple form

You've had to take in a lot in the last couple of chapters—an entirely new language, no less! I know many of you will be eager to dive straight into creating media-rich applications, but you need to learn to walk before you can run. JavaFX Script gives us a lot of powerful tools for writing great software, but all you've seen thus far is a few abstract examples.

So for this, our first project, we won't be developing any flashy visuals or clever animations. Be patient. Instead we need to start putting all the stuff you learned over the last few dozen pages to good use. A common paradigm in UI software is Model/View/Controller, where data and UI are separate, interacting by posting updates to each other. The *model* is the data, while the *view/controller* is the display and its input. We're going to develop a data class and a corresponding UI to see how the language features of JavaFX Script allow us to bind them together (pun

only partially intended). But first we need to decide on a simple project to practice on, something fun yet informative.

We're going to develop a version of the simple, yet addictive, number puzzle game found in countless newspapers and magazines around the world. If you've never encountered such puzzles before, take a look at figure 4.1.

The general idea is to fill in the missing cells in a grid with unique numbers (in a standard puzzle, 1 to 9) in each *row*, each *column*, and each *box*. A successful solution is a grid completely filled in, without duplicates in any row, column, or box.

The number puzzle

Number puzzles like the one we're developing have been published in magazines and newspapers since the late nineteenth century. By the time of WWI, their popularity had waned, and they fell into obscurity. In the late 1970s the puzzle was reinvented, legend has it, by Howard Garns, an American puzzle author, and it eventually found its way to Japan where it gained the title "Sudoku." It took another 25 years for the puzzle to become popular in the West. Its inclusion in the UK's *Sunday Times* newspaper was an overnight success, and from there it has gone on to create addicts around the world.

By far the most common puzzle format is a basic 9 x 9 grid, giving us nine rows of nine cells each, nine columns of nine cells each, and nine 3 x 3 boxes of nine cells each. At the start of the puzzle a grid is presented with some of the numbers already in place. The player must fill in the missing cells using only the numbers 1 to 9, such that all 27 *groups* (nine rows, nine columns, nine boxes) contain only one occurrence of each number.

We'll be using JavaFX's `javafx.ext.swing` package to develop our UI. These classes wrap Swing components, allowing us to use Java's UI toolkit in a JavaFX desktop application. For those of you who have developed with Swing in the past, this will be a real eye opener. You'll see firsthand how the same UIs you previously created with reams and reams of Java code can be constructed with relatively terse declarative JavaFX code. For those who haven't encountered the delights of Swing, this will be a gentle introduction to creating traditional UIs with the power tools JFX provides. Either way, we'll have fun.

This project is not a comprehensive Swing tutorial. Swing is a huge and very complex beast, with books the size of telephone directories published about it. JavaFX

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 4.1 A number puzzle grid, shown both empty and recently completed

itself provides direct support (JFX wrappers) for only a handful of core Swing components, although the whole of Swing can be used directly as Java objects, of course. The project is primarily about showing how a Swing-like UI can be constructed quickly and cleanly, using JavaFX Script.

JavaFX and Swing

JavaFX has two UI toolkits: its own `javafx.scene.control` library and the `javafx.ext.swing` wrappers around Java's Swing. What's the difference? The Swing wrappers will allow desktop apps to have native look 'n' feel, but Swing can't easily be ported to phones and TVs. So JavaFX's own library will allow greater portability between devices and better harmony with JavaFX's *scene graph*.

The new controls are really where the engineers at Sun want JavaFX to go; the Swing wrappers were initially a stop-gap until the controls library was ready. But the Swing wrappers are unlikely to vanish for a while—some developers of existing Swing applications have expressed interest in moving over to JavaFX Script for their GUI coding. The Swing library may, eventually, become an optional extension, rather than a standard JavaFX API.

Enough about Swing—what about our number puzzle? I don't know if you've ever noticed, but often the simpler the idea, the harder it is to capture in words alone. Sometimes it's far quicker to learn by seeing something in action. Our number puzzle uses blissfully simple rules, yet it's hard to describe in the abstract. So to avoid confusion we need to agree on a few basic terms before we proceed:

- The *grid* is the playing area on which the puzzle is played.
- A *row* is horizontal line of cells in the grid.
- A *column* is a vertical line of cells in the grid.
- A *box* is a subgrid within the grid.
- A *group* is any segment of the grid that must contain unique numbers (all rows, columns, and boxes).
- A *position* is a single cell within a group.

The elements are demonstrated in figure 4.2.

Rather than throw everything at you at once, we're going to develop the application piece by piece, building it as we go. With each stage we'll add in a more functionality, using the language features we learned in the previous two chapters, and you'll see how they can be employed to make our application work.

Right then, compiler to the ready, let's begin.

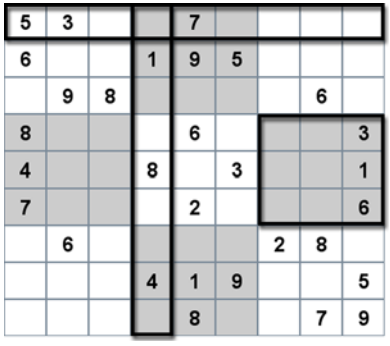

Figure 4.2 Groups are rows, columns, or boxes within the grid, which must hold unique values.

4.1 Swing time: Puzzle, version 1

What is Swing? When Java first entered the market, its official UI API was known as AWT (Abstract Window Toolkit), a library that sought to smooth over the differences between the various *native* GUI toolkits on Windows, the Mac, Linux, and any other desktop environment it was ported to. AWT wrapped the operating system's own native GUI widgets (buttons, scrollbars, text areas, etc.) to create a consistent environment across all platforms. Yet because of this it came under fire as being a lowest-common-denominator solution, a subset of only the features available on all platforms. So in answer to this criticism a more powerful alternative was developed: Swing!

Swing sits atop AWT but uses only its lowest level features—pixel pushing and keyboard/mouse input mainly—to deliver an entirely Java-based library of UI widgets. Swing is a large and very powerful creature, quite possibly one of the most powerful (certainly one of the most complex) UI toolkits ever created. In this project, we'll be looking at only a small part of it.

Because we're developing our puzzle bit by bit, in version 1 we won't expect to have a working game. We're laying the foundations for what's to come.

What's in a name?

The generic name for a UI control differs from system to system and from toolkit to toolkit. In the old Motif (X/X-Windows) toolkit they were called *widgets*; Windows uses the boring term *controls*, Java AWT/Swing calls them by the rather bland name *components*, and I seem to recall the Amiga even referred to them by rather bizarre name *gadgets*. Looks like nobody can agree!

With so many terms in use for the same thing, the conversation could quickly become confusing. Therefore, when referring to GUI elements in the abstract, I'll use the term *widgets*; when referring specifically to Swing or AWT elements, I'll use the term *components*; and when referring specifically to JavaFX elements, I'll use the term *controls*.

