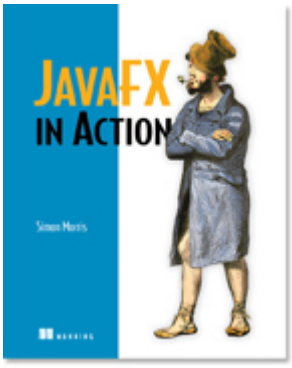
MANNING PUBLICATIONS

Sequences: JavaFX arrays

Excerpted from



JavaFX in Action
EARLY ACCESS EDITION

Simon Morris

MEAP Release: August 2008
Softbound print: May 2009 (est.) | 375 pages
ISBN: 1933988991

*This article is based on chapter 2 from **JavaFX in Action** by Simon Morris.*

Sequences are arrays by another name, with some clever extra functionality, making them more useful to the type of work JavaFX is designed to do. They differ widely from Java primitive arrays; indeed, they have more in common with Java's collection classes.

In the following sections we'll look at how to define, extend, retract, slice and filter JavaFX sequences. As you may already have guessed, sequences have quite a rich syntax associated with them, so let's jump straight in.

Basic sequence declaration and access

We won't get very far if we cannot define new sequences. The following code snippet shows us how to do just that:

```
import java.lang.System;
var seq1:String[] = [ "A" , "B" , "C" ];
var seq2:String[] = [ seq1 , "D" , "E" ];
var flag1 = (seq2 == ["A","B","C","D","E"]);
System.out.printf("seq1 = {seq1.toString()}"+
    "%nseq2 = {seq2.toString()}%nflag1 = {flag1}%n");

seq1 = [ A, B, C ]
seq2 = [ A, B, C, D, E ]
flag1 = true
```

The above code demonstrates a few important principles of sequences:

- A new sequence is declared using square brackets syntax.
- When one sequence is used inside another, it is expanded in place.
- Sequences are equal if they are the same size and each corresponding element is equal.
- The notation for referring to the type of a sequence uses square brackets after the plain object type – the same as Java primitive arrays. For example String[] would refer to a sequence of String objects.

To reference a value in a sequence we use the same square bracket syntax as many other programming languages:

```
import java.lang.System;
var faceCards = [ "Jack" , "Queen" , "King" ];
var king = faceCards[2];
var ints = [10,11,12,13,14,15,16,17,18,19,20];
var oneInt = ints[3];
System.out.printf("faceCards[2] = {king}%n"+
    "ints[3] = {oneInt}%n");

faceCards[2] = King
ints[3] = 13
```

As with Java arrays, square brackets surrounding an integer value are used to reference a given element within a sequence. The first element has the index zero.
So far we haven't seen anything spectacular – what about those clever features I promised above? Fair enough, let's experience our first bit of JFX sequence cleverness by looking at ranges and slices.

Sequence creation using ranges ([..], step, etc.)

The examples thus far have seen sequences created explicitly, by listing all their values as a comma separated list inside square brackets. This may not be convenient in many circumstances, so JFX supports a handy range syntax:

```
import java.lang.System;
var seq3 = [ 1 .. 100 ];                                                    A
System.out.println("seq3[0] = {seq3[0]},"
    +"seq3[11] = {seq3[11]}, seq3[89] = {seq3[89]}");

seq3[0] = 1,seq3[11] = 12, seq3[89] = 90
```

A Two dots creates a range

Here the sequence is populated with all the values from 1 to 100, inclusive. Two dots separate the start and end delimiters of the range, enclosed in the familiar square brackets. Simple enough – so, is that all there is to it? Oh no, not by a long stretch!

```
import java.lang.System;
var range1 = [0..100 step 5];
var range2 = [100..0 step -5];
var range3 = [0..100 step -5];
var range4 = [0.0 .. 1.0 step 0.25];
System.out.println("range1 = {range1.toString()}");
System.out.println("range2 = {range2.toString()}");
System.out.println("range3 = {range3.toString()}");
System.out.println("range4 = {range4.toString()}");

range1 = [ 0, 5, 10, 15, 20, 25, 30, 35, 40, 45,
[CA]50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100 ]
range2 = [ 100, 95, 90, 85, 80, 75, 70, 65, 60, 55,
[CA]50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0 ]
range3 = [ ]
range4 = [ 0.0, 0.25, 0.5, 0.75, 1.0 ]
```

The above shows various ranges created using an extra step parameter. The first runs from zero to one hundred in steps of five (0, 5, 10 ..) and the second does the same in reverse (100, 95, 90 ..) The third goes from zero to one hundred backwards, resulting (quite rightly!) in an empty sequence. Finally, just to prove ranges aren't all about integers, we have a range of type Number[] from zero to one in steps of a quarter.

We can include ranges inside larger declarations – for example, you'll recall we can expand one sequence inside another?

```
import java.lang.System;
var blackjackValues = [ [1..10] , 10,10,10 ];                          A
System.out.println("blackjackValues = "+
    "{blackjackValues.toString()}");

blackjackValues = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10 ]
```

A Expanding a range inside another declaration

The example above creates a sequence representing the card values in the game of Blackjack: aces to tens use their face value, while picture cards (jack, queen, king) are valued at ten. (Yes, I am aware aces are also eleven – what do you want, blood?!)

