

JavaFX Technology

Learning the JavaFX Script Programming Language - Tutorial Overview


[Print-friendly Version](#) [Download tutorial](#)
[« Previous](#) | [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next »](#)

The JavaFX Script programming language lets you create modern looking applications with sophisticated graphical user interfaces. It was designed from the ground up to make GUI programming *easy*; its declarative syntax, data binding model, animation support, and built-in visual effects let you accomplish *more work with less code*, resulting in shorter development cycles and increased productivity.

This tutorial is your starting point for learning the JavaFX Script programming language. It focuses on the fundamentals only: that is, on the underlying, non-visual, core constructs that are common to all FX applications. When finished, you will be ready for [Building GUI Applications with JavaFX](#), the second tutorial in this series. After that, the [Media Browser](#) tutorial will walk you through the complete end-to-end development of a real-world application.

In addition, advanced developers will be interested in the [JavaFX Script Programming Language Reference](#) and [Application Programming Interface \(API\)](#) documentation. These reference documents provide a lower-level discussion of the syntax, semantics, and supported libraries of the JavaFX Script programming language and SDK.

Tutorial Contents

- [Tutorial Overview](#)
- [1. Getting Started](#)
- [2. Writing Scripts](#)
- [3. Using Objects](#)
- [4. Data Types](#)
- [5. Sequences](#)
- [6. Operators](#)
- [7. Expressions](#)
- [8. Data Binding and Triggers](#)
- [9. Writing Your Own Classes](#)
- [10. Packages](#)
- [11. Access Modifiers](#)

The lessons in this tutorial include:

- [Lesson 1: Getting Started with JavaFX Script](#) — Provides software download and installation instructions, plus a discussion on choosing an appropriate development environment.
- [Lesson 2: Writing Scripts](#) — Provides an introduction to compiling source code, running an application, declaring script variables, and invoking script functions.
- [Lesson 3: Using Objects](#) — Provides an introduction to objects, showing how to declare an object literal and how to invoke an object's functions.
- [Lesson 4: Data Types](#) — Discusses the built-in data types `String`, `Number`, `Integer`, `Boolean` and `Duration`, plus the use of `Void` and `null`.
- [Lesson 5: Sequences](#) — Introduces the *sequences* data structure, which is used to store and manipulate a list of objects. This lesson discusses how to create, use, and compare sequences and their subsets (called *slices*).
- [Lesson 6: Operators](#) — Introduces the supported operators (assignment, arithmetic, unary, relational, conditional and type comparison).
- [Lesson 7: Expressions](#) — The JavaFX Script programming language is an *expression language*. This lesson explains what that means, and discusses the different types of expressions that are available for you to use.
- [Lesson 8: Data Binding and Triggers](#) — One of the most powerful features of the JavaFX Script programming language is the ability to automatically synchronize a GUI with its underlying data. This lesson explores the basic mechanics of the data binding and trigger constructs.
- [Lesson 9: Writing Your Own Classes](#) — The JavaFX API provides a large number of classes for you to use in your applications. There may be times, however, when you will want to write a custom class of your own design. This lesson explores the basics of what is involved when doing so.
- [Lesson 10: Packages](#) — Placing your source files into named packages will make your code more organized in terms of namespace management. This lesson explores the creation and use of packages with a step-by-step example that walks you through the various considerations.
- [Lesson 11: Access Modifiers](#) — Access modifiers are used to specify various levels of visibility for your variables, functions and classes. This lesson explores the available access modifiers, and discusses what happens when no access modifier is provided.

In addition to the conceptual explanations and sample code, some lessons also include SDK demo code excerpts. Studying these snippets will help you to recognize patterns in `.fx` source files. It will also help solidify your understanding of each new concept.

[« Previous](#) | [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next »](#)



Learning the JavaFX Script Programming Language

Lesson 1: Getting Started with JavaFX Script

[Download tutorial](#)[« Previous](#) | [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next »](#)

Ready to explore the JavaFX Script programming language? Great! This lesson describes the software that must be installed on your system before you can begin. It also provides NetBeans IDE and command line instructions for compiling and running your first application.

Contents

- [Step 1: Download and install the JDK](#)
- [Step 2: Choose a Development Environment](#)
- [Step 3 : Download and Install the JavaFX Compiler and Runtime](#)

Step 1: Download and Install the JDK

The JavaFX Script programming language is based on the Java Platform, and as such, requires JDK 5 or JDK 6 (6 is faster) to be installed on your system. If you have not done so already, [download and install JDK 6](#) or [JDK 5](#) now, before proceeding with this tutorial.

Step 2: Choose a Development Environment

When it comes to choosing a development environment, you have two broad categories of choices: use an Integrated Development Environment (IDE), or use a plain text editor. This choice is entirely a matter of personal taste, but the following summary might help you to make an informed decision.

Generally speaking:

- An IDE provides a complete development environment all in one place. You download one piece of software, or perhaps a *plug-in* to that software, which provides everything you need to compile/run/debug your application. IDEs present the most commonly used functions as Graphical User Interface (GUI) elements, and offer many useful features, such as automatic code completion and custom source code views. An IDE also gives you immediate feedback on errors and highlights code so that it is easier to understand. The officially supported IDE for the JavaFX Script programming language is [NetBeans IDE 6.5](#). The NetBeans IDE website provides instructions for downloading, installing, and configuring the IDE.
- A text editor provides simplicity and familiarity. Experienced programmers often rely on their text editor of choice, preferring to work in that environment whenever possible (some editors, like `vi`, have a rich set of built-in keystroke commands that some programmers simply cannot live without!) If you already have a preferred editor, rest assured that you can download the sample code for each lesson as `.fx` source files for use in your editor of choice.

Step 3: Download and Install the JavaFX Compiler and Runtime

The JavaFX Script programming language is a *compiled* language, which means that any source code you write must first be converted into Java bytecode — the language of the Java Virtual Machine — before it can run on your system. This is true regardless of your development environment (be it command line or IDE). After installing the JDK and choosing a development environment, you will need to download and install the JavaFX Script compiler and runtime. The easiest way to obtain this software is to download the entire [JavaFX SDK](#), which gives you the NetBeans IDE (optional), compiler, runtime, and a number of other tools.

Another way is to simply [download the latest compiler binary](#) from the `openjfx` project website. The compiler itself is written in the Java programming language; installing the precompiled binary therefore becomes a matter of extracting the downloaded file and adding the `javafx` and `javafx` tools to your path. The complete set of instructions for this approach can be found on the [PlanetJFX Wiki](#).

Finally — if you want to live on the bleeding edge — you can join the [OpenJFX Compiler Project](#), create your own copy of the compiler workspace, and build everything yourself from the compiler source files. (If you choose this approach, you will also need the 1.7.0 version of [Apache Ant](#), plus a recent copy of [Mercurial](#)).

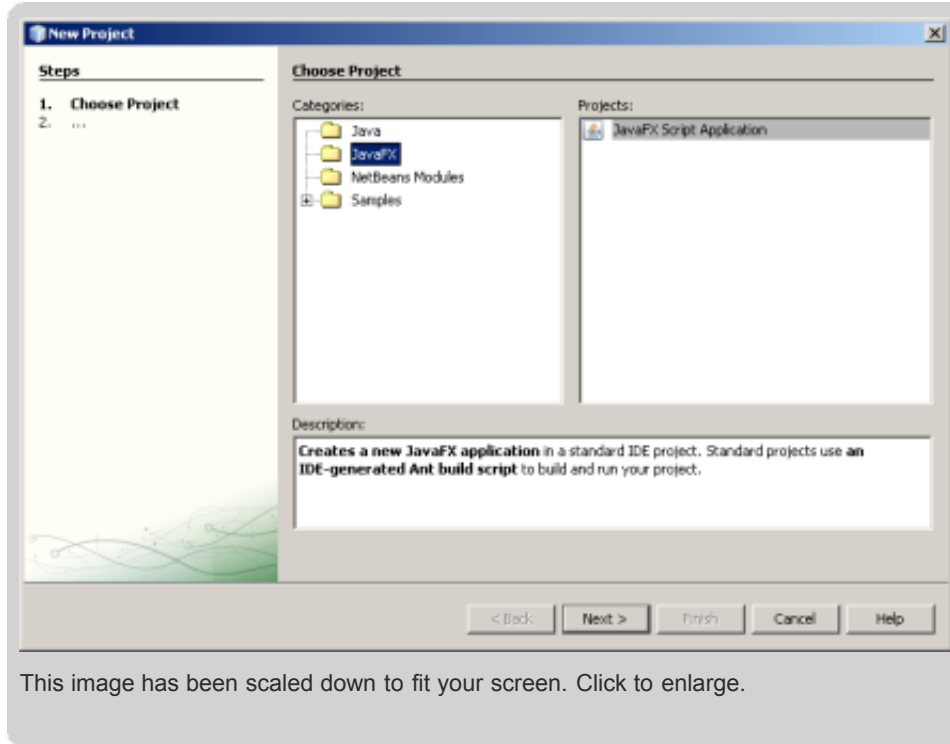
Working with the NetBeans IDE

If you have chosen NetBeans IDE 6.5 as your development environment, you can use the following instructions to create a project for your first script: a simple calculator.

Step 1: Create a New Project

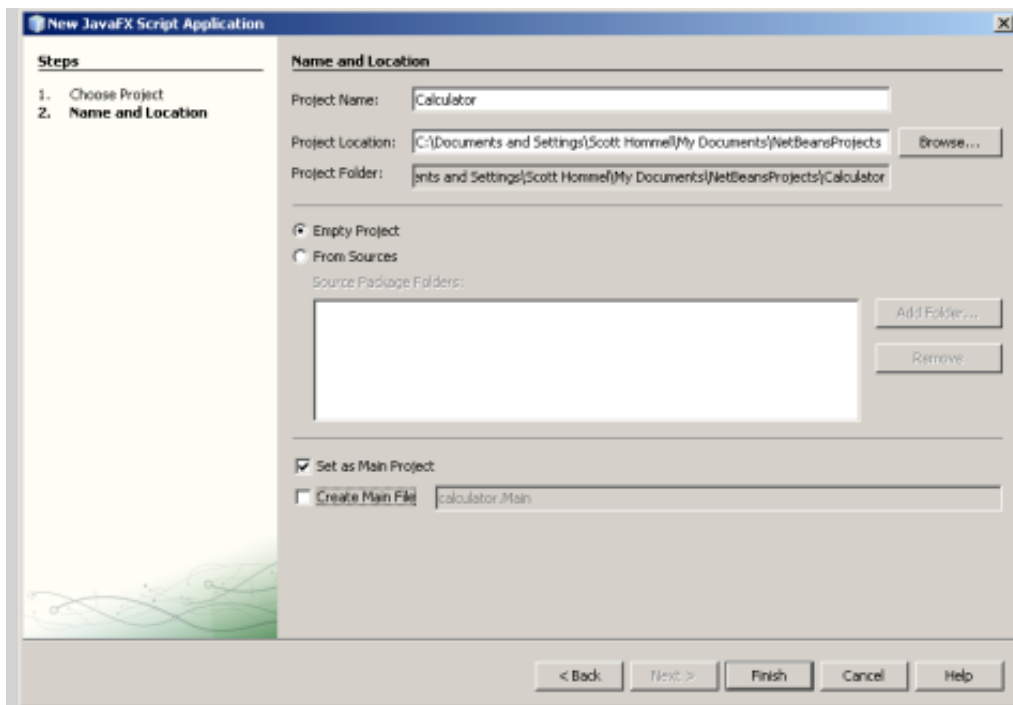
Launch the NetBeans IDE 6.5 and choose **File | New Project**.

When the new project wizard appears, choose **JavaFX** as the category and press **Next** to continue.