4.1.1 Our initial puzzle data class

We need to start somewhere, so here's some basic code that will get the ball rolling, defining the data we need to represent our puzzle. Listing 4.1 is our initial shot at a main puzzle grid class, but as you'll see it's far from finished.

Listing 4.1 PuzzleGrid.fx (version 1)

```
package jfxia.chapter4;

package class PuzzleGrid {
    public-init var boxDim:Integer = 3;
    public-read def gridDim:Integer = bind boxDim*boxDim;
    public-read def gridSize:Integer = bind gridDim*gridDim;

    package var grid:Integer[] =
        for(a in [0..<gridSize]) { 0 }
}
```

So far all we have is four variables, which do the following:

- The `boxDim` variable is the dimension of each box inside the main grid. Boxes are square, so we don't need separate width and height dimensions.
- The `gridDim` variable holds the width and height of the game grid. The grid is square, so we don't need to hold separate values for both dimensions. This value is always the square of the `boxDim` variable—a 3 x 3 box results in a 9 x 9 grid, a 4 x 4 box results in a 16 x 16 grid, and so on—so we utilize a bind to ensure they remain consistent.
- The `gridSize` variable is a convenience for when we need to know the total number of cells in the grid. You'll note that we're using a bind here too; when `gridDim` is changed, `gridSize` will automatically be updated.
- The `grid` itself, represented as a sequence of `Integer` values. We'll create an initial, default, grid with all zeros for now using a `for` loop.

So far, so good. We've defined our basic data class and used some JavaFX Script cleverness to ensure `gridSize` and `gridDim` are always up to date whenever the data they depend on changes. When `boxDim` is set, it begins a chain reaction that sees `gridDim` and then `gridSize` recalculated. Strictly speaking, it might have made more sense to bind `boxDim` to the square root of `gridDim` rather than the other way around, but I don't fancy writing a square root function for a project like this.

Note that although the puzzle requires the numbers 1 to 9, we also use the number 0 (zero) to represent an empty cell. Thus the permissible values for each cell range from 0 to 9.

Our initial data class is missing a lot of important functionality, but it should be sufficient to get a UI on the screen. We can then further refine and develop both the puzzle class and the interface as the chapter progresses.

4.1.2 Our initial GUI class

So much for the puzzle grid class; what about a GUI?

Recall that JavaFX encourages software to be built in a declarative fashion, especially UIs. Until now this has been a rather cute idea floating around in the ether, but right now you're finally about to see a concrete example of how this works in the real world and why it's so powerful.

Listing 4.2 is the entry point to our application, building a basic UI for our application using the previously touted declarative syntax.

Listing 4.2 Game.fx (version 1)

```
package jfxia.chapter4;

import javafx.ext.swing.SwingButton;
import javafx.scene.Scene;
import javafx.scene.text.Font;
import javafx.stage.Stage;

def cellSize:Number = 40;            <-- Cell dimensions
```

```
def puz = PuzzleGrid { boxDim:3 };

def gridFont = Font {
    name: "Arial Bold"
    size: 15
};

Stage {
    title: "Game"
    visible: true
    scene: Scene {
        content: for(idx in [0..<puz.gridSize]) {
            var x:Integer = (idx mod puz.gridDim);
            var y:Integer = (idx / puz.gridDim);
            SwingButton {
                layoutX: x * cellSize;
                layoutY: y * cellSize;
                width: cellSize;
                height: cellSize;

                text: bind notZero(puz.grid[idx]);
                font: gridFont;
                action: function():Void {
                    var v = puz.grid[idx];
                    v = (v+1) mod (puz.gridDim+1);
                    puz.grid[idx] = v;
                }
            }
        }
        width: puz.gridDim * cellSize;
        height: puz.gridDim * cellSize;
    }
}

function notZero(i:Integer):String { if(i==0) " " else "{i}"; }
```

Our puzzle class

A font we wish to reuse

Index to grid x/y

Each grid cell is a Swing button

Hide zero

You can see from the `import` statements at the head of the source file that it's pulling in several GUI-based classes from the standard JavaFX APIs. The Swing wrapper classes live in a JavaFX package named, conveniently, `javafx.ext.swing`. We could use the classes in the original Swing packages directly (like other Java API classes, they *are* available), but the JFX wrappers make it easier to use common Swing components declaratively.

The `cellSize` variable defines how big, in pixels, each square in the grid will be. Our game needs a `PuzzleGrid` object, and we create one with the variable `puz`, setting `boxDim` to declaratively describe its size. After `puz` we create a font for our GUI. Since we're using the same font for each grid cell, we may as well create one font object and reuse it. Again, this is done declaratively using an object literal. The final chunk of the listing—and quite a hefty chunk it is too—consists of the actual user interface code itself.

We've using Swing buttons to represent each cell in the grid, so we can easily display a label and respond to a mouse click, but let's strip away the button detail for the moment and concentrate on the outer detail of the window.

```
Stage {
    title: "Game"
    visible: true
    scene: Scene {
        content: /*** STRIPPED, FOR NOW ***/
        width: puz.gridDim * cellSize
        height: puz.gridDim * cellSize
    }
};
```

Here's an abridged reproduction of the code we saw in listing 4.2. We see two objects being created, one nested inside the other. At the outermost level we have a `Stage`, and inside that a `Scene`. There are further objects inside the `Scene`, but the previous snippet doesn't show them.

The `Stage` represents the link to the outside world. Because this is a desktop application, the `Stage` will take the form of a window. In a web browser applet, the `Stage` would be an applet container. Using a common top-level object like this aids portability between desktop, web, mobile, TV, and the like.

Three variables are set on `Stage`: the window's title as shown in its drag bar, the window's visibility to determine whether it's shown (we didn't *need* to set this because it's `true` by default), and the window's content. The content is a `Scene` object, used to describe the root of a scene graph. We'll look at the scene graph in far more detail in the next chapter; for now all you need to know is that it's where our UI will live.

The `Scene` object has its own variables, aside from `content`. They are `width` and `height`, and they determine the size of the inner window area (the part inside the borders and title bar). The window will be sized around these dimensions, with the total window size being the `Scene` dimensions plus any borders and title (drag) bars the operating system adds to decorate the window.

There's a chunk of code missing from the middle of the previous snippet, and now it's time to see what it does.

4.1.3 Building the buttons

Here's a reminder of the mysterious piece of code we left out of our discussion in the last section:

```
content: for(idx in [0..<puz.gridSize]) {
    var x:Integer = (idx mod puz.gridDim);
    var y:Integer = (idx / puz.gridDim);
    SwingButton {
        layoutX: x * cellSize;
        layoutY: y * cellSize;
        width: cellSize;
        height: cellSize;

        text: bind notZero(puz.grid[idx]);
        font: gridFont
        action: function():Void {
            var v = puz.grid[idx];
            v = (v+1) mod (puz.gridDim+1);
```

```
                puz.grid[idx] = v;
            }
        }
}
```

The code goes inside a `Scene`, which in turn provides the contents for our `Stage`, you'll recall—but what does it do? Put simply, it creates a square grid of `SwingButton` objects, tied to data inside the `puz` object. You can see the effect in figure 4.3.