Sequence creation using slices ([..<] , etc.)

The range syntax can be useful in many circumstances, but it's not the only trick JavaFX Script has up its sleeve. For situations which demand a little more control, we can also take a slice from an existing sequence to create a new one:

```
import java.lang.System;
var source = [0 .. 100];
var slice1 = source[0 .. 10];
var slice2 = source[0 ..< 10];
var slice3 = source[95..];
var slice4 = source[95..<];
var format = "%s = %d to %d%n";
System.out.printf(format, "slice1",
    slice1[0], slice1[(sizeof slice1)-1] );
System.out.printf(format, "slice2",
    slice2[0], slice2[(sizeof slice2)-1] );
System.out.printf(format, "slice3",
    slice3[0], slice3[(sizeof slice3)-1] );
System.out.printf(format, "slice4",
    slice4[0], slice4[(sizeof slice4)-1] );

slice1 = 0 to 10                                                       A
slice2 = 0 to 9                                                        A
slice3 = 95 to 100                                                     A
slice4 = 95 to 99                                                      A
```

A Just the start/end values

Here we see the double dot syntax used to create a slice of an existing sequence, source. The numbers defining the slice are element indexes, so in the case of slice1 the range [0..10] refers to the first eleven elements in source, resulting in the values 0 to 10.

The '..' syntax describes an inclusive range, while the '..<' syntax can be used to define an exclusive range (0 to 10, not including 10 itself.) If you miss the trailing delimiter off a '..' range, the slice will be taken to the end of the sequence – effectively making the end delimiter the sequence size minus one. If you miss the trailing delimiter off a '..<' range the slice will be taken to the end of the sequence minus one element – effectively dropping the last index.

Sequence creation using a predicate

The final weapon in the sequence arsenal (and perhaps the most powerful) is the predicate syntax, which allows us to take a conditional slice from inside another sequence. The predicate syntax takes the form of a variable and a condition separated by a bar character. The destination (output) sequence is constructed by loading each element in the source sequences into the variable, and applying the condition to determine whether it should be included in the destination or not.

```
import java.lang.System;
var source2 = [0 .. 9];
var lowVals = source2[n|n<5];
```

```
System.out.println("lowVals = {lowVals.toString()}");
var people = ["Alan","Andy","Bob","Colin","Dave","Eddie"];
System.out.println("predicate = "+
    people[s | s.startsWith("A")].toString());

lowVals = [ 0, 1, 2, 3, 4 ]
predicate = [ Alan, Andy ]
```

Each of the numbers in source2 is assigned to n, and the condition n<5 is tested to determine whether the value will be added to lowVals . The second example applies a test to see if the sequence element begins with the character 'A', meaning in our example only "Alan" and "Andy" will make it into the destination sequence.

Predicates are pretty useful, particularly as their syntax is nice and compact. But even this isn't the end of what can do with sequences. Next up we'll complete our discussion by looking are editing existing sequences rather than creating new ones.

Sequence manipulation (insert, delete, reverse)

Sequences can be manipulated by inserting and removing elements dynamically. We can do this either to the end of the sequence, before an existing element, or after an existing element. The three variations are demonstrated with this piece of code:

```
import java.lang.System;
var seq1 = [1..5];
insert 6 into seq1;
System.out.println("Insert1: {seq1.toString()}");
insert 0 before seq1[0];
System.out.println("Insert2: {seq1.toString()}");
insert 99 after seq1[2];
System.out.println("Insert3: {seq1.toString()}");

Insert1: [ 1, 2, 3, 4, 5, 6 ]
Insert2: [ 0, 1, 2, 3, 4, 5, 6 ]
Insert3: [ 0, 1, 2, 99, 3, 4, 5, 6 ]
```

This quick example shows a basic range sequence created with the values 1 through 5. The first insert appends a new value, 6, to the end of the sequence, the next inserts a new value, 0, before the current first value, and the final insert shoehorns a value, 99, after the third element in the sequence.

```
import java.lang.System;
var seq2 = [1..10];
delete seq2[0];
System.out.println("Delete1: {seq2.toString()}");
delete seq2[0..2];
System.out.println("Delete2: {seq2.toString()}");
delete seq2;
System.out.println("Delete3: {seq2.toString()}");

Delete1: [ 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
Delete2: [ 5, 6, 7, 8, 9, 10 ]
Delete3: [ ]
```

It should be obvious what the above does. Starting with a sequence of 1 through 10, the first delete removes the first index, the second deletes a range from index positions 0 to 2 (inclusive), and the final delete simple removes the entire sequence.

One final trick is the ability to reverse the order of a sequence.

```
import java.lang.System;
var seq3 = [1..10];
seq3 = reverse seq3;
System.out.println("Reverse: {seq3.toString()}");

 Reverse: [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]
```

And finally a simple example to end the section on: the second line flips the order of the sequences, from 1 through 10, to 10 through 1.

And that's pretty much it for sequences. Over the last few pages we've learned about all manner of useful things we can do with JavaFX Script sequences, which we cannot do with the basic 'primitive' arrays in Java.