Step 2: Choose a Project Name and Location

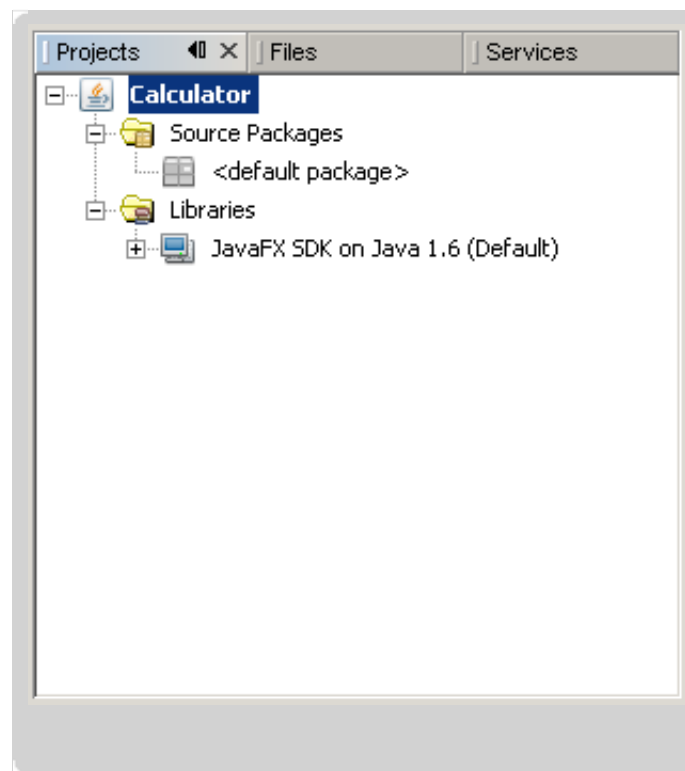
Next, type "Calculator" as the project name. The NetBeans IDE will provide a default location on your system for this project. You can keep this suggestion, or specify a new one. Make sure that "Empty Project" and "Set as Main Project" are selected, but do not check the "Create Main File" checkbox. Press the "Finish" button when done.



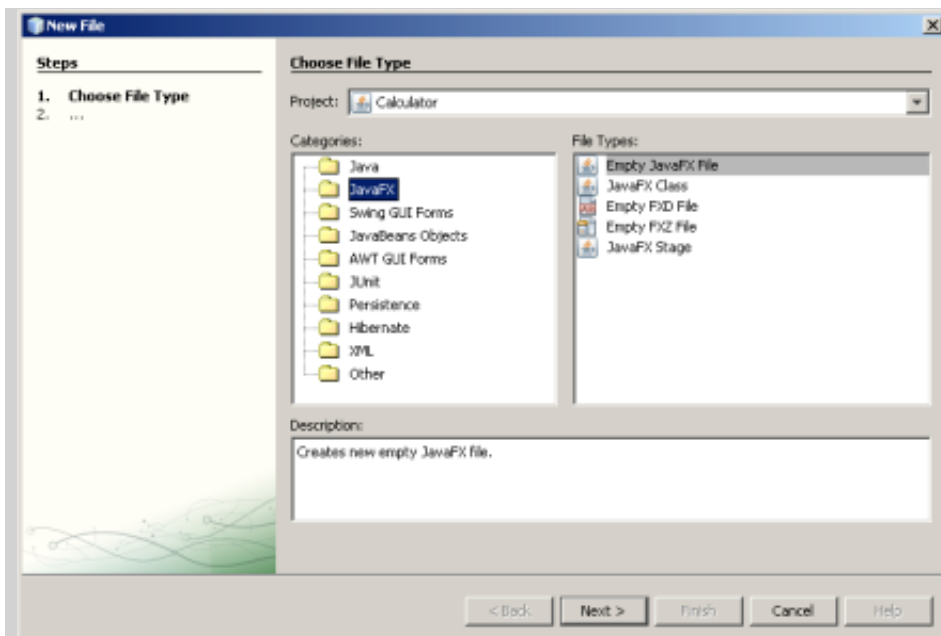
This image has been scaled down to fit your screen. [Click to enlarge.](#)

Step 3: Add a Source File to the Project

The left side of the IDE contains a file browser window as shown below. You can see that the Calculator project exists, but currently has no source files:

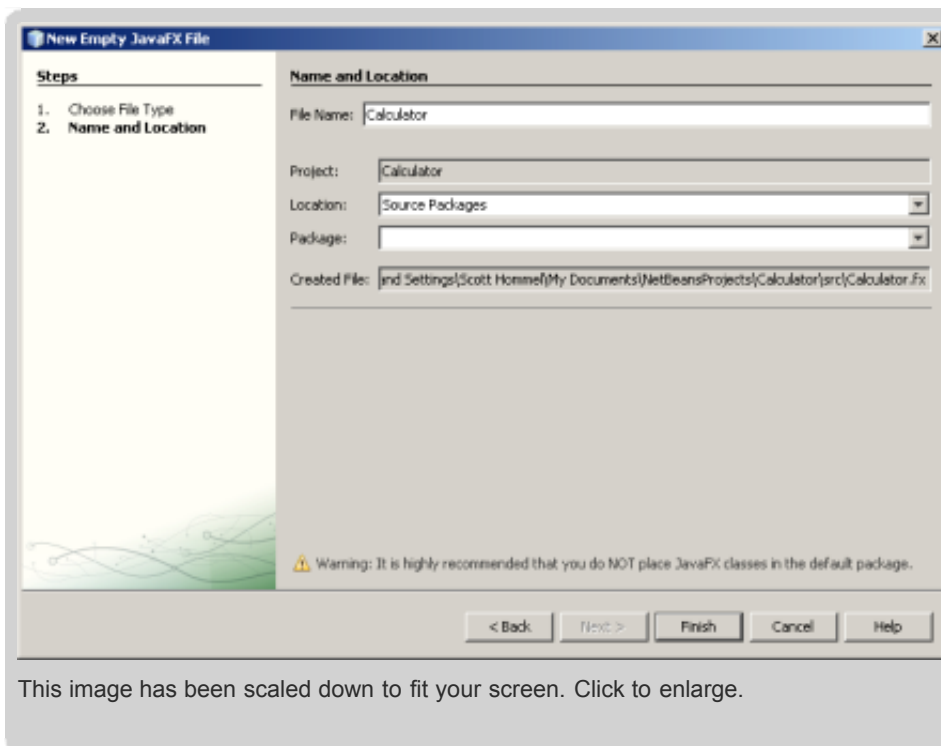


To add a source file to the project, choose **File | New File**. Select **JavaFX** as the category and **Empty JavaFX File** as the file type:



This image has been scaled down to fit your screen. [Click to enlarge.](#)

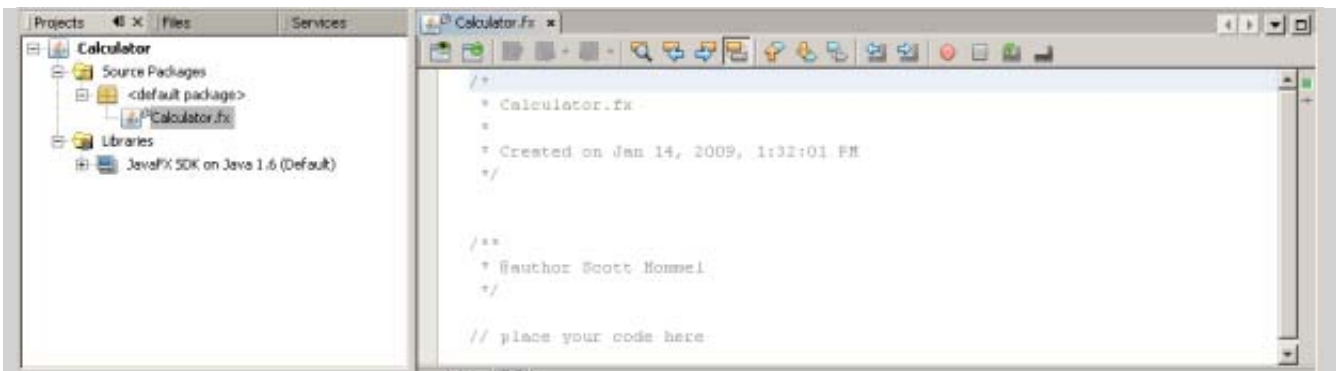
Next, type "Calculator" as the file name, but leave the package selection empty. You will see a package warning at the bottom of the screen, but ignore this for now; throughout most of this tutorial, we will be placing code into the default package. Press the "Finish" button when done.



This image has been scaled down to fit your screen. [Click to enlarge.](#)

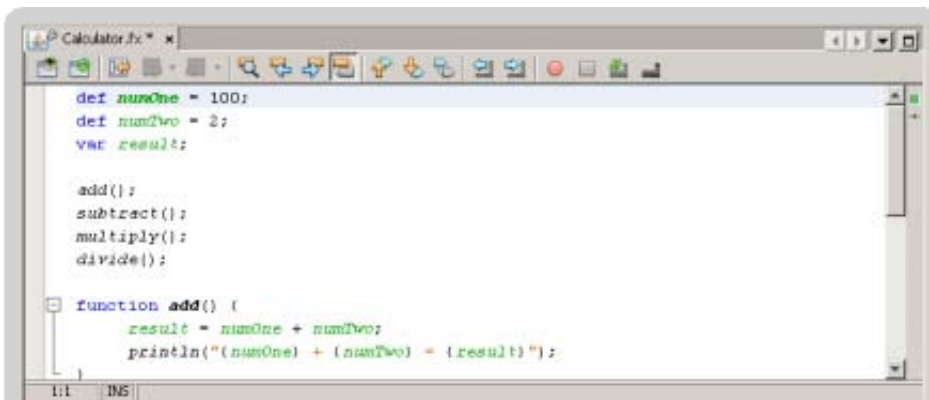
Step 4: Paste Source Code, Compile, and Run the Application!

The file browser now shows `Caclulator.fx` as part of the the default package. The source code editor (right hand pane) now contains some default code, which you can safely erase:



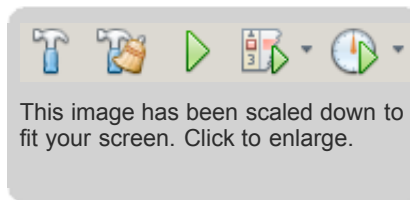
This image has been scaled down to fit your screen. Click to enlarge.

In its place, paste in the contents of [calculator.fx](#). Certain keywords are now highlighted, showing that the editor now recognizes the syntax of the JavaFX Script programming language:



This image has been scaled down to fit your screen. Click to enlarge.

Now look for these buttons along the top of the screen:



This image has been scaled down to fit your screen. Click to enlarge.

Press the green button in the center to compile and run the application:



This image has been scaled down to fit your screen. Click to enlarge.

You should see output similar to the above. If so, then congratulations, your project works!

Working from the Command Line

If you will be working from the command line only, save `calculator.fx` to a directory of your choice. Assuming that the JDK and JavaFX SDK are already installed and in your path, you should be able to compile this program with the following command:

```
javafx compiler calculator.fx
```

After compilation, you will find that the corresponding Java bytecode has been generated and placed into a file named `calculator.class`. You might also notice that another file, `calculator$Intf.class` has been created. This supporting file is needed to run the application — you can ignore it, but don't delete it.

You can now run the compiled class in the Java Virtual Machine with the following command:

```
javafx calculator
```

The output is:

```
100 + 2 = 102
100 - 2 = 98
100 * 2 = 200
100 / 2 = 50
```

This application may be small, but it introduces you to some important programming language constructs (discussed in the next lesson). Learning these constructs is your first step towards mastering the JavaFX Script programming language.

« [Previous](#) [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.



Learning the JavaFX Script Programming Language

Lesson 2: Writing Scripts

[Download tutorial](#)[« Previous](#) | [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next »](#)

This lesson provides a hands-on introduction to the JavaFX Script programming language. In it you will learn the basics of variables and functions, by writing a simple calculator that runs from the command line. Each section will introduce one new core concept, discuss it, and provide sample code that you can compile and run. The discussions also contain "real world" code excerpts showing how a particular construct is used in an actual SDK demo. Following the link for each demo will take you to the javafx.com website, where you can obtain the full code listing, plus additional notes from the developer.

Contents

- [Declaring Script Variables](#)
- [Defining and Invoking Script Functions](#)
- [Passing Arguments to Script Functions](#)
- [Returning Values from Script Functions](#)
- [Accessing Command-Line Arguments](#)

Declaring Script Variables

The previous lesson walked you through setting up a development environment; here we will take a closer look at the [calculator.fx](#) source code. The code in red below declares the program's *script variables*. Script variables are declared using the `var` or `def` keywords. The difference between the two is that `var` variables may be assigned new values throughout the life of the script, whereas `def` variables remain constant at their first assigned value. Here we have assigned some values to `numOne` and `numTwo`, but have left `result` uninitialized because this variable will hold the result of our future calculations:

```
def numOne = 100;
def numTwo = 2;
var result;

add();
subtract();
multiply();
divide();

function add() {
    result = numOne + numTwo;
    println("{numOne} + {numTwo} = {result}");
}

function subtract() {
    result = numOne - numTwo;
    println("{numOne} - {numTwo} = {result}");
}

function multiply() {
    result = numOne * numTwo;
    println("{numOne} * {numTwo} = {result}");
}

function divide() {
    result = numOne / numTwo;
    println("{numOne} / {numTwo} = {result}");
}
```

You may have noticed that we did not need to explicitly specify these variables as holding numerical data (as opposed to character strings or any other kind of data). The compiler is smart enough to figure out your intent based on the context in which the variable is used. This is known as *type inference*. Type inference makes your job as a script programmer a little easier because it frees you from the burden of declaring the data types that your variable is compatible with.