Figure 4.3 The game as it appears after clicking on a few cells (note the highlight on the lower 3). Depending on your JRE version, you'll get Ocean- (left) or Nimbus- (right) themed buttons.

The `for` loop runs for as many times as the puzzle's grid size, creating a fresh button for each cell. Before the button is created we convert the loop index, stored in `idx`, into an actual grid `x` and `y` position. Dividing by the grid width gives us the `y` position (ignoring any fractional part), while the remainder of the same division gives us the `x` position.

Each button has seven variables declaratively set. The first four are the button's position and size inside the scene, in pixels. They lay the buttons out as a grid, relying on the `cellSize` variable we created at the head of the file. The three other variables are the button's text, its font, and an event handler that fires when the button is clicked.

The button's text is bound to the corresponding element in the puzzle's grid sequence. Thus the first button will be bound to the first sequence element, the second to the second, and so on, as shown in figure 4.4. We do not want the value 0 to be shown to the user, so we've created a convenience function called `notZero()`, which you can see at the foot of the script. This function returns any `Integer` passed into it as a `String`, except for 0, which results in a space.

You may recognize the `action` as an anonymous function. The function reads its corresponding element in the puzzle grid sequence, increments it, ensuring that the

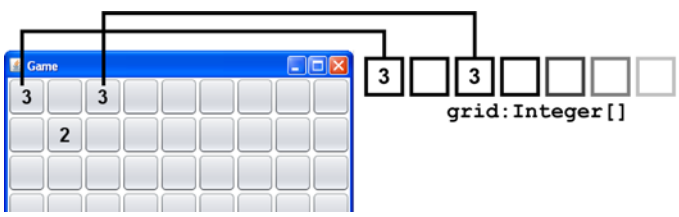

Figure 4.4 The text of each button is bound to the corresponding value in the puzzle's grid sequence.

Sometimes binds can be too clever!

Some of you may be wondering why it was necessary to wrap the button text's `if/else` code in the `notZero()` function. What's wrong with the following?

```
text: bind if(puz.grid[idx]==0) " " else "{puz.grid[idx]}"
```

The answer lies with the way binds work and how they strive to be as efficient as possible. When `puz.grid[idx]` goes from 0 to 1, the result of the condition changes from `true` to `false`, but when it goes from 1 to 2, the result remains `false`. The bind tries to be clever; it thinks to itself, "Well, the result of the condition hasn't changed, so I'm not going to reevaluate its body," and so the `SwingButton` displays 1 for every value except 0. Fortunately, by hiding the `if` condition inside a black-box function (see chapter 2, section 2.6.8), we can neatly sidestep bind's cleverness.

value wraps around to 0 if it exceeds the maximum number allowed, and then stores it back in the sequence. Here's the clever part: because the `text` variable is bound to the sequence element, whenever `action`'s function changes the element value, the button's text is automatically updated.

4.1.4 Model/View/Controller, JavaFX Script style

We touched on the Model/View/Controller paradigm briefly in the introduction, and hopefully those readers familiar with the MVC concept will already have seen how this is playing out under JavaFX Script.

In languages like Java, MVC is implemented by way of interfaces and event classes. In JavaFX Script this is swept away in favor of a much cleaner approach built deep into the language syntax. The relationships between our game UI and its data are formed using binds and triggers. The *model* (game data) uses triggers to respond to input from the *view/controller* (UI display). In turn, the *view* binds against variables in the *model*, establishing its relationship with them as expressions. The relationship is shown in figure 4.5.

This is how MVC works in JavaFX Script. Any code that is dependent on a model expresses that dependency using a bind, and JavaFX Script automatically honors that relationship without the programmer having to manually maintain it. The beauty of

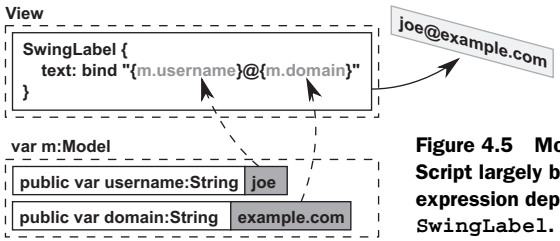

Figure 4.5 Model/View/Controller is achieved in JavaFX Script largely by way of bound expressions. Here one such expression depends on two strings for the contents of its `SwingLabel`.

this approach is it strips away the boilerplate classes and interfaces of other languages, like Java, distilling everything down to its purest form. If a given part of our UI is dependent on some external data, that dependency is expressed immediately (meaning inline) as part of its definition. It is not scattered throughout our code in disparate event handlers, interfaces, and event objects, as in Java.

If there is a two-way relationship between the UI and its data, a bidirectional bind (the `with inverse` syntax) can be used. For example, a text field may display the contents of a given variable in a model. If the variable changes, the text field should update automatically; if the text field is edited, the variable should update automatically. Providing the relationship is elemental in nature, in other words a direct one-to-one relationship, a bidirectional bind will achieve this.

When I told you binds were really useful things, I wasn't kidding!

4.1.5 Running version 1

It may not be the most impressive game so far, but it gives us a solid foundation to work from. When version 1 of the puzzle is run, it displays the puzzle grid (see figure 4.3) and responds to mouse clicks by cycling through the available numbers.

This is just a start, but already we've seen some of the power tools we learned about in the previous two chapters making a big contribution: the declarative syntax, bound variables, and anonymous functions are all in full effect.

So let's continue to build up the functionality of our game by making it more useful.

4.2 Better informed and better looking: Puzzle, version 2

So far we have a basic UI up and running, but it lacks the functionality to make it a playable game. There are two problems we need to tackle next:

- The buttons don't look particularly appealing, and it's hard to see where the boxes are on the puzzle grid.
- The game doesn't warn us when we duplicate numbers in a given group. If I, as the player, put two 3s on the same row, for example, the game does not flag this as an error.

In this section we'll remedy these faults. The first is entirely the domain of the UI class, `Game`, while the second is predominantly the domain of the data class, `PuzzleGrid`.

4.2.1 Making the puzzle class clever, using triggers and function types

The data class was tiny in version 1, with only a handful of variables to its name. To make the class more aware of the rules of the puzzle, we need to add a whole host of code. Let's start with PuzzleGrid.fx, shown in listing 4.3, and see what changes need to be made. Following the listing I'll explain the key changes, one by one. (In listing 4.3, and in other listings throughout this book, code unchanged from the previous revision is shown as slightly fainter, allowing the reader to immediately see where the additions/alterations are.)