Real-World Example: [Effects Playground](#)



```

...
def width = (6 * (82 + 10)) + 20;

def canvasWidth = width-10;
def canvasHeight = 275;

var stage:Stage;

var inBrowser = "true".equals(FX.getArgument("isApplet") as String);
var dragTextVisible =
    bind inBrowser and AppletStageExtension.appletDragSupported;
var closeButtonVisible = bind (not inBrowser);

var selectedPreview:Preview = null on replace {
    if (selectedPreview == null) {
        hideControls();
    }
};

var fgUrl = "{_DIR_}images/flower.jpg";
var bgUrl = "{_DIR_}images/water.jpg";
...

```

The screenshot above shows the "Effects Playground" demo application. The code excerpt to the right shows a few of its script variables. You won't understand this entire listing — yet — but the highlighted portions should make sense based on what you've just learned. Keep in mind that while this tutorial is focused on non-graphical, core constructs only, you will eventually put this knowledge to use in your own GUI-based applications.

For a lower-level discussion of variables, see [Chapter 3: Variables](#) of the [JavaFX Script Programming Language Reference](#).

Defining and Invoking Script Functions

Our calculator example also defines some *script functions* that add, subtract, multiply, and divide the two numbers. A *function* is an executable block of code that performs a specific task. The red code below defines four functions; each performs a simple mathematical calculation and then prints out the result. Organizing code into functions is a common practice that makes your programs easier to read, easier to use, and easier to debug. The body of a function will typically be indented for readability.

```

def numOne = 100;
def numTwo = 2;
var result;

add();
subtract();
multiply();
divide();

function add() {
    result = numOne + numTwo;
    println("{numOne} + {numTwo} = {result}");
}

function subtract() {
    result = numOne - numTwo;
    println("{numOne} - {numTwo} = {result}");
}

function multiply() {
    result = numOne * numTwo;
    println("{numOne} * {numTwo} = {result}");
}

function divide() {
    result = numOne / numTwo;
    println("{numOne} / {numTwo} = {result}");
}

```

You should also know that function code does not execute until it is explicitly *invoked*. This makes it possible to run a function from any location within your script.

It does not matter if the function invocation is placed before or after the function definition (in our example we invoke the functions earlier in the source file than where they are actually defined):

```
def numOne = 100;
def numTwo = 2;
var result;

add();
subtract();
multiply();
divide();

function add() {
    result = numOne + numTwo;
    println("{numOne} + {numTwo} = {result}");
}

function subtract() {
    result = numOne - numTwo;
    println("{numOne} - {numTwo} = {result}");
}

function multiply() {
    result = numOne * numTwo;
    println("{numOne} * {numTwo} = {result}");
}

function divide() {
    result = numOne / numTwo;
    println("{numOne} / {numTwo} = {result}");
}
```

Real-World Example: [Draggable MP3 Player](#)



```
...
function stopCurrentSong():Void {
    mediaPlayer.stop();
    mediaPlayer.media = null;
    if (playlist.currentPlayingSong != null) {
        playlist.currentPlayingSong.closeMedia();
    }
}

function playCurrentSong():Void {
    playlist.currentPlayingSong = playlist.songs[playlist.currentSong];
    mediaPlayer.media = playlist.currentPlayingSong.getMedia();
    mediaPlayer.play();
}
...
```

In the "Draggable MP3 Player" demo, the programmer has defined functions to stop or play the current song. Although we are looking at this code completely out of context, the naming choice for these functions (`stopCurrentSong` and `playCurrentSong`) makes the code self-documenting, and therefore much easier to analyze. When naming your variables and functions, try to always use meaningful words like this. The convention is to spell the first word in all lowercase letters, then capitalize the first letter of each subsequent word in the name.

Passing Arguments to Script Functions

Script functions may also be defined to accept *arguments*. Arguments are specific values that you pass in while invoking a function. With this approach we can make the calculator application perform computations on any two numbers, not just the values that were hard-coded into the `numOne` and `numTwo` variables. In fact, in this version, we have removed `numOne` and `numTwo` entirely, leaving `result` as the only remaining script variable:

```
var result;

add(100,10);
subtract(50,5);
```

```

multiply(25,4);
divide(500,2);

function add(argOne: Integer, argTwo: Integer) {
    result = argOne + argTwo;
    println("{argOne} + {argTwo} = {result}");
}

function subtract(argOne: Integer, argTwo: Integer) {
    result = argOne - argTwo;
    println("{argOne} - {argTwo} = {result}");
}

function multiply(argOne: Integer, argTwo: Integer) {
    result = argOne * argTwo;
    println("{argOne} * {argTwo} = {result}");
}

function divide(argOne: Integer, argTwo: Integer) {
    result = argOne / argTwo;
    println("{argOne} / {argTwo} = {result}");
}

```

The output of this script is now:

```

100 + 10 = 110
50 - 5 = 45
25 * 4 = 100
500 / 2 = 250

```

Real-World Example: Interesting Photos



```

...
// Load image and data specified in given Photo object
function loadImage(photo: Photo, thumbImageView: ThumbImageView): Void {
    thumbImageView.image = Image {
        url: "http://farm{photo.farm}.static.flickr.com/"
            "{photo.server}/{photo.id}_{photo.secret}_{imgSuffix}.jpg";

        width: thumbSize
        height: thumbSize
        backgroundLoading: true
        placeholder: thumbImageView.image
    };
    thumbImageView.photo = photo;
}
...

```

In this code excerpt from the "Interesting Photos" demo, we see a script function named `loadImage` that accepts a list of arguments. Again, the choice of function and argument names makes the code easier to understand. Understanding the full implementation of this function is not important at this time. What is important, however, is recognizing the function that accepts two arguments. As you start writing your own applications, you will probably rely on sample code such as this for examples of the correct syntax.

Returning Values from Script Functions

A function may also *return* a value to the code that invokes it. For example, we could change the calculator's `add` function so that it returns the result of each calculation:

```
function add(argOne: Integer, argTwo: Integer) : Integer {
    result = argOne + argTwo;
    println("{argOne} + {argTwo} = {result}");
    return result;
}
```

The first code in red specifies that the function returns an `Integer`; the second is the code that actually returns the value.

The `add` function can now be invoked like this:

```
var total;

total = add(1,300) + add(23,52);
```

If no return value is specified, a function returns `Void` by default.

Real-World Example: Animating Photos from Flickr



```
...
public function magnitude(): Number {
    return Math.sqrt(x*x + y*y);
}

public function distance(v1: Vector2D, v2: Vector2D): Number {
    var dx = v1.x - v2.x;
    var dy = v1.y - v2.y;
    return Math.sqrt(dx*dx + dy*dy);
}

public function sub(v1: Vector2D, v2: Vector2D): Vector2D {
    var dx = v1.x - v2.x;
    var dy = v1.y - v2.y;
    return Vector2D {x: dx, y: dy };
}
...
```

In this code excerpt from the "Animating Photos from Flickr" demo, we see return values at work in three different functions. The returned values are slightly more complex than what you've seen so far, but the core concept is the same: each function performs some specific calculation, then returns a result. The first two functions subsequently invoke a `Math` function (to calculate a square root) and return its result. The third function returns a new `Vector2D` object. There isn't enough information in this listing alone to know exactly what that means, but with the complete source code, it would make sense (that is, providing that you've taken the time to first learn the language!)

For a lower-level discussion of functions, see [Chapter 4: Functions of the JavaFX Script Programming Language Reference](#).

Accessing Command-Line Arguments

Finally, scripts can also accept command-line arguments. In our calculator example, this will enable end users to specify the numbers to be calculated at runtime.

```
var result;

function run(args : String[]) {

    // Convert Strings to Integers
    def numOne = java.lang.Integer.parseInt(args[0]);
    def numTwo = java.lang.Integer.parseInt(args[1]);

    // Invoke Functions
    add(numOne,numTwo);
    subtract(numOne,numTwo);
    multiply(numOne,numTwo);
    divide(numOne,numTwo);
```

```

}

function add(argOne: Integer, argTwo: Integer) {
    result = argOne + argTwo;
    println("{argOne} + {argTwo} = {result}");
}

function subtract(argOne: Integer, argTwo: Integer) {
    result = argOne - argTwo;
    println("{argOne} - {argTwo} = {result}");
}

function multiply(argOne: Integer, argTwo: Integer) {
    result = argOne * argTwo;
    println("{argOne} * {argTwo} = {result}");
}

function divide(argOne: Integer, argTwo: Integer) {
    result = argOne / argTwo;
    println("{argOne} / {argTwo} = {result}");
}

```

This change introduces the `run` function, which is where the command-line arguments are received by the script. Unlike the other functions that you've seen, `run` is a special function that serves as the script's main entry point. The `run` function stores all command-line arguments in `args`, which is a *sequence* of `String` objects. (Sequences are ordered lists of objects, similar to arrays in other programming languages; they are explained in detail in [Lesson 5: Sequences](#).)

To run this script, the user now must specify the first and second numbers at runtime:

```
javafx calculator 100 50
```

The output is now:

```

100 + 50 = 150
100 - 50 = 50
100 * 50 = 5000
100 / 50 = 2

```

Note that in all previous versions of the calculator script, we did not explicitly provide a `run` function. We just typed the code to be executed at the script level and it ran as expected. In such cases, the compiler silently generates a no-arg `run` function and places the code to be executed inside of it. When specifying your own `run` function, the name `args` can be anything you want; we use `args` but you will probably see programmers use other variations of it, such as `arg`, `ARGS`, `__ARGS__`, `argv` etc.

Also note that in this version we have brought back the `numOne` and `numTwo` variables, which are now defined inside the `run` function instead of at the script level. (When a variable is defined inside a function, it is technically known as a *local variable*, because it is only visible to other code within that same function.) Our calculator functions expect numbers, but the command-line arguments are character strings; we therefore must convert each command-line argument from `String` to `Integer` before we can pass them along to the functions:

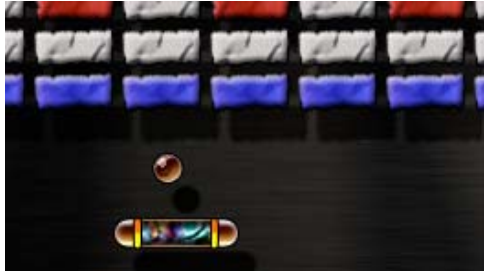
```

// Convert Strings to Integers
def numOne = java.lang.Integer.parseInt(args[0]);
def numTwo = java.lang.Integer.parseInt(args[1]);

```

To do this we have enlisted the help of the Java programming language to perform the actual type conversion. Tapping into the existing Java ecosystem as needed brings tremendous power to this otherwise simple scripting language.

Real-World Example: [Brick Breaker](#)



```
function run( _ARGS_ : String[] ) {  
    // Initialization should be the first  
    Config.initialize(IS_MOBILE);  
  
    mainFrame = MainFrame {  
        title: "Brick Breaker"  
        resizable: false  
  
        scene: Scene {  
            fill: Color.BLACK  
            width: Config.screenWidth  
            height: Config.screenHeight  
        }  
    }  
}
```

This excerpt from the "Brick Breaker" demo shows the run function serving as the game's main entry point. While this particular example does not actually use its command-line arguments, we can see that the run function initializes the main frame of the application, setting its title, width, height, etc.

« [Previous](#) | [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.

copyright © Sun Microsystems, Inc



Learning the JavaFX Script Programming Language

Lesson 3: Using Objects

[Download tutorial](#)[« Previous 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | Next »](#)

This lesson provides an introduction to using objects. It describes, at a high level, what an object is, then introduces you to object literals, instance variables, and instance functions. This lesson provides `Address` and `Customer` examples that you can compile, and concludes with a code excerpt showing how objects are used a real-world application.

Contents

- [What is an Object?](#)
- [Declaring an Object Literal](#)
- [Object Literal Syntax](#)
- [Invoking Instance Functions](#)

What is an Object?

The JavaFX Script programming language is an *object-oriented* language. But what does that mean? What exactly *is* an object? Simply put, an *object* is a discrete software bundle that consists of *state* and *behavior*. To better understand software objects, it helps to step outside of software for a minute and think about concepts that you're already are familiar with.

Your television set, for example, is an object. It has both state (current channel, current volume, on, off) and behavior (change channels, adjust volume, turn on, turn off). We tend to think of a television as a single object, but in reality, a television is actually *composed* of many other objects (the buttons and knobs on its face are all objects, as are the various components *inside* the television). In many cases, these smaller objects are also made up of... you guessed it... other objects.

We can break down a television's components until we reach a point where we can go no further (a screw, for example, really is just a single object, it's not composed of anything else). If you've ever bought a piece of exercise gear, or anything else with "some assembly required", you've probably seen an exploded diagram in its documentation showing every single object in its construction. At a glance you can understand how many objects there are, and how those objects all fit together. The same is true of `.fx` source files: you can easily see how a script's objects all fit together to form a full application.

Declaring an Object Literal

So what does a software object actually look like in terms of code? In the JavaFX Script programming language, an object is created with an *object literal*:

```
Address {
    street: "1 Main Street";
    city: "Santa Clara";
    state: "CA";
    zip: "95050";
}
```

This creates one `Address` object, for use in a hypothetical address book application. It is initialized with specific values for its `street`, `city`, `state`, and `zip`. In a real application, you can imagine that the address book's GUI components would be synchronized with its underlying `Address` objects, so that what appears on screen reflects the actual data that is stored by the program.

But before you try compiling this code, you need to know one more thing: the compiler first needs a special "blueprint" (called a *class*) that describes exactly *what* data an `Address` object will contain. (In this example, we are saying that an `Address` has a `street`, `city`, `state`, and `zip`, but the compiler won't know that until we supply a class file that provides that information.) We don't discuss writing your own

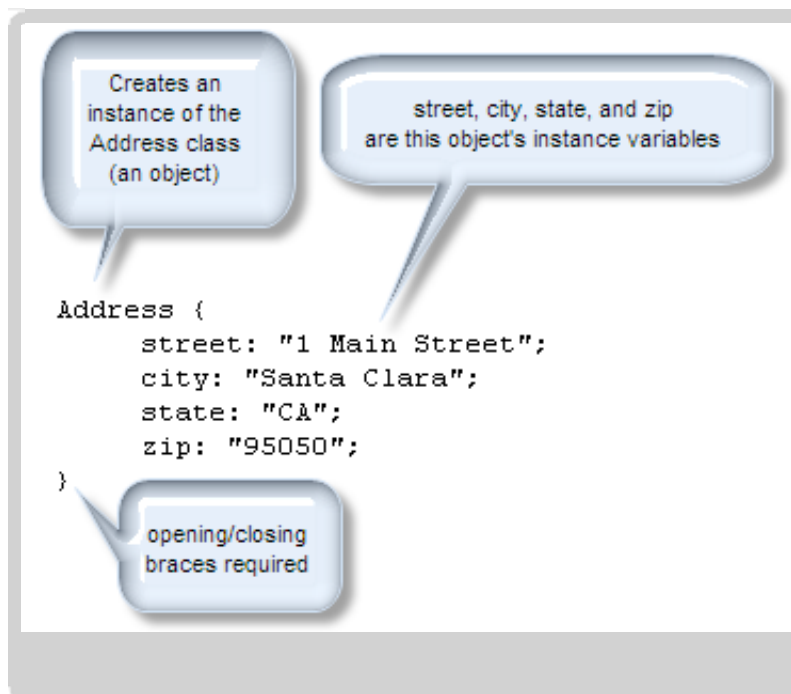
classes until the end of the tutorial, because the JavaFX Application Programming Interface (API) contains a large library of built-in classes that are ready to use in your own programs. Furthermore, since the JavaFX Script programming language is built on the Java platform, you can also access the Java programming language API as well.

Having said that, you can compile the `Address` example by downloading the file `Address.fx` (which provides the `Address` class definition) and placing it in the same directory as the sample code shown above. (Just be sure to save the sample code to a file with a different name, such as `AddressTest.fx`.) Nothing in this code produces any output, but the very fact that it compiles is proof that object creation was successful.

Note: The `Address` object's variables (`street`, `city`, `state`, `zip`) in this example are technically known as *instance variables*. You can think of instance variables as the set of built-in attributes that each object is guaranteed to contain. In fact, the term "attribute" was used in early versions of the JavaFX Script programming language (you may still occasionally see it used in older documentation and demos.) In the world of object-oriented programming, the terms "instance" and "object" are synonymous, which is where this term comes from.

Object Literal Syntax

Object literal syntax is both simple to learn and intuitive to use. Continuing with our example, the first word (`Address`) specifies the type of object that you are creating. The opening and closing braces define the body of the object. Everything else (`street`, `city`, `state`, `zip`) initializes the object's instance variables.



Note that when declaring an object literal, the instance variables may be separated by commas, whitespace, or semi-colons. You can use either, but we generally stick to semi-colons throughout this tutorial. For a more formal description of this syntax, refer to [Figure 6.38](#) in the [JavaFX Script Programming Language Reference](#).

You can also assign the newly created object to a variable for later access:

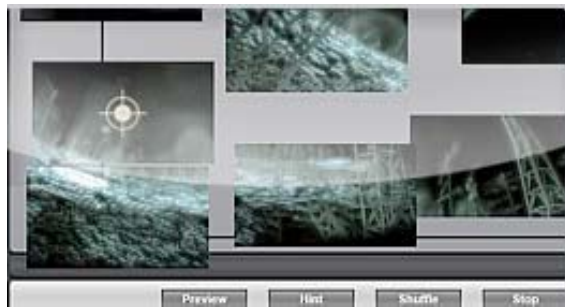
```
def myAddress = Address {
    street: "1 Main Street";
    city: "Santa Clara";
    state: "CA";
    zip: "95050";
}
```

Or nest one object inside another:

```
def customer = Customer {
    firstName: "John";
    lastName: "Doe";
    phoneNum: "(408) 555-1212";
    address: Address {
        street: "1 Main Street";
        city: "Santa Clara";
        state: "CA";
        zip: "95050";
    }
}
```

In this last example, the `Customer` object introduces a few new variables, then contains the original `Address` object in a variable named `address`. Notice how we've indented the code as the objects begin to nest. Real-world applications will typically contain lots of nested objects; this nested pattern makes it easy to see at a glance what objects belong where. To compile this example, you will need to add `Customer.fx` to your current directory.

Real-World Example: Video Puzzle



```
...
var previewSpotX = 12; var previewSpotY = 50;
var overlaySpotX = 95; var overlaySpotY = 55;
var previewDimOverlay = Group {
    visible: false
    content:[
        Rectangle {
            x: 7 y: 25 width: 685 height: 332
            arcWidth: 10 arcHeight: 10
            fill: Color.rgb(0,0,0,0.4)
            blocksMouse: true
        }
    ]
}
...
```

The video puzzle demo takes a streaming video feed and chops it up into a puzzle. Here we have highlighted a section of its code that uses object literals. You should be able to recognize the syntax, even if you are not familiar with the given classes. In this code excerpt, the `visible` and `content` instance variables belong to the `Group` object. The `x`, `y`, `width`, `height`, `arcWidth`, `arcHeight`, `fill`, and `blockMouse` instance variables belong to the `Rectangle` object. The `Group` object has been stored in a variable called `previewDimOverlay` for later access.

Invoking Instance Functions

In addition to instance variables, objects may contain *instance functions* as well. You can invoke an object's instance functions by typing the name of the variable (`customer` in the following example), followed by a dot ("`.`"), followed by the function that you want to invoke:

```
def customer = Customer {
    firstName: "John";
    lastName: "Doe";
    phoneNum: "(408) 555-1212"
    address: Address {
        street: "1 Main Street";
```

```
        city: "Santa Clara";
        state: "CA";
        zip: "95050";
    }
}

customer.printName();
customer.printPhoneNum();
customer.printAddress();
```

This results in the following output:

```
Name: John Doe
Phone: (408) 555-1212
Street: 1 Main Street
City: Santa Clara
State: CA
Zip: 95050
```

Instance functions can also be defined to accept parameters and return values, as you learned in the previous lesson.

« [Previous](#) [1](#) | [2](#) | **[3](#)** | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.

copyright © Sun Microsystems, Inc



Learning the JavaFX Script Programming Language

Lesson 4: Data Types

[Download tutorial](#)[« Previous 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | Next »](#)

By now you've learned about variables, functions, and objects. This lesson explores the built-in *data types*. The JavaFX Script programming language supports character string types, numeric types, boolean (true/false) types. It also supports time-based (duration) types, plus special types that represent void and null.

This lessons focuses on the basic information that most script writers will want to know. For a lower-level discussion of the type system, see [Chapter 2. Types and Values of the JavaFX Script Programming Language Reference](#).

Contents

- String
- Number and Integer
- Boolean
- Duration
- Void
- Null

String

You have seen many `String` examples already, but let's take a closer look at this data type. A `String` can be declared using either single or double quotes:

```
var s1 = 'Hello';
var s2 = "Hello";
```

Single and double quotes are symmetrical: you can embed single quotes within double quotes, and double quotes within single quotes. There is no difference between single- and double-quoted strings.

You can also embed [expressions](#) inside a string using curly braces "`{}`":

```
def name = 'Joe';
var s = "Hello {name}"; // s = 'Hello Joe'
```

The embedded expression can itself contain quoted strings, which, in turn, can contain further embedded expressions:

```
def answer = true;
var s = "The answer is {if (answer) "Yes" else "No"}"; // s = 'The answer is Yes'
```

At runtime, the compiler will substitute the bold expression above with the string "Yes" if the value of `answer` is true or "No" otherwise.

To join (concatenate) multiple strings, use curly braces inside the quotes:

```
def one = "This example ";
def two = "joins two strings.";
def three = "{one}{two}";      // join string one and string two
println(three);                // 'This example joins two strings.'
```

Number and Integer

The `Number` and `Integer` types represent numerical data, although for most scripting tasks, you will usually just let the compiler infer the correct type:

```
def numOne = 1.0; // compiler will infer Number
def numTwo = 1;   // compiler will infer Integer
```

You can, however, explicitly declare a variable's type:

```
def numOne : Number = 1.0;
def numTwo : Integer = 1;
```

The difference between these two types is that `Number` represents floating-point numbers, but `Integer` only represents integers. Use `Number` only when you absolutely need floating-point precision. In all other cases, `Integer` should be your first choice.

Note: As of SDK 1.1, the language also contains numeric types that align with those found in the Java programming language. The full list of numeric types is therefore: `Byte`, `Short`, `Number`, `Integer`, `Long`, `Float`, `Double`, and `Character`. However, the advice given above still holds true: most programmers will only need `Integer` (or `Number`) for the scripts that they write. If you are coming to this language from a Java programming language background and have a task that absolutely requires one of the other types, just know that these additional types are now available to your scripts.

Boolean

The `Boolean` type represents two values: true or false. Use this type when setting some application-specific internal state:

```
var isAsleep = true;
```

or when evaluating a conditional expression:

```
if (isAsleep) {
    wakeUp();
}
```

If the expression inside the braces, "`()`", is true, the code inside the curly braces, "`{ }`" is executed. For more information on conditional expressions, please see the [Expressions](#) lesson.

Duration

The `Duration` type represents a fixed unit of time (millisecond, second, minute, or hour.)

```
5ms; // 5 milliseconds
10s; // 10 seconds
30m; // 30 minutes
1h; // 1 hour
```

Durations are notated with *time literals* — for example, `5m` is a time literal representing five minutes. Time literals are most often used in animation (which you will learn about in the [Creating Animated Objects](#) lesson, part of the [Building GUI Applications with JavaFX](#)).

Void

`Void` is used to indicate that a function does not return any value:

```
function printMe() : Void {
    println("I don't return anything!");
}
```

This is equivalent to the following, which omits the function's return type:

```
function printMe() {
    println("I don't return anything!");
}
```

The JavaFX keyword `Void` begins with a capital **V**. If you are familiar with `void` in the Java Programming Language, you should take note of this.

Note: In JavaFX, everything is an expression. The return type of the second `printMe` function is also `Void`, as the compiler was able to infer its type. You will learn more about this in the [Expressions](#) lesson.

Null

`Null` is a special value used to indicate a missing normal value. `Null` is not the same as zero or an empty string, so comparing for `null` is not the same as comparing for zero or an empty string.

The `null` keyword allows comparisons to be made. It is common to see `null` used like this:

```
function checkArg(arg1: Address) {
    if(arg1 == null) {
        println("I received a null argument.");
    } else {
        println("The argument has a value.");
    }
}
```

This function accepts one argument, then performs a simple test to check if its value is `null`.