Listing 4.3 PuzzleGrid.fx (version 2)

```
package jfxia.chapter4;

package class PuzzleGrid {
    public-init var boxDim:Integer = 3;
    public-read def gridDim:Integer = bind boxDim*boxDim;
    public-read def gridSize:Integer = bind gridDim*gridDim;

    package var grid:Integer[] =
        for(a in [0..<gridSize]) { 0 }
        on replace current[lo..hi] = replacement {
            update();
        }

    package var clashes:Boolean[] =
        for(a in [0..<gridSize]) false;

    function update() : Void {
        clashes = for(a in [0..<gridSize]) false;
        for(grp in [0..<gridDim]) {
            checkGroup(grp,row2Idx);
            checkGroup(grp,column2Idx);
            checkGroup(grp,box2Idx);
        }
    }

    function checkGroup (
        group:Integer ,
        func:function(:Integer,:Integer):Integer
    ) : Void {
        var freq = for(a in [0..gridDim]) 0;
        for(pos in [0..<gridDim]) {
            var val = grid[ func(group,pos) ];
            if(val > 0) { freq[val]++; }
        }
        for(pos in [0..<gridDim]) {
            var idx = func(group,pos);
            var val = grid[idx];
            clashes[idx] = clashes[idx] or (freq[val]>1);
        }
    }

    function row2Idx(group:Integer,pos:Integer) : Integer {
        return group*gridDim + pos;
    }

    function column2Idx(group:Integer,pos:Integer) : Integer {
        return group + pos*gridDim;
    }

    function box2Idx(group:Integer,pos:Integer) : Integer {
        var xOff = (group mod boxDim) * boxDim;
        var yOff = ((group/boxDim) as Integer) * boxDim;
        var x = pos mod boxDim;
        var y = (pos/boxDim) as Integer;
        return (xOff+x) + (yOff+y)*gridDim;
    }
}
```

Trigger an update to clashes

Does this cell clash with another?

Update clashes sequence

Check a given group for clashes

Whew! That's quite bit of code to be added in one go, but don't panic; it's all rather straightforward when you know what it's meant to do.

The purpose of listing 4.3 is to update a new sequence, called `clashes`, which holds a flag set to `true` if a given cell currently conflicts with others and `false` if it does not. The UI can then bind to this sequence, changing the way a grid cell is displayed to warn the player of any duplicates.

The function `update()` clears the `clashes` sequence and then checks each group in turn. In our basic 9 x 9 puzzle there are 27 groups: nine rows, nine columns, and nine boxes. Each group has nine positions it needs to check for duplicates. However, most of the work is deferred to the function `checkGroup()`, which handles the checking of an individual group. Let's take a closer look at this function, so we can understand how it fits into `update()`.

4.2.2 Group checking up close: function types

Here's the `checkGroup()` function we're looking at, reproduced on its own to refresh your memory and save ambiguity:

```
function checkGroup (
    group:Integer ,
    func:function(:Integer,:Integer):Integer
) : Void {
    var freq = for(a in [0..gridDim]) 0;
    for(pos in [0..<gridDim]) {
        var val = grid[ func(group,pos) ];
        if(val > 0) { freq[val]++; }
    }
    for(pos in [0..<gridDim]) {
        var idx = func(group,pos);
        var val = grid[idx];
        clashes[idx] = clashes[idx] or (freq[val]>1);
    }
}
```

The first four lines are the function's signature; unfortunately it's quite long, so I split it over four separate lines in an attempt to make it more readable. You can see the function is called `checkGroup`, and it accepts two parameters: an `Integer` and a function type. The function type accepts two `Integer` variables and gives a single `Integer` in return. The `Void` on the end signifies `checkGroup()` has no return value.

So, why do we need to pass a function to `checkGroup()`? Think about it: when we're checking each position in a row group we're working horizontally across the grid, when we're checking each position in a column group we're working vertically down the grid, and for a box group we're working line by line within a portion of the grid. Figure 4.6 demonstrates this.

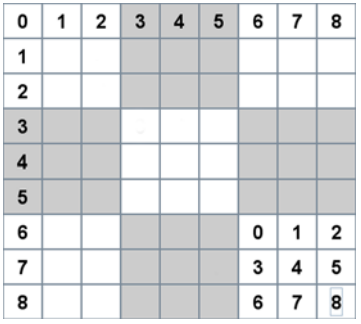

Figure 4.6 Coordinate translations for column, row, and box groups

We have three different ways of translating group and position to grid coordinates:

- For rows the group is the y coordinate in the grid and the position is the x coordinate. So position 0 of group 0 will be (0,0) on the grid, and position 2 of group 1 will be (2,1) on the grid.
- For columns the group is the x coordinate in the grid and the position is the y coordinate. So position 2 of group 1 will be (1,2) when translated into grid coordinates.
- For boxes we need to do some clever math to translate the group to a subsection of the grid, so group 8 (in the southeast corner) will have its first cell at coordinates (6,6). We then need to do some more clever math to turn the position into an offset within this subgrid. Position 1, for example, would be offset (1,0), giving us an absolute position of (7,6) on the grid when combined with group 8.

The code that actually checks for duplicates within a given group is identical, as we've just seen—all that differs is the way the group type translates its group number and position into grid coordinates. So the function passed in to `checkGroup()` abstracts away this translation. The two values it accepts are the group and the position. The value it returns is the grid coordinate, or rather the index in the grid sequence that corresponds with the group and position. (One of the benefits of storing the grid as a single-dimension sequence is that we don't need to figure out a way to return two values, an `x` and a `y`, from these three translation functions.)

Now that you understand what the passed-in function does, let's examine the code inside `checkGroup()` to see what it does. It's broken into two stages:

```
var freq = for(a in [0..gridDim]) 0;
for(pos in [0..<gridDim]) {
    var val = grid[ func(group,pos) ];
    if(val > 0) { freq[val]++; }
}
```

Here's the first stage reproduced on its own. We kick off by defining a new sequence called `freq`, with enough space for each unique number in our puzzle. This will hold the frequency of the numbers in our group. Recall that we're using 0 to represent an empty cell, so to make the code easier we've allowed space in the sequence for 0 plus the 1 to 9. Then we extract each position in the group, using the parameter function to translate group/position to a `grid` index. We increment the frequency corresponding to the value at the grid index—so if the value was 1, `freq[1]` would get incremented.

This builds us a table of how often each value occurs in the group. Now we want to act on that data.

```
for(pos in [0..<gridDim]) {
    var idx = func(group,pos);
    var val = grid[idx];
    clashes[idx] = clashes[idx] or (freq[val]>1);
}
```

The second stage of the `checkGroup()` function is reproduced here. It should be quite obvious what needs doing; we take a second pass over each position in the group, pulling out its value once again with help from our translation function. Then we set the flag in `clashes` if the value appears in the group more than once (signifying a clash!)

Note: a given cell in the puzzle grid may cause a clash in some groups but not others. It is important, therefore, that we preserve the clashes already discovered with other groups. This is why the clash check is `or`'d with the current value in the `clashes` sequence, rather than simply overwriting it.