« [Previous](#) [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.

copyright © Sun Microsystems, Inc



Learning the JavaFX Script Programming Language

Lesson 5: Sequences

[Download tutorial](#)[« Previous](#) | [1](#) | [2](#) | [3](#) | [4](#) | **[5](#)** | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next »](#)

In addition to the basic data types, the JavaFX Script programming language also provides special data structures called *sequences*. Sequences represent ordered lists of objects (although sequences themselves are *not* objects). The objects within a sequence are called *items*. Sequences are declared with square brackets `[]`, and each item is separated by a comma. This lesson covers all the basics of creating and using sequences. It also discusses how to work with sequence *slices* (portions of sequences). For additional information, see [Sequence Types](#) in the [JavaFX Script Programming Language Reference](#).

Contents

- [Creating Sequences](#)
- [Using Predicates](#)
- [Accessing Sequence Items](#)
- [Inserting Items into a Sequence](#)
- [Deleting Items from a Sequence](#)
- [Reversing the Items in a Sequence](#)
- [Comparing Sequences](#)
- [Using Sequences Slices](#)

Creating Sequences

One way to create a sequence is to explicitly list its items. Each item is separated by a comma and the list is enclosed in square brackets `[]` and `]`. For example, the following code:

```
def weekDays = ["Mon", "Tue", "Wed", "Thu", "Fri"];
```

declares a sequence and assigns it to `weekDays`. We use `def` in this example because we do not plan on changing its value after the sequence has been created. Here the compiler knows that we intend to create a "sequence of strings" because the individual items are all declared as `String` literals. If the sequence had been declared with `Integers` (for example, `def nums = [1, 2, 3];`), the compiler would know that we want a "sequence of integers" instead.

You can also explicitly specify a sequence's type by modifying its declaration to include the name of the type followed by `[]`:

```
def weekDays: String[] = ["Mon", "Tue", "Wed", "Thu", "Fri"];
```

This tells the compiler that `weekDays` will hold a sequence of `Strings` (as opposed to a single `String`).

Sequences can also be declared within other sequences:

```
def days = [weekDays, ["Sat", "Sun"]];
```

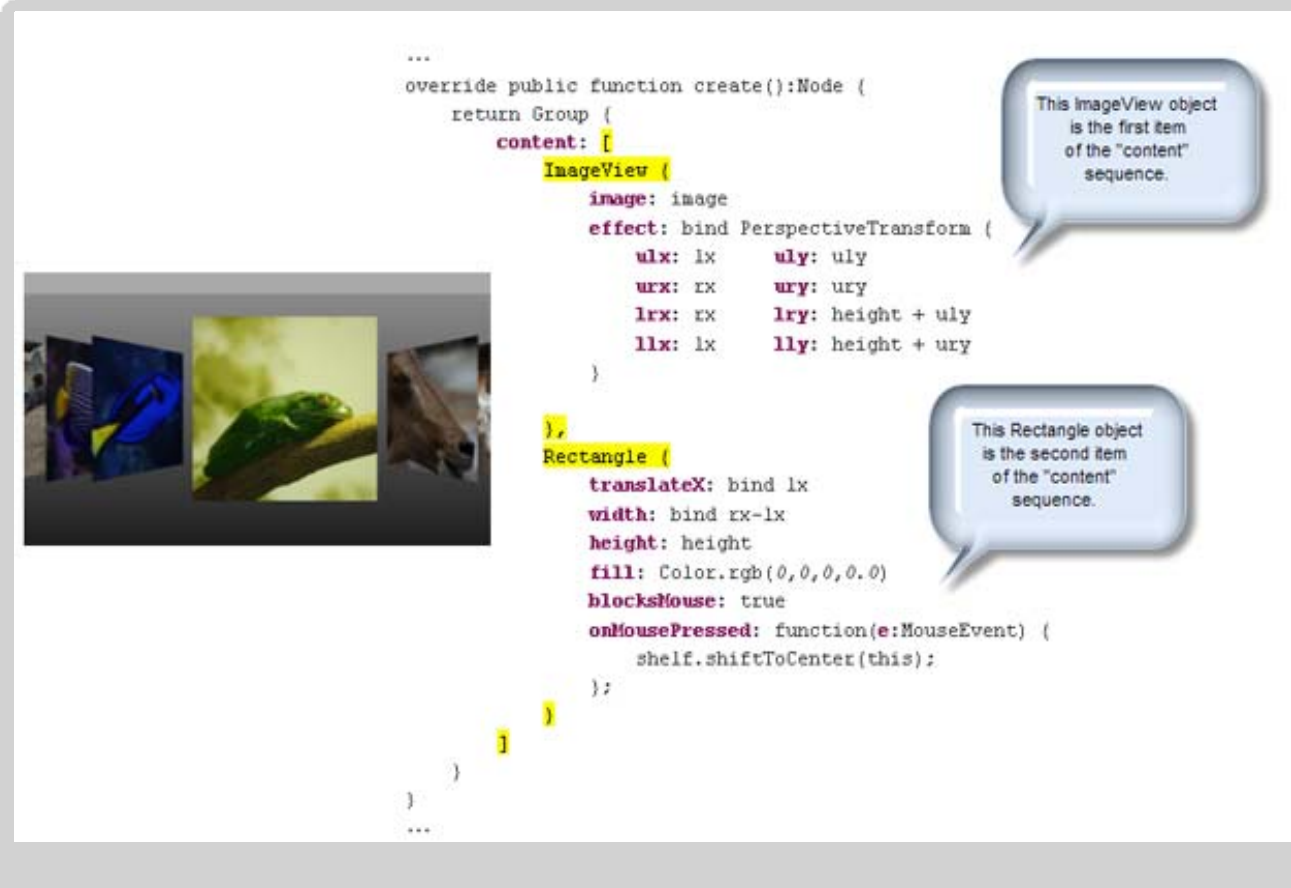
In such cases, the compiler will automatically flatten the nested sequences to form a single sequence, making the above equivalent to:

```
def days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
```

There is also a shorthand notation that makes it easier to create sequences that form an arithmetic series. To create a sequence consisting of the numbers 1 through 100, use the following:

```
def nums = [1..100];
```

Real-World Example: *Display Shelf* (Sequence Example #1)



```

...
override public function create():Node {
    return Group {
        content: [
            ImageView {
                image: image
                effect: bind PerspectiveTransform {
                    ulx: lx    uly: uly
                    urx: rx    ury: ury
                    lrx: rx    lry: height + uly
                    llx: lx    lly: height + ury
                }
            },
            Rectangle {
                translateX: bind lx
                width: bind rx-lx
                height: height
                fill: Color.rgb(0,0,0,0.0)
                blocksMouse: true
                onMousePressed: function(e:MouseEvent) {
                    shelf.shiftToCenter(this);
                }
            }
        ]
    }
}
...

```

This ImageView object is the first item of the "content" sequence.

This Rectangle object is the second item of the "content" sequence.

This screenshot shows the "Display Shelf" demo application. The code excerpt to the right shows an instance variable (named "content") belonging to the Group object. This variable holds a sequence that has been initialized with two items (the highlighted ImageView and Rectangle objects). You can spot the presence of a sequence by looking for its opening and closing square brackets (also highlighted). Sequence items are delimited by commas, which is why "," appears between the ImageView and Rectangle objects.

Using Predicates

You can use a boolean expression, (also called a *predicate*), to declare a new sequence that is a subset of an existing sequence. For example, consider the following:

```
def nums = [1,2,3,4,5];
```

To create a second sequence (based on items found in this first sequence) but containing only numbers greater than 2, use the following:

```
def numsGreaterThanTwo = nums[n | n > 2];
```

You could express the previous line of code in English like this: "select all items from the `num` sequence where **the value of an item is greater than 2** and assign those items to a new sequence called `numsGreaterThanTwo`." The "where" clause, highlighted in bold, is the predicate.

In this code:

1. The newly created sequence is stored in `numsGreaterThanTwo`.
2. The code (marked in bold): `nums[n | n > 2];` specifies the original sequence that the items are to be copied from. In our example, `nums` is the name of the original sequence.
3. This selects the items from `nums`, and returns a new sequence consisting of those items, in order, for which the expression is true.
4. The `|` character is used to visually separate the variable "n" from the rest of the code: `nums[n | n > 2];`
5. The code (marked in bold): `nums[n | n > 2];` defines a boolean *expression* specifying the criteria to be met before the current item will be copied into the new sequence.

Accessing Sequence Items

Sequence items are accessed by numerical index, starting at 0. To access an individual item, type the sequence name, followed by the item's numerical index enclosed within square brackets:

```
def days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];

println(days[0]);
println(days[1]);
println(days[2]);
println(days[3]);
println(days[4]);
println(days[5]);
println(days[6]);
```

This prints the following to the screen:

```
Mon
Tue
Wed
Thu
Fri
Sat
Sun
```

Also, you can determine the size of a sequence using the `sizeof` operator followed by the name of the sequence:

```
sizeof days
```

The following code prints "7" to the screen:

```
def days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];  
println(sizeof days);
```

Inserting Items into a Sequence

The `insert` keyword allows you to insert an item *into* a sequence, *before* a specific item, or *after* a specific item.

Note: Technically speaking, sequences are *immutable*, meaning that once created they never change. When you modify a sequence by inserting or deleting items, for example, behind the scenes a new sequence is created and the sequence variable is reassigned, giving the impression that the sequence was modified.

Let's explore this by re-creating the `days` sequence. Note that we now declare the `days` variable with `var`, because we will be changing its value after the original sequence has been created:

```
var days = ["Mon"];
```

At this point, the sequence contains only one item, "Mon".

We can insert "Tue" into the end of this sequence using the `insert` and `into` keywords:

```
insert "Tue" into days;
```

Similarly, we could add "Fri", "Sat" and "Sun" as well:

```
insert "Fri" into days;  
insert "Sat" into days;  
insert "Sun" into days;
```

The sequence now contains: "Mon", "Tue", "Fri", "Sat", and "Sun".

We can also use the `insert` and `before` keywords to insert an item before an item at a given index. Keep in mind that indexing begins at 0, so in our current sequence, "Fri" sits at index position 2. Therefore, we can insert "Thu" before "Fri" as follows:

```
insert "Thu" before days[2];
```

The sequence now contains: "Mon", "Tue", "Thu", "Fri", "Sat", and "Sun".

To insert "Wed" after "Tue", we can use the `insert` and `after` keywords:

```
insert "wed" after days[1];
```

The sequence now contains all the days of the week: "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", and "Sun".

Deleting Items from a Sequence

The `delete` and `from` keywords make it easy to delete items from a sequence:

```
delete "Sun" from days;
```

The sequence now contains: "Mon", "Tue", "Wed", "Thu", "Fri", and "Sat".

You can also delete an item from a specific index position. The following code deletes "Mon" from the sequence (remember that "Mon" is the first item, so its index position is 0.)

```
delete days[0];
```

To delete all items from a sequence, use the `delete` keyword followed by the name of the sequence:

```
delete days;
```

Note that `delete` only removes the items from the sequence; it does not delete the `days` variable from your script. You can still access the `days` variable and add new items to it as before.

Reversing the Items in a Sequence

You can easily reverse the items in a sequence using the `reverse` operator:

```
var nums = [1..5];  
reverse nums; // returns [5, 4, 3, 2, 1]
```

Comparing Sequences

There may be times when you will want to compare sequences for equality. Sequences are compared for equality by value: if their lengths are equal, and their items are equal, then they are equal.

Let's test this out by creating two sequences with identical contents:

```
def seq1 = [1,2,3,4,5];
```

```
def seq2 = [1,2,3,4,5];
println(seq1 == seq2);
```

The expression `seq1 == seq2` evaluates to `true` because both sequences have the same number of items, and the value of each item is the same in both sequences. This code therefore prints "true" to the screen.

By changing the number of items in one sequence (but not the other), the sequences now become different lengths:

```
def seq1 = [1,2,3,4,5];
def seq2 = [1,2,3,4,5,6];
println(seq1 == seq2);
```

Here the output of this script is "false" because the second sequence is longer than the first, thereby making them unequal.

We can also make the two sequences unequal by changing the item values, even if the sequences are still the same length:

```
def seq1 = [1,2,3,4,5];
def seq2 = [1,3,2,4,5];
println(seq1 == seq2);
```

Again, this code will print "false" because the two sequences are not equal.

Using Sequence Slices

Sequence *slices* provide access to portions of a sequence.

seq[a..b]

This syntax provides access to the items between index *a* and index *b*, inclusive. The following script creates a weekend sequence consisting of only the items "Sat" and "Sun".

```
def days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
def weekend = days[5..6];
```

seq[a.<b]

Use the "<" character to access the items between index *a* inclusive and index *b* *exclusive*. We could use this on `days` to create a weekdays sequence consisting of the items "Mon" through "Fri".

```
def days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
def weekdays = days[0..<5];
```

seq[a..]

By omitting the second index, you can access all items from index *a* through the end of the sequence. In keeping with the same example,

we could create the weekend sequence as follows:

```
def days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
def weekend = days[5..];
```

seq[a..<]

Finally, you can use "<" without a second index to access everything from index *a* to the end of the sequence, *excluding* the last item.

```
def days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
def days2 = days[0..<];
```

This version creates a `days2` sequence consisting of the items "Mon" through "Sat".

Real-World Example: *Display Shelf* (Sequence Example #2)

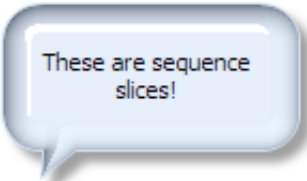
```
...
override public function create():Node {
    var half = content.size()/2-1;

    left.content = content[0..half-2];
    center.content = content[half-1];
    right.content = content[half..content.size()-1];
    right.content = Sequences.<<reverse>>(right.content) as Node[];

    centerIndex = half-1;

    doLayout();
    return Group {
        content: [
            left,
            right,
            center
        ]
    }
}
...

```



This code listing from the "Display Shelf" demo application shows the use of two sequence slices. The slices are declared with ranges of "0 .. half-2" and "half..content.size()-1", respectively. To identify a sequence slice, look for ".." inside the square brackets.

« [Previous](#) [1](#) | [2](#) | [3](#) | [4](#) | **[5](#)** | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.



Learning the JavaFX Script Programming Language

Lesson 6: Operators

[Download tutorial](#)[« Previous 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | Next »](#)

Operators are special symbols that perform specific operations on one or two *operands*, and then return a result. The JavaFX Script programming language provides assignment operators, arithmetic operators, unary operators, equality and relational operators, conditional operators, and a type comparison operator. For a formal description of the supported operators, see [Tables 6.2 - 6.8](#) in the [JavaFX Script Programming Language Reference](#).

Contents

- Assignment Operators
- Arithmetic Operators
- Unary Operators
- Equality and Relational Operators
- Conditional Operators
- Type Comparison Operator

Assignment Operators

The *assignment operator* "=" is one of the most commonly used operators that you will encounter. Use it to assign the value on its right to the operand on its left:

```
result = num1 + num2;
days = ["Mon", "Tue", "Wed", "Thu", "Fri"];
```

You have already used this operator in many of the previous lessons.

Arithmetic Operators

The *arithmetic operators* make it possible to perform addition, subtraction, multiplication, and division. The `mod` operator divides one operand by another and returns the remainder as its result.

```
+ (additive operator)
- (subtraction operator)
* (multiplication operator)
/ (division operator)
mod (remainder operator)
```

The following script provides some examples:

```
var result = 1 + 2; // result is now 3
println(result);

result = result - 1; // result is now 2
println(result);
```

```
result = result * 2; // result is now 4
println(result);

result = result / 2; // result is now 2
println(result);

result = result + 8; // result is now 10
println(result);

result = result mod 7; // result is now 3
println(result);
```

You can also combine the arithmetic operators with the assignment operator to create *compound assignments*. For example, `result += 1`; and `result = result+1`; both increment the value of `result` by 1.

```
var result = 0;
result += 1;
println(result); // result is now 1

result -= 1;
println(result); // result is now 0

result = 2;
result *= 5; // result is now 10
println(result);

result /= 2; // result is now 5
println(result);
```

The only arithmetic operator that cannot be used in this manner is `mod`. If, for example, you want to divide `result` by 2, then assign the remainder back to itself, you would need to write: `result = result mod 2`;

Unary Operators

Most operators require two operands; the *unary operators* use just one operand and perform operations such as incrementing/decrementing a value by one, negating a number, or inverting the value of a boolean.

| | |
|-----|---|
| - | Unary minus operator; negates a number |
| ++ | Increment operator; increments a value by 1 |
| -- | Decrement operator; decrements a value by 1 |
| not | Logical complement operator; inverts the value of a boolean |

The following script tests the unary operators:

```
var result = 1; // result is now 1

result--; // result is now 0
println(result);

result++; // result is now 1
```

```
println(result);

result = -result; // result is now -1
println(result);

var success = false;
println(success); // false
println(not success); // true
```

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code `result++`; and `++result`; will both end in `result` being incremented by one. The only difference is that the prefix version (`++result`) evaluates to the incremented value, whereas the postfix version (`result++`) evaluates to the original value. (You can remember which is which by `++result` does the increment and then gets the value, while `result++` gets the value and then does the increment.) If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

The following script illustrates this distinction:

```
var result = 3;
result++;
println(result); // result is now 4
++result;
println(result); // result is now 5
println(++result); // result is now 6
println(result++); // this still prints 6!
println(result); // but the result is now 7
```

Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand.

```
==      equal to
!=      not equal to
>       greater than
>=     greater than or equal to
<       less than
<=     less than or equal to
```

The following script tests these operators:

```
def num1 = 1;
def num2 = 2;

println(num1 == num2); // prints false
println(num1 != num2); // prints true
println(num1 > num2); // prints false
println(num1 >= num2); // prints false
println(num1 < num2); // prints true
println(num1 <= num2); // prints true
```

Conditional Operators

The *conditional and* and *conditional or* operators perform conditional operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed: For example, for an `and` operation, if the result of the first expression is `false`, the second expression will not be evaluated. For an `or` operation, if the result of the first expression is `true` the second expression will not be evaluated.

```
and  
or
```

The following script demonstrates the use of these operators by defining `username` and `password` variables, then printing out matches of various conditions:

```
def username = "foo";  
def password = "bar";  
  
if ((username == "foo") and (password == "bar")) {  
    println("Test 1: username AND password are correct");  
}  
  
if ((username == "") and (password == "bar")) {  
    println("Test 2: username AND password is correct");  
}  
  
if ((username == "foo") or (password == "bar")) {  
    println("Test 3: username OR password is correct");  
}  
  
if ((username == "") or (password == "bar")) {  
    println("Test 4: username OR password is correct");  
}
```

The output is:

```
Test 1: username AND password are correct  
Test 3: username OR password is correct  
Test 4: username OR password is correct
```

Type Comparison Operator

The `instanceof` operator compares an object to a specified type. You can use it to determine if an object is an instance of a particular class:

```
def str1="Hello";  
println(str1 instanceof String); // prints true  
  
def num = 1031;  
println(num instanceof java.lang.Integer); // prints true
```

You will find this operator to be useful as you learn more about classes and inheritance in the last lesson of this tutorial.

« [Previous](#) [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | **[6](#)** | [7](#) | [8](#) | [9](#) | [10](#) | [11](#) | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.

Excellent **Good** **Fair** **Poor**

Comments:

Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.

Submit »

copyright © Sun Microsystems, Inc



Learning the JavaFX Script Programming Language

Lesson 7: Expressions

Print-friendly Version Download tutorial

« [Previous](#) [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | **[7](#)** | [8](#) | [9](#) | [10](#) | [11](#) | [Next](#) »

Expressions are pieces of code that evaluate to a result value, and that can be combined to produce "bigger" expressions. The JavaFX Script programming language is an expression language, which means that everything, including loops, conditionals, and even blocks, are expressions. In some cases (such as `while` expressions) the expressions have `Void` type, which means they don't return a result value. For a lower-level discussion of expressions, see [Chapter 6. Expressions in the JavaFX Script Programming Language Reference](#).

Contents

- [Block Expressions](#)
- [The if Expression](#)
- [Range Expressions](#)
- [The for Expression](#)
- [The while Expression](#)
- [The break and continue Expressions](#)
- [The throw, try, catch and finally Expressions](#)

Block Expressions

A *block expression* consists of a list of declarations or expressions surrounded by curly braces and separated by semicolons. The value of a block expression is the value of the last expression. If the block expression contains no expressions, the block expression has `Void` type. Note that `var` and `def` are expressions.

The following block expression adds a few numbers and stores the result in a variable named `total`:

```
var nums = [5, 7, 3, 9];
var total = {
    var sum = 0;
    for (a in nums) { sum += a };
    sum;
}
println("Total is {total}.");
```

Running this script produces the following output:

```
Total is 24.
```

The first line (`var nums = [5, 7, 3, 9];`) declares a sequence of integers.

The next line declares a variable named `total` that will hold the sum of these integers.

The block expression that follows consists of everything between the curly braces:

```
{
var sum = 0;
  for (a in nums) { sum += a };
  sum;
}
```

Within this block, the first line of code declares a variable named `sum`, to hold the sum of the numbers in this sequence. The second line (a `for expression`) loops through the sequence and adds each number to `sum`. The last line sets the return value of the block expression (24, in this case).

The if Expression

The `if` expression makes it possible to direct the flow of a program by executing certain blocks of code only *if* a particular condition is true.

For example, the following script sets a ticket price based on age. Ages 12 to 65 pay regular the price of \$10. Seniors and children pay \$5; except for children under 5 who are admitted for free.

```
def age = 8;
var ticketPrice;

if (age < 5 ) {
  ticketPrice = 0;
} else if (age < 12 or age > 65) {
  ticketPrice = 5;
} else {
  ticketPrice = 10;
}
println("Age: {age} Ticket Price: {ticketPrice} dollars.");
```

With `age` set to 8, the script produces the following output:

```
Age: 8 Ticket Price: 5 dollars.
```

The program flows through this example as follows:

```
if (age < 5 ) {
  ticketPrice = 0;
} else if (age < 12 or age > 65) {
  ticketPrice = 5;
} else {
  ticketPrice = 10;
}
```

If `age` is less than 5, the ticket price is set to 0.

The program then jumps past the remaining conditional tests and prints out the result.

If `age` is not less than 5, the programs proceeds to the next conditional test (indicated by the `else` keyword followed by another `if`

expression):

```
if (age < 5 ) {
    ticketPrice = 0;
} else if (age < 12 or age > 65) {
    ticketPrice = 5;
} else {
    ticketPrice = 10;
}
```

This sets the ticket price to \$5 if the person's age is between 5 and 12 or over 65.

If the age is between 12 and 65, the program flows to the final block of code, marked with the `else` keyword:

```
if (age < 5 ) {
    ticketPrice = 0;
} else if (age < 12 or age > 65) {
    ticketPrice = 5;
} else {
    ticketPrice = 10;
}
```

This block executes only if none of the previous conditions are satisfied. It sets the ticket price to \$10 for ages between 12 and 65.

Note: The previous code can be collapsed to a very concise conditional expression:

```
ticketPrice = if (age < 5) 0 else if (age < 12 or age > 65) 5 else 10;
```

This is a useful technique to master and you will see it used again later in the tutorial.

Range Expressions

The Sequences lesson taught you a shorthand notation for declaring a sequence of numbers that form an arithmetic series:

```
var num = [0..5];
```

Technically speaking, `[0..5]` is a *range expression*. By default the interval between the values is 1, but you can use the `step` keyword to specify a different interval. For example, to define a sequence consisting of the odd numbers between 1 and 10:

```
var nums = [1..10 step 2];
println(nums);
```

The output of this script is:

```
[ 1, 3, 5, 7, 9 ]
```

To create a *descending range*, make sure the second value is less than the first, and specify a negative step value:

```
var nums = [10..1 step -1];  
println(nums);
```

The output is:

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

If you do not provide a *negative* step value while creating a descending range, you will end up with an empty sequence.

The following code:

```
var nums = [10..1 step 1];  
println(nums);
```

results in the following compile-time warning:

```
range.fx:1: warning: empty sequence range literal, probably not what you meant.  
var nums = [10..1 step 1];  
           ^  
1 warning
```

You would also end up with an empty sequence if you omit the step value altogether.

The for Expression

Another sequence-related expression is the *for* expression. The *for* expression provides a convenient mechanism for looping through the items of a sequence.