4.2.3 Firing the update: triggers

Now we know how each group is checked, and we've seen the power of using function types to allow us to reuse code with *plug-in-able* variations, but we still need to complete the picture. The function `update()` will call `checkGroup()` 27 times (assuming a standard 9 x 9 grid), but what makes `update()` run?

Perhaps a better question might be, "When should `update()` run?" To which the answer should surely be, "Whenever the `grid` sequence is changed!"

We *could* wire something into our button event handler to always call `update()`, but this would be exposing the `PuzzleGrid` class's inner mechanics to another class, something we should avoid if we can. So why not wire something into the actual `grid` sequence itself?

```
package var grid:Integer[] =
    for(a in [0..<gridSize]) { 0 }
        on replace current[lo..hi] = replacement {
            update();
        }
```

Using a trigger we can ensure `update()` runs whenever the `grid` sequence is modified. It's a simple, effective, and clean solution that means the `clashes` sequence will never get out of step with `grid`.

4.2.4 Better-looking GUI: playing with the underlying Swing component

We've managed to soup up the puzzle class itself by making it responsive to duplicates under the rules of the puzzle. We can now exploit that functionality in our GUI, but we also need to make the game look more appealing.

Listing 4.4 gives us version 2 of the `Game` class.

Listing 4.4 Game.fx (version 2)

```
package jfxia.chapter4;

import javax.swing.JButton;
import javax.swing.border.LineBorder;
import javafx.ext.swing.SwingButton;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
```

```
import javafx.stage.Stage;

def gridCol1 = java.awt.Color.WHITE;
def gridCol2 = new java.awt.Color(0xCCCCCC);
def border = new LineBorder(java.awt.Color.GRAY);

def cellSize:Number = 40;

def puz = PuzzleGrid { boxDim:3 };

def gridFont = Font {
    name: "Arial Bold"
    size: 20
};

Stage {
    title: "Game"
    visible: true
    scene: Scene {
        content: for(idx in [0..<puz.gridSize]) {
            var x:Integer = (idx mod puz.gridDim);
            var y:Integer = (idx / puz.gridDim);
            var b = SwingButton {
                layoutX: x * cellSize;
                layoutY: y * cellSize;
                width: cellSize;
                height: cellSize;

                text: bind notZero(puz.grid[idx]);
                font: gridFont;
                foreground: bind
                    if(puz.clashes[idx]) Color.RED
                    else Color.GRAY;
                action: function():Void {
                    var v = puz.grid[idx];
                    v = (v+1) mod (puz.gridDim+1);
                    puz.grid[idx] = v;
                }
            };

            x/=puz.boxDim;  y/=puz.boxDim;
            var bg = if((x mod 2)==(y mod 2)) gridCol1
                else gridCol2;

            var jb = b.getJComponent() as JButton;
            jb.setContentAreaFilled(false);
            jb.setBackground(bg);
            jb.setOpaque(true);
            jb.setBorder(border);
            jb.setBorderPainted(true);
            b;
        }
        width: puz.gridDim * cellSize;
        height:puz.gridDim * cellSize;
    }
}

function notZero(i:Integer):String { if(i==0) " " else "{i}"; }
```

Two-tone background

Lines between grid cells

A Button reference

Bind the foreground color to clashes

Background color from index

Manipulate the button via Swing

Listing 4.4 adds a couple of new imports at the head of the file and two new variables: `gridCol1` and `gridCol2`. These will help us to change the background color of the buttons to represent the boxes on the grid. We'll use the `border` variable to give us a gray pin line around each box, as shown in figure 4.7.

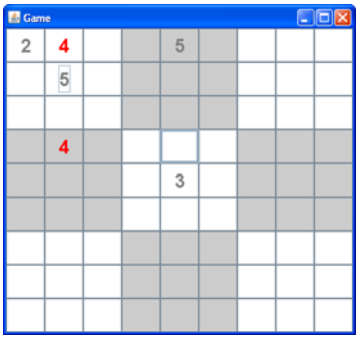

Figure 4.7 The restyled user interface, with differentiated boxes using background color and duplicate warnings using foreground (text) color

All the changes center on the button sequence being created and added into the `Scene`. You'll note from figure 4.7 that the button's `foreground` color is now bound to the `clashes` sequence we developed previously. You'll recall that when the contents of `puz.grid` change, `puz.clashes` is automatically updated via the trigger we added to the `PuzzleGrid` class. With this bind in place, the UI's buttons are immediately recolored to reflect any change in the clash status.

But those aren't the only changes we've made. In version 1 (listing 4.2) we added a new `SwingButton` straight into the scene graph, but now we capture it in a variable reference to manipulate it further before it gets added. After the button is created, we extract the underlying Swing component using the function `getJComponent()`, giving us full access to all its methods. First we remove the shaded fill effect. Next we set our own flat color background, creating a checkerboard effect for the boxes, and ensure the background is painted by making the button opaque. Finally we assign the `border` we created earlier, to create a gray pin line around each button.

Note that the background color is calculated by translating the cell's raw grid coordinate into a box coordinate (just scale them by the box size) and using this to assign colors based on a checkerboard pattern.

At the end of the loop we restate the variable used to hold the button reference. Because JavaFX Script is an expression language, this becomes our returned value to the loop's sequence.

With our new version of the game class in place, we're ready to try running the code again and seeing how the changes play out on the desktop.

4.2.5 Running version 2

Thanks to some clever trigger action, and a little bit of Swing coding, we've managed to get a puzzle game that now looks more the part and can warn players when they enter duplicate numbers within a group. See figure 4.7 for how the game currently looks.

It may seem like we're still a million miles away from a completed game, but actually we're in the home stretch, and the finishing line is within sight. So let's push on to the next, and final, version of the game.

Boxing clever: how to create a checkerboard pattern

In the `box2Idx()` function we witnessed in `PuzzleGrid`, and now with the checkerboard background pattern, we've had to do some clever math to figure out where the boxes are. Perhaps you're not interested in how this was done—but for those who are curious, here's an explanation.

We need to convert a sequence index (from 0 to 80, assuming a standard 9 x 9 grid) to a box coordinate (0,0) through (2,2) assuming nine boxes. The pattern itself is quite easy to produce once you have these coordinates: if x and y are both odd or both even, we use one color; if x and y are odd/even or even/odd, we use the other color. We can figure out whether a number is odd or even by dividing the remainder (the `mod`) of a division by 2: odd numbers result in 1; even numbers result in 0. Try it for yourself on a piece of paper if you don't believe me.

Let's assume we're given a `grid` sequence index, like 29; should this cell be shaded with a white or a gray background? First we convert the number 29 into a coordinate on the grid. The y coordinate is the number of times the grid dimension will divide into the number: 29 divided by 9 is 3. The x coordinate is the remainder of this division: 29 mod 9 is 2. Therefore index 29 is grid coordinate (x = 2,y = 3), assuming the coordinates start at 0. But we need to translate this into a box coordinate, which is easily done by scaling it by the box size: 2 divided by 3 is 0, and 3 divided by 3 is 1. So grid index 29 becomes grid coordinate (2,3) and becomes box coordinate (0,1). We then compare the oddness/evenness of these coordinates to determine which background shade to use.