The following code provides an example:

```
var days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];  
  
for (day in days) {  
    println(day);  
}
```

The output of this script is:

```
Mon
Tue
Wed
Thu
Fri
Sat
Sun
```

Let's break this example down into its individual parts. The "for" keyword begins the `for` expression:

```
for (day in days) {
    println(day);
}
```

The `days` variable is the name of the *input sequence* to be processed by the `for` expression:

```
for (day in days) {
    println(day);
}
```

The `day` variable holds the current item as the `for` expression loops through the sequence:

```
for (day in days) {
    println(day);
}
```

Note that the `day` variable does not need to be declared elsewhere in the script before using it in the `for` expression. Furthermore, `day` cannot be accessed after the loop has run its course. Programmers will often give temporary variables such as this very short (or one-letter) names.

In the previous example, `for` was not shown returning a value; however `for` is also an expression that returns a sequence. The following code shows two examples of creating a sequence from another sequence, using the `for` expression:

```
// Resulting sequence squares the values from the original sequence.
var squares = for (i in [1..10]) i*i;

// Resulting sequence is ["MON", "TUE", "WED", and so on...]
var capitalDays = for (day in days) day.toUpperCase();
```

Note that the `toUpperCase` function is provided by the `String` object. You can see a full list of available functions by consulting the API documentation.

The while Expression

Another looping construct is the *while* expression. Unlike the *for* expression, which operates on the items of a sequence, the *while* expression loops until a given condition is *false*. While *while* is syntactically an expression, it has type *Void*, and doesn't return a value.

The following provides an example:

```
var count = 0;
while (count < 10) {
    println("count == {count}");
    count++;
}
```

The output of this script is:

```
count == 0
count == 1
count == 2
count == 3
count == 4
count == 5
count == 6
count == 7
count == 8
count == 9
```

The first line declares a variable named *count* and initializes it to 0:

```
var count = 0;
while (count < 10) {
    println("count == {count}");
    count += 1;
}
```

The next line begins the *while* expression. This expression creates a loop (between the opening and closing braces) that lasts until *count < 10* evaluates to *false*:

```
var count = 0;
while (count < 10) {
    println("count == {count}");
    count += 1;
}
```

The body of the *while* expression prints out the current value of *count*, then increments its value of *count* by one:

```
var count = 0;
while (count < 10) {
    println("count == {count}");
    count += 1;
}
```

When `count` becomes equal to 10, the loop will exit. To create an infinite loop, place the `true` keyword between the parenthesis, as in: `while(true){}`

The `break` and `continue` Expressions

Related to the looping expressions are the `break` and `continue` expressions. These two expressions affect loop iteration: `break` abandons the loop entirely, whereas `continue` abandons only the current iteration.

While `break` and `continue` are syntactically expressions, they have type `Void` and don't return a value.

Example:

```
for (i in [0..10]) {
    if (i > 5) {
        break;
    }

    if (i mod 2 == 0) {
        continue;
    }

    println(i);
}
```

Output:

```
1
3
5
```

Without the `if` expressions, the program would simply output the numbers 0 through 10.

With only the first `if` expression, the program would *break* out of the loop when the value of `i` becomes greater than 5:

```
if (i > 5) {
    break;
}
```

The program would therefore print only the numbers 1 through 5.

By adding the second `if` expression, the program abandons the loop's current iteration only to *continue* with the next iteration:

```
if (i mod 2 == 0) {
    continue;
}
```

In this case `continue` only executes when `i` is even (that is, when 2 divides evenly into `i`, leaving no remainder.) When this happens, the `println()` invocation is never reached, and the number is not included in the output.

The throw, try, catch and finally Expressions

In real-world applications, there will be times when some event will disrupt the normal flow of a script's execution. For example, if a script reads input from a file, and that file cannot be found, the script will not be able to proceed. We call this condition an "exception."

Note: Exceptions are objects. Their types generally are named after the conditions they represent (for example, `FileNotFoundException` represents the condition of a file that cannot be found.) However, defining a set of exceptions specific to the upcoming examples is beyond the scope of this section. We will therefore use a general-purpose `Exception` object (borrowed from the Java programming language) to demonstrate the `throw`, `try`, `catch`, and `finally` expressions.

The following script defines (and invokes) a function that throws an exception:

```
import java.lang.Exception;

foo();

println("The script is now executing as expected... ");

function foo() {
    var somethingWeird = false;

    if(somethingWeird){
        throw new Exception("Something weird just happened!");
    } else {
        println("We made it through the function.");
    }
}
```

Running this script as-is (with `somethingWeird` set to `false`) prints out the following message:

```
We made it through the function.
The script is now executing as expected...
```

But change that variable to `true`, and an exception will be thrown. At runtime, the script will crash with the following message:

```
Exception in thread "main" java.lang.Exception: Something weird just happened!
at exceptions.foo(exceptions.fx:10)
at exceptions.javafx$run$(exceptions.fx:3)
```

To prevent this crash, we would need to wrap the `foo()` invocation with `try/catch` expressions. As their names imply, these expressions *try* to execute some code, but *catch* an exception if there is a problem:

```
try {
    foo();
} catch (e: Exception) {
    println("{e.getMessage()} (but we caught it)");
}
```

Now, instead of crashing, the program simply prints:

```
Something weird just happened! (but we caught it)
The script is now executing as expected...
```

There is also a *finally* block (it's not technically an expression), which always executes at some point after the `try` expressions exits, regardless of whether an exception was thrown or not. The *finally* block is used to perform cleanup that needs to occur regardless of whether the `try`-body succeeds or throws an exception.

```
try {
    foo();
} catch (e: Exception) {
    println("{e.getMessage()} (but we caught it)");
} finally {
    println("We are now in the finally expression...");
}
```

The program output is now:

```
Something weird just happened! (but we caught it)
We are now in the finally expression...
The script is now executing as expected...
```

« [Previous](#) 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

Your email address (no reply is possible without an address):



Learning the JavaFX Script Programming Language

Lesson 8: Data Binding and Triggers

[Download tutorial](#)[« Previous](#) | [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | **[8](#)** | [9](#) | [10](#) | [11](#) | [Next »](#)

Data binding, or the ability to create an immediate and direct relationship between two variables, is one of the most powerful features of the JavaFX Script programming language. This lesson begins with binding two simple variables and progresses to more sophisticated binding: between a variable and the outcome of a function or an expression. Once you understand the concept, [Applying Data Binding to UI Objects](#), a lesson in [Building GUI Applications with JavaFX](#), gives an example of how data binding can be a powerful tool when building JavaFX applications.

A replace trigger is a block of code that is attached to a variable — when the variable changes, the code is automatically executed. A practical application of a replace trigger is presented.

For more information on data binding, see [Chapter 7. Binding](#) in the [JavaFX Script Programming Language Reference](#).

Contents

- [Binding Overview](#)
- [Binding and Objects](#)
- [Binding and Functions](#)
- [Binding with Sequences](#)
- [Replace Triggers](#)

Binding Overview

The `bind` keyword associates the value of a target variable with the value of a bound expression. The bound expression can be a simple value of some basic type, an object, the outcome of a function, or the outcome of an expression. The following sections present examples of each.

Binding and Objects

In most real-world programming situations, you will use data binding to keep an application's Graphical User Interface (GUI) in sync with its underlying data. (GUI programming is the topic of the [Building GUI Applications with JavaFX](#); below we demonstrate the basic underlying mechanics with some non-visual examples.)

Let's start simple: the following script binds variable `x` to variable `y`, changes the value of `x`, then prints out the value of `y`. Because the variables are bound, the value of `y` automatically updates to the new value.

```
var x = 0;
def y = bind x;
x = 1;
println(y); // y now equals 1
x = 47;
println(y); // y now equals 47
```

Note that we have declared variable `y` as a `def`. This prevents any code from directly assigning a value to it (yet its value is allowed to change as the result of a `bind`). You should use this same convention when binding to an object (recall that we introduced `Address` in [Using Objects](#)):

```
var myStreet = "1 Main Street";
```

```

var myCity = "Santa Clara";
var myState = "CA";
var myZip = "95050";

def address = bind Address {
    street: myStreet;
    city: myCity;
    state: myState;
    zip: myZip;
};

println("address.street == {address.street}");
myStreet = "100 Maple Street";
println("address.street == {address.street}");

```

By changing the value of `myStreet`, the `street` variable inside the `address` object is affected:

```

address.street == 1 Main Street
address.street == 100 Maple Street

```

Note that changing the value of `myStreet` actually causes a new `Address` object to be created and then re-assigned to the `address` variable. To track changes *without* creating a new `Address` object, `bind` directly to the object's instance variables instead:

```

def address = bind Address {
    street: bind myStreet;
    city: bind myCity;
    state: bind myState;
    zip: bind myZip;
};

```

You can also omit the first `bind` (the one just before `Address`) if you are explicitly binding to the instance variables:

```

def address = Address {
    street: bind myStreet;
    city: bind myCity;
    state: bind myState;
    zip: bind myZip;
};

```

Binding and Functions

The previous lessons taught you about functions, but there is another distinction that you must also learn: *bound functions* vs. *non-bound functions*.

Consider the following function, which creates and returns a `Point` object:

```

var scale = 1.0;

```

```
bound function makePoint(xPos : Number, yPos : Number) : Point {
    Point {
        x: xPos * scale
        y: yPos * scale
    }
}

class Point {
    var x : Number;
    var y : Number;
}
```

This is known as a *bound* function, because it is preceded with the `bound` keyword.

Note: the `bound` keyword does not replace the `bind` keyword; the two are used in conjunction as described below.

Next let's add some code to invoke this function and test out the binding:

```
var scale = 1.0;

bound function makePoint(xPos : Number, yPos : Number) : Point {
    Point {
        x: xPos * scale
        y: yPos * scale
    }
}

class Point {
    var x : Number;
    var y : Number;
}

var myX = 3.0;
var myY = 3.0;
def pt = bind makePoint(myX, myY);
println(pt.x);

myX = 10.0;
println(pt.x);

scale = 2.0;
println(pt.x);
```

The output of this script is:

```
3.0
10.0
20.0
```

Let's analyze this script one section at a time.

The code:

```
var myX = 3.0;
var myY = 3.0;
def pt = bind makePoint(myX, myY);
println(pt.x);
```

initializes the script variables `myX` and `myY` to `3.0`. These values are then passed as arguments to the `makePoint` function, which creates and returns a new `Point` object. The `bind` keyword, placed just before the invocation of `makePoint`, binds the newly created `Point` object (`pt`) to the outcome of the `makePoint` function.

Next, the code:

```
myX = 10.0;
println(pt.x);
```

changes the value of `myX` to `10.0` and prints out the value of `pt.x`. The output shows that `pt.x` is also now `10.0`.

Lastly, the code:

```
scale = 2.0;
println(pt.x);
```

changes the value of `scale` and again prints out the value of `pt.x`. The value of `pt.x` is now `20.0`. However, if we remove the `bound` keyword from this function (thereby making it a non-bound function), the output would be:

```
3.0
10.0
10.0
```

This is because non-bound functions only get re-invoked when one of their arguments change. Since `scale` is not a function argument, changing its value does not result in another function invocation.

Binding with Sequences

You can also use `bind` with `for` expressions. To explore this, let's first define two sequences and print out the values of their items:

```
var seq1 = [1..10];
def seq2 = bind for (item in seq1) item*2;
printSeqs();

function printSeqs() {
    println("First Sequence:");
    for (i in seq1){println(i);}
    println("Second Sequence:");
}
```

```
    for (i in seq2){println(i);}
}
```

`seq1` contains ten items (the numbers 1 through 10). `seq2` also contains ten items; these items would have the same values as `seq1`, but we have applied the expression `item*2` to each, so their values are doubled.

The output is therefore:

```
First Sequence:
1
2
3
4
5
6
7
8
9
10
Second Sequence:
2
4
6
8
10
12
14
16
18
20
```

We can bind the two sequences by placing the `bind` keyword just before the `for` keyword.

```
def seq2 = bind for (item in seq1) item*2;
```

The question now becomes: "if `seq1` changes in some way, will *all* of the items — or just *some* of the items — in `seq2` be affected?" We can test this out by inserting an item (the value 11) into the end of `seq1`, then printing the values of both sequences to see what changed:

```
var seq1 = [1..10];
def seq2 = bind for (item in seq1) item*2;
insert 11 into seq1;
printSeqs();

function printSeqs() {
    println("First Sequence:");
    for (i in seq1){println(i);}
    println("Second Sequence:");
    for (i in seq2){println(i);}
}
```

Output:

```
First Sequence:
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11
```

```
Second Sequence:
```

```
2  
4  
6  
8  
10  
12  
14  
16  
18  
20  
22
```

The output shows that inserting 11 into the end of `seq1` does not affect the first 10 items in `seq2`; a new item is automatically added to the end of `seq2`, with the value of 22.