What about the `box2Idx()` function? It's similar in principle, except we're almost working backwards: we're given a group and a position, and we need to work out the `grid` sequence index. To do this we first need to find the origin (northwest) coordinate of the box representing the group in the grid, which we can do by dividing and `mod`-ing the group by the box size to get x and y coordinates. We do the same thing to the position to get the coordinate offset within the box. Then we add the offset to the origin to get the absolute x and y within the grid. Finally, to convert the grid x,y to an index, we multiple y by the grid dimension and add on x.

4.3 Game on: Puzzle, version 3

We're almost there; the puzzle is nearly complete. But what work do we have left to do on our game? Let's make a list:

1. We need to add actual numbers for the start point of the puzzle; otherwise, the grid is just empty, and the puzzle wouldn't be very...well...puzzling.
2. We need to lock these starting numbers, so the player can't accidentally change them.
3. We need to notify the player when they've solved the puzzle. It would also be nice to inform them how many empty grid cells, and how many clashes, they currently have.

This is really just a mopping-up exercise, dealing with all the outstanding issues necessary to make the puzzle work. Yet there's still opportunity for learning, and for practicing our JavaFX skills, as you'll shortly see.

4.3.1 Adding stats to the puzzle class

In order to inform the player of how many clashes and empty cells we have, the puzzle class needs added functionality. We also need the class to provide some way of fixing the starting cells, so the GUI knows not to change them.

Listing 4.5 is the final version of the class, with all the new code added.

Listing 4.5 PuzzleGrid.fx (version 3)

```
package jfxia.chapter4;

package class PuzzleGrid {
    public-init var boxDim:Integer = 3;
    public-read def gridDim:Integer = bind boxDim*boxDim;
    public-read def gridSize:Integer = bind gridDim*gridDim;

    package var grid:Integer[] =
        for(a in [0..<gridSize]) { 0 }
        on replace current[lo..hi] = replacement {
            update();
        }

    package var clashes:Boolean[] =
        for(a in [0..<gridSize]) false;

    package var fixed:Boolean[] =
        for(a in [0..<gridSize]) false;

    package var numEmpty = 0;
    package var numClashes = 0;
    package def completed:Boolean = bind
        ((numEmpty==0) and (numClashes==0));

    public function fixGrid() : Void {
        for(idx in [0..<gridSize]) {
            fixed[idx] = (grid[idx]>0);
        }
    }

    function update() : Void {
        clashes = for(a in [0..<gridSize]) false;
        for(grp in [0..<gridDim]) {
            checkGroup(grp,row2Idx);
            checkGroup(grp,column2Idx);
            checkGroup(grp,box2Idx);
        }
        checkStats();
    }

    function checkGroup (
        group:Integer ,
        func:function(:Integer,:Integer):Integer
```

The new variables

Fix the static starting cells

Call the stats updater

```
    ) : Void {
        var freq = for(a in [0..gridDim]) 0;
        for(pos in [0..<gridDim]) {
            var val = grid[ func(group,pos) ];
            if(val > 0) { freq[val]++; }
        }
        for(pos in [0..<gridDim]) {
            var idx = func(group,pos);
            var val = grid[idx];
            clashes[idx] = clashes[idx] or (freq[val]>1);
        }
    }

    function checkStats() : Void {
        numEmpty = 0;
        numClashes = 0;
        for(idx in [0..<gridSize]) {
            if(grid[idx]==0)  numEmpty++;
            if(clashes[idx])  numClashes++;
        }
    }

    function row2Idx(group:Integer,pos:Integer) : Integer {
        return group*gridDim + pos;
    }

    function column2Idx(group:Integer,pos:Integer) : Integer {
        return group + pos*gridDim;
    }

    function box2Idx(group:Integer,pos:Integer) : Integer {
        var xOff = (group mod boxDim) * boxDim;
        var yOff = ((group/boxDim) as Integer) * boxDim;
        var x = pos mod boxDim;
        var y = (pos/boxDim) as Integer;
        return (xOff+x) + (yOff+y)*gridDim;
    }
}
```

Count empty and clashing cells

Four new variables have been added in listing 4.5: the first, `fixed`, is a sequence that denotes which cells in the grid should not be changeable. The next three provide basic stats about the puzzle: `numEmpty`, `numClashes`, and `completed`.

We've also added a new function, `fixGrid()`, which walks over the puzzle grid and marks any cells that are non-zero in the `fixed` sequence. The GUI class can call this function to lock all existing cells in the puzzle, but why didn't we use some clever device like a bind or a trigger to automatically update the `fixed` sequence?

The `grid` sequence gets updated frequently, indeed, each time the player changes the value of a cell in the puzzle. We need the `fixed` sequence to update only when the `grid` is initially loaded with the starting values of the puzzle. We could rather cleverly rewrite the trigger to spot when the entire `grid` is being written, rather than a single cell (it can see how many cells are being changed at once, after all), but this might cause confusion later on. For example, suppose we added a load/save feature to our

game. Restoring the grid after a load operation would cause the trigger to mistakenly fix all the existing non-zero cells, including those added by the player. Some may be wrong; indeed some may be clashes! How would the player feel if she were unable to change them?

For all the power JavaFX Script gives us, it must be acknowledged that sometimes the simplest solution is the best, even if it doesn't give us a chance to show fellow programmers just how clever-clever we are.

We have one final new function in our class: `checkStats()` populates the `numEmpty` and `numClashes` variables. It's called from the `update()` function, so it will run whenever the `grid` sequence is changed. The `completed` variable is bound to these variables and will become `true` when both are 0.

Let's now turn to the final piece of the puzzle (groan!), the GUI.

4.3.2 Finishing off the puzzle grid GUI

If you survived the horrendous pun at the end of the last section, you'll know this is the part where we pull everything together in one final burst of activity on the GUI class, to complete our puzzle game.

We have two aims with these modifications:

- Provide an actual starting grid, to act as a puzzle. The game is pointless without one.
- Plug in a status line at the bottom of the grid display, to inform the player of empty cells, clashing cells, and a successfully completed puzzle.

We'll look at the former in this section and the latter in the next section.

You might think these changes would be pretty mundane, but I've thrown in a layout class to keep you on your toes. Check out listing 4.6.

Listing 4.6 Game.fx (version 3)

```
package jfxia.chapter4;

import javax.swing.JButton;
import javax.swing.border.LineBorder;
import javafx.ext.swing.SwingButton;
import javafx.ext.swing.SwingLabel;
import javafx.scene.Scene;
import javafx.scene.layout.Flow;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;

def gridCol1 = java.awt.Color.WHITE;
def gridCol2 = new java.awt.Color(0xCCCCCC);
def border = new LineBorder(java.awt.Color.GRAY);

def cellSize:Number = 40;

var puz = PuzzleGrid {
    boxDim: 3
```

```
    grid: [
        5,3,0 , 0,7,0 , 0,0,0 ,
        6,0,0 , 1,9,5 , 0,0,0 ,
        0,9,8 , 0,0,0 , 0,6,0 ,
        8,0,0 , 0,6,0 , 0,0,3 ,
        4,0,0 , 8,0,3 , 0,0,1 ,
        7,0,0 , 0,2,0 , 0,0,6 ,
        0,6,0 , 0,0,0 , 2,8,0 ,
        0,0,0 , 4,1,9 , 0,0,5 ,
        0,0,0 , 0,8,0 , 0,7,9
    ]
};
puz.fixGrid();

var gridFont = Font {
    name: "Arial Bold"
    size: 20
};

Stage {
    title: "Game"
    visible: true
    scene: Scene {
        content: [
            for(idx in [0..<puz.gridSize]) {
                var x:Integer = (idx mod puz.gridDim);
                var y:Integer = (idx / puz.gridDim);
                var b = SwingButton {
                    layoutX: x * cellSize;
                    layoutY: y * cellSize;
                    width: cellSize;
                    height: cellSize;
                    text: bind notZero(puz.grid[idx]);
                    font: gridFont;
                    foreground: bind
                        if(puz.clashes[idx]) Color.RED
                        else if(puz.fixed[idx])  Color.BLACK
                        else Color.GRAY;
                    action: function():Void {
                        if(puz.fixed[idx]) { return; }
                        var v = puz.grid[idx];
                        v = (v+1) mod (puz.gridDim+1);
                        puz.grid[idx] = v;
                    }
                };

                x/=puz.boxDim;  y/=puz.boxDim;
                var bg = if((x mod 2)==(y mod 2)) gridCol1
                    else gridCol2;

                var jb = b.getJComponent() as JButton;
                jb.setContentAreaFilled(false);
                jb.setBackground(bg);
                jb.setOpaque(true);
                jb.setBorder(border);
                jb.setBorderPainted(true);
                b;
```

Some puzzle data, at last!

Fix the initial puzzle cells

Grid loop now inside larger sequence

```
            } ,                          <-- End of grid loop
            Flow {
                layoutX: 10;
                layoutY: bind puz.gridDim * cellSize;
                hgap: 50;
                content: [
                    SwingLabel {
                        text: bind "Empty: {puz.numEmpty}"
                            "   Clashes: {puz.numClashes}";
                    } ,
                    SwingLabel {
                        text: "Complete!";
                        visible: bind puz.completed;
                        foreground: Color.GREEN
                    }
                ]
            }
        ]
        width: puz.gridDim * cellSize;
        height:puz.gridDim * cellSize + 20;    <-- Allow for status line
    }
}
function notZero(i:Integer):String { if(i==0) " " else "{i}"; }
```

The status panel

You can see a couple of new imports at the head of the file—one is a Swing label class, and the other is the promised JFX layout class.

You'll note the `grid` variable of the `PuzzleGrid` class is now being set, and `fixGrid()` is being called to lock the initial puzzle numbers in place. You could provide your own puzzle data here if you want; I used the data that illustrates the Wikipedia Sudoku article as an example.

In this version of the game we'll be adding more elements to the scene graph beyond those created by the `for` loop. For this reason the loop (which creates a sequence of Swing buttons) has been moved inside a set of square brackets, effectively wrapping it inside a larger sequence. Embedded sequences like this are expanded in place, you'll recall, so the buttons yielded from the loop are expanded into the outer sequence, and our new elements (discussed later) are added after them.

Taking a closer look at the `SwingButton` definition you see a couple of minor but important additions. Here's the snippet of code we're talking about, extracted out of the main body of the button creation:

```
foreground: bind
    if(puz.clashes[idx]) Color.RED
    else if(puz.fixed[idx])  Color.BLACK
    else Color.GRAY
action: function():Void {
    if(puz.fixed[idx]) { return; }
    var v = puz.grid[idx];
    v = (v+1) mod (puz.gridDim+1);
    puz.grid[idx] = v;
}
```

In the `foreground` code we now look for and colorize fixed cells as solid black. And to complement this, in the event handler we now check for unchangeable cells, exiting if one is clicked without modifying its contents. These two minor changes, coupled with the work we did with the `PuzzleGrid` class, ensure the starting numbers of a puzzle will not be editable.

4.3.3 Adding a status line to our GUI with a label

The bulk of the changes come with the introduction of a status line to the foot of the GUI declaration. Here, for your convenience, is the code once more, devoid of its surrounding clutter:

```
Flow {
    layoutX: 10;
    layoutY: bind puz.gridDim * cellSize;
    hgap: 50;
    content: [
        SwingLabel {
            text: bind "Empty: {puz.numEmpty}"
                "   Clashes: {puz.numClashes}";
        } ,
        SwingLabel {
            text: "Complete!";
            visible: bind puz.completed;
            foreground: Color.GREEN
        }
    ]
}
```

This code is placed inside a larger sequence, used to populate the `Scene`. The UI elements it defines appear after all the grid buttons, which were created using a `for` loop, as you saw earlier.

You'll immediately notice a `Flow`, which is one of JavaFX's layout nodes, otherwise known as a *container*. It controls how its children are positioned on screen. Look at figure 4.8 and you'll see another example of `Flow` in action. The gray shapes are arranged, from left to right, in the order in which they appear inside the `content` sequence.

Like the `Scene` class, `Flow` accepts a sequence for its contents but doesn't add any new graphics itself. Instead it positions its children on the display by placing them one after another, either horizontally or vertically (depending upon how it has been configured), wrapping onto a new row or column when necessary. It has several options, including `hgap`, which determines the number of pixels to place between consecutive nodes in a row. The `javafx.scene.layout` package has several different container nodes, each specializing in a different geometry, and no doubt new ones will be added with each JavaFX release. Using these layout nodes we can place screen

Figure 4.8 Elements inside a `Flow` are lined up in rows or columns, wrapping when necessary.

objects in relation to one another without resorting to absolute x/y positioning as we did with the button grid.

To ensure the status line is at the foot of the display, we use its `layoutY` variable to lower it below the button grid. It you check out the `Scene`, you'll note it has had its `height` amended to accommodate the new content. You can see how it looks in figure 4.9.

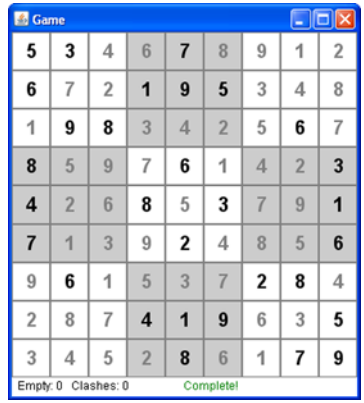

Figure 4.9 The puzzle game with its status panel, implemented using `Flow`

To implement the status bar we need to arrange five pieces of text, so we'll use `SwingLabel` classes. These are the equivalent of the Swing `JLabel` class, designed to show small quantities of text (typically used for labeling, hence the name) on our UI. We can group the "Empty" and "Clashes" text into one label, but the "Completed!" text needs to be a different color. So we need two labels.

```
SwingLabel {
    text: bind "Empty: {puz.numEmpty}"
        "   Clashes: {puz.numClashes}";
} ,
SwingLabel {
    text: "Complete!";
    visible: bind puz.completed;
    foreground: Color.GREEN
}
```

Here's the declarative code for the two labels again, and you can immediately see how they are both bound to variables in the `puz` object. The first label binds its text to a string expression containing the number of empty and clashing cells; the second shows the "Complete!" message, its visibility dependent on the `puz.completed` variable.

So there we go—our GUI!

4.3.4 Running version 3

We now have a functioning number puzzle game, as depicted in figure 4.9. Clicking on a changeable cell causes its numbers to cycle through all the possibilities. Duplicate numbers (clashes) are shown in red, while the fixed numbers of the initial puzzle are shown in black. The status bar keeps track of our progress and informs us when we have a winning solution.

Almost all of this was done by developing an intelligent puzzle class, `PuzzleGrid`, which automatically responds to changes in the grid with updates to its other data. The GUI in turn is bound to this data, using it to colorize cells in the grid and show status information.

A single click on a button sets off a chain reaction, updating the grid, which updates the other status variables, which updates the GUI. Once the bind and trigger

relationships are defined, the code runs automatically, without us having to prompt it each time the grid is altered. In our example the grid gets altered from only one place (the anonymous function event handler on each button), so this might not seem like much of a saving, but imagine how much easier life would be if we expanded our game to include a load/save feature or a hint feature—both of which alter the grid contents. Indeed, it becomes no extra work at all, so long as the relationships between the variables are well defined through binds.

4.4 Other Swing components

In our lightning tour of JavaFX's Swing support we looked at a couple of common widgets, namely `SwingButton` and `SwingLabel`. We also looked at some core scene graph containers like `Stage`, `Scene`, and `Flow`. Obviously it's hard to create an example project that would include every different type of widget, so here's a quick rundown of a few key UI classes we didn't look at:

- `SwingCheckBox`—A button that is either checked or unchecked.
- `SwingComboBox`—Displays a drop-down list of items, optionally with a free text box.
- `SwingList` and `SwingListItem`—Displays a list of items from which the user can select.
- `SwingRadioButton`, `SwingToggleGroup`—Together these classes allow for groups of buttons, in which only one button is selectable at any given time.
- `SwingScrollPane`—Allows large UI content to be displayed through a restricted *viewport*, with scrollbars for navigation. Useful if you have a big panel of widgets, for example, which you want to display inside a scrollable area.
- `SwingSlider`—A thumb and track widget, for selecting a value from a range of possibilities using a mouse.
- `SwingTextField`—Provides text entry facilities, unsurprisingly.

These are just a few of the classes in the `javafx.ext.swing` package, and Swing itself provides many more. You could get some practice with them by expanding our puzzle application; for example, how about a toggle that switches the highlighting of clashing cells on or off? This could be done by way of a `SwingCheckBox`, perhaps.

4.5 Bonus: using bind to validate forms

This chapter has been a fun way to introduce key JavaFX Script language constructs, like binds and triggers. These tools are very useful, particularly for things like form validation. Before we move on, let's detour to look at a bonus example, by way of listing 4.7.

Listing 4.7 Using bind for form validation

```
import javafx.ext.swing.*;
import javafx.geometry.VPos;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
```

```
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

var ageTF:SwingTextField;
def ageValid:Boolean = bind checkRange(ageTF.text,18,65);

Stage {
    scene: Scene {
        content: VBox {
            layoutX: 5;  layoutY: 5;
            content: [
                Flow {
                    nodeVPos: VPos.CENTER;
                    hgap: 10;  vgap: 10;
                    width: 190;
                    content: [
                        SwingLabel {
                            text: "Age: ";
                        },
                        ageTF = SwingTextField {
                            columns: 10
                        },
                        Circle {
                            radius: bind
                                ageTF.layoutBounds.height/4;
                            fill: bind if(ageValid)
                                Color.LIGHTGREEN else Color.RED;
                        },

                        SwingButton {
                            text: "Send";
                            disable: bind not ageValid;
                        }
                    ]
                }

            ]
        }
        width: 190;  height: 65;
    }
}

function checkRange(s:String,lo:Integer,hi:Integer) :Boolean {
    try {
        def i:Integer = Integer.parseInt(ageTF.text);
        return (i>=lo and i<=hi);
    }
    catch(any) { return false; }
}
```

This self-contained demo uses a function, `checkRange()`, to validate the contents of a text field. Depending on the validity state, an indicator circle changes and the Send button switches between disabled and enabled. We'll be dealing with raw shapes like circles in the next chapter, so don't worry too much about the unfamiliar code right now; the important part is in the binds involving `ageValid`.

The circle starts out red, and the button is disabled. As we type, these elements update, as shown in figure 4.10. An age between 18 and 65 changes the circle's color and enables the button, all thanks to the power of binds (you may need to squint to see the Swing button's subtle appearance change). The `ageValid` variable is bound to a function for checking whether the text field content is an integer within the specified range. This variable is in turn bound by the circle and the Send button.

Figure 4.10 Age must be between 18 and 65 inclusive. Incorrect content shows a red circle and disables Send (left and right); correct content shows a light-green circle and enables Send (middle).

In a real application we would have numerous form fields, each with its own validity boolean. We would use all these values to control the Send button's behavior. We might also develop a convenience class using the circle, pointing it at a UI component (for sizing) and binding it to the corresponding validity boolean. In the next chapter we'll touch on creating custom graphic classes like this, but for now just study the way bind is used to create automatic relationships between parts of our UI.

4.6 Summary

In this chapter we developed a working number-puzzle game, complete with a fully responsive desktop GUI. And in less than 200 lines of code—not bad!

Assuming you haven't fallen foul to our fiendish number puzzle game (may I remind you, the screen shots give the solution away, so there's really no excuse!), you've—I hope—learned a lot about writing JavaFX code during the course of this chapter. Although the number puzzle wasn't the most glamorous of applications in terms of visuals, it was still fun, I think, and afforded us much-needed practice with the JFX language. And that's all we need right now.

The game could be improved; for example, it would be nice for the cells to respond to keyboard input so the player didn't have to cycle through each number in turn, but I'll leave that as an exercise to the reader. You've seen enough of JavaFX by now that you can extract the required answers from the API documentation and implement them yourself.

In the next chapter we're getting up close and personal with the scene graph, JavaFX's backbone for presenting and animating flashy visuals. So make sure you pack your ultratrendy shades.