Replace Triggers

Replace Triggers are arbitrary blocks of code that attach to variables and execute whenever the variable's value changes. The following example shows the basic syntax: it defines a `password` variable and attaches a replace trigger to it; when the password changes, the trigger prints out a message reporting its new value:

```
var password = "foo" on replace oldValue {  
    println("\nALERT! Password has changed!");  
    println("Old Value: {oldValue}");  
    println("New Value: {password}");  
};  
  
password = "bar";
```

The output of this example is:

```
ALERT! Password has changed!  
Old Value:  
New Value: foo
```

ALERT! Password has changed!

Old Value: foo

New Value: bar

The trigger in this example fires two times: first, when `password` is initialized to "foo", and then again when its value becomes "bar". Note that the `oldValue` variable stores the value of variable before the trigger was invoked. You can name this variable anything you want; we just happen to use `oldValue` because it is a descriptive name.

« [Previous](#) [1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | **[8](#)** | [9](#) | [10](#) | [11](#) | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.

copyright © Sun Microsystems, Inc



Learning the JavaFX Script Programming Language

Lesson 9: Writing Your Own Classes

[Download tutorial](#)[« Previous 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | Next »](#)

The JavaFX Script programming language API contains a number of classes ready for immediate use in your applications. However, at some point, you will probably find that it makes sense to write some application-specific classes of your own. This lesson provides a high-level introduction to what that entails. For a lower-level discussion, see [Chapter 5. Classes](#) in the [JavaFX Script Programming Language Reference](#).

Contents

- [The Customer Example](#)
- [Inheriting from Other Classes](#)

The Customer Example

In [Using Objects](#), you learned how to *use* objects. Recall, however, that we provided the necessary `Address.fx` and `Customer.fx` files. These source files contained *class definitions*, but we didn't discuss their contents at all. This section re-visits that example, this time placing everything you'll need into the same file:

```
def customer = Customer {
    firstName: "John";
    lastName: "Doe";
    phoneNum: "(408) 555-1212"
    address: Address {
        street: "1 Main Street";
        city: "Santa Clara";
        state: "CA";
        zip: "95050";
    }
}

customer.printName();
customer.printPhoneNum();
customer.printAddress();

class Address {
    var street: String;
    var city: String;
    var state: String;
    var zip: String;
}

class Customer {
    var firstName: String;
    var lastName: String;
    var phoneNum: String;
    var address: Address;

    function printName() {
        println("Name: {firstName} {lastName}");
    }
}
```

```
function printPhoneNum(){
    println("Phone: {phoneNum}");
}

function printAddress(){
    println("Street: {address.street}");
    println("City: {address.city}");
    println("State: {address.state}");
    println("Zip: {address.zip}");
}
}
```

This example should look familiar based on what you already know about variables and functions. The `Address` class declares `street`, `city`, `state`, and `zip` instance variables all of type `String`. The `Customer` class declares a few instance variables as well, plus functions to print out their values. Because these variables and functions are declared within classes, they will be available to any `Address` and `Customer` objects that you create.

Inheriting from Other Classes

You can also write classes that *inherit* variables and functions from other classes. For example, imagine a savings and checking account at a bank. Each has an account number and a balance. You can query the balance, make deposits, or make withdrawals. We can model this by creating a base `Account` class that provides the most common variables and functions:

```
abstract class Account {

    var accountNum: Integer;
    var balance: Number;

    function getBalance(): Number {
        return balance;
    }

    function deposit(amount: Number): Void {
        balance += amount;
    }

    function withdraw(amount: Number): Void {
        balance -= amount;
    }
}
```

We have marked this class as *abstract*, meaning that `Account` objects cannot be created directly (the intent of this design is that you may only create savings accounts or checking accounts.)

The `accountNum` and `balance` variables hold the account number and current balance. The remaining functions provide the basic behavior for getting the balance, making a deposit, or making a withdrawal.

We can then define a `SavingsAccount` class that uses the `extends` keyword to inherit these variables and functions:

```
class SavingsAccount extends Account {

    var minBalance = 100.00;
    var penalty = 5.00;
}
```

```

function checkMinBalance() : Void {
    if(balance < minBalance){
        balance -= penalty;
    }
}
}
}

```

Because `SavingsAccount` is a subclass of `Account`, it will automatically contain all of `Account`'s instance variables and functions. This leaves the `SavingsAccount` source code free to focus on how it *differs* from that of its parent (a requirement that the savings account holder must maintain a minimum balance of \$100 to avoid a penalty.)

We can similarly define a `CheckingAccount` class that also extends `Account`:

```

class CheckingAccount extends Account {

    var hasOverDraftProtection: Boolean;

    override function withdraw(amount: Number) : Void {
        if(balance-amount<0 and hasOverDraftProtection){

            // code to borrow money from an overdraft account would go here

        } else {
            balance -= amount; // may result in negative account balance!
        }
    }
}
}

```

This differs from `Account` by defining a variable that tracks whether or not the account holder has overdraft protection (if a withdrawal — such as writing a check — would result in the balance becoming less than zero, overdraft protection would activate to ensure that the check does not bounce). Note that in this case we are changing the behavior of the inherited `withdraw` function. This is known as *function overriding*, as indicated by the `override` keyword.

« [Previous 1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | **[9](#)** | [10](#) | [11](#) | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.



Learning the JavaFX Script Programming Language

Lesson 10: Packages

[Download tutorial](#)[« Previous 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | Next »](#)

Packages allow you to organize and structure your classes and their relationships to one another. This lesson walks you through how to set up and use a package.

Contents

- [Step 1: Choose a Package Name](#)
- [Step 2: Create the Directory](#)
- [Step 3: Add the Package Declaration](#)
- [Step 4: Add the Access Modifiers](#)
- [Step 5: Compile the Source](#)
- [Step 6: Use the Class](#)

By this point, you have a solid grasp of JavaFX Script programming language basics. However, the location of source files is likely to be a little disorganized (by now you probably have a single directory filled with lots of unrelated examples). We can improve our overall organization by placing our code into *packages*.

Packages make it possible to group your code according to functionality. They also give your classes a unique namespace. We'll explore this below with a step-by-step example that places the `Address` class into a specific package.

Step 1: Choose a Package Name

Before we modify any code, we must first choose a name for the package that we'll be creating. Since our `Address` class is intended to be used in a (hypothetical) addressbook application, we will use "addressbook" as the package name.

Step 2: Create the Directory

Next, we must create an `addressbook` directory on the filesystem itself. This directory will contain the `.fx` source files for any classes that we designate as belonging to the `addressbook` package. Feel free to create this directory anywhere you want; we will use `/home/demo/addressbook` in this example, but the scripts must be in a directory matching the name of the package, in this case `addressbook`.

Step 3: Add the Package Declaration

Now, go to the `addressbook` directory and create the `Address.fx` source file. Paste in the following source code. The first line provides a *package declaration*, which states that this class belongs to the `addressbook` package:

```
package addressbook;

class Address {
    var street: String;
    var city: String;
    var state: String;
    var zip: String;
}
```

Note that when a package declaration is present, it must appear by itself as the first line of code in the source file. Only one package

declaration per source file is allowed.

Step 4: Add the Access Modifiers

Next, we must add the `public` keyword to the `Address` class and its variables:

```
package addressbook;

public class Address {
    public var street: String;
    public var city: String;
    public var state: String;
    public var zip: String;
}
```

This keyword is one of five available *access modifiers*. We'll explore the access modifiers in the [next lesson](#). For now, just know that `public` makes this code accessible to other classes and scripts.

Step 5: Compile the Source

While still in the `addressbook` directory, compile this source as usual with the `javafx Address.fx` command. (In larger software projects, there are more sophisticated ways to build code from multiple packages, but compiling the source in this directory will work fine for this example.) After compilation, this directory will contain the resulting `.class` files.

Step 6: Use the Class

We are now ready to test the modified `Address` class. But first we will change back to the parent directory `/home/demo`. Here we will create a simple script, `packagetest.fx`, that tests the use of the `addressbook` package.

There are a couple of approaches that we can take to access this class:

```
// Approach #1

addressbook.Address {
    street: "1 Main Street";
    city: "Santa Clara";
    state: "CA";
    zip: "95050";
}
```

Approach #1 creates an object using the fully qualified class name (which is now `addressbook.Address`). Compared to other methods, this approach will probably feel too cumbersome (especially in large scripts), but still, you should know that it exists.

```
// Approach #2

import addressbook.Address;

Address {
    street: "1 Main Street";
    city: "Santa Clara";
    state: "CA";
    zip: "95050";
}
```

Approach #2 uses the *import* keyword, which allows the short name (*Address*) to now be used anywhere within the script. This approach is recommended for larger programs because it is self-documenting. At a glance, you can tell what package each class belongs to.

```
// Approach #3

import addressbook.*;

Address {
    street: "1 Main Street";
    city: "Santa Clara";
    state: "CA";
    zip: "95050";
}
```

Approach #3 uses the wildcard "*" to import all of the public classes defined in the *addressbook* package. In this example there's not much advantage to doing this, but if a package contains a large number of classes, you can use this technique to import them all at once. Some programmers find this convenient; the downside is that with this approach, it becomes more difficult to tell what package a given class lives in, especially when multiple packages are involved.

« [Previous 1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | **10** | [11](#) | [Next](#) »

Rate and Review

Tell us what you think of the content of this page.

Excellent Good Fair Poor

Comments:

Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.

copyright © Sun Microsystems, Inc



Learning the JavaFX Script Programming Language

Lesson 11: Access Modifiers

[Download tutorial](#)[« Previous 1](#) | [2](#) | [3](#) | [4](#) | [5](#) | [6](#) | [7](#) | [8](#) | [9](#) | [10](#) | **[11](#)** | [Next »](#)

Now that you understand packages, we can discuss the various *access modifiers* provided by the JavaFX Script programming language. These special keywords allow you to set various levels of visibility for your variables, functions, and classes. For more information, see [Chapter 8. Access Modifiers](#) in the [JavaFX Script Programming Language Reference](#).

Contents

- [Default Access](#)
- [The package Access Modifier](#)
- [The protected Access Modifier](#)
- [The public Access Modifier](#)
- [The public-read Access Modifier](#)
- [The public-init Access Modifier](#)

Default Access

The default access, known as "script-only", is what you get when no access modifier is provided. This is what we've been using throughout most of the tutorial.

Some examples are:

```
var x;  
var x : String;  
var x = z + 22;  
var x = bind f(q);
```

With this level of access, variables can be initialized, overridden, read, assigned, or bound from within the script only. No other source files can read or access this information.

The package Access Modifier

To make a variable, function, or class accessible to other code in the same package, use the `package` access modifier:

```
package var x;
```

Be careful not to confuse this access modifier with package declarations as described in the previous lesson!

Example:

```
// Inside file tutorial/one.fx  
package tutorial; // places this script in the "tutorial" package  
package var message = "Hello from one.fx!"; // this is the "package" access modifier
```

```
package function printMessage() {
    println("{message} (in function printMessage)");
}

// Inside file tutorial/two.fx
package tutorial;
println(one.message);
one.printMessage();
```

You can compile and run this example (from the `tutorial` parent directory) with the following commands.

```
javafx tutorial/one.fx tutorial/two.fx
javafx tutorial/two
```

The output is:

```
Hello from one.fx!
Hello from one.fx! (in function printMessage)
```

The **protected** Access Modifier

The `protected` access modifier makes a variable or function accessible to other code in the same package, and to subclasses that are in any package.

Example:

```
// Inside file tutorial/one.fx
package tutorial;
public class one {
    protected var message = "Hello!";
}

// Inside file two.fx
import tutorial.one;
class two extends one {
    function printMessage() {
        println("Class two says {message}");
    }
};

var t = two{};
t.printMessage();
```

Compile and run this demo:

```
javafx tutorial/one.fx two.fx
javafx two
```

The output is:

```
Class two says Hello!
```

Note: this access modifier cannot be applied to classes, which is why we have marked class one as `public`.

The `public` Access Modifier

A `public` class, variable, or function has the most visibility. It can be accessed from any class or script, in any package.

Example:

```
// Inside file tutorial/one.fx
package tutorial;
public def someMessage = "This is a public script variable, in one.fx";
public class one {
    public var message = "Hello from class one!";
    public function printMessage() {
        println("{message} (in function printMessage)");
    }
}

// Inside file two.fx
import tutorial.one;
println(one.someMessage);
var o = one{};
println(o.message);
o.printMessage();
```

Compile and run this example:

```
javafx tutorial/one.fx two.fx
javafx two
```

Output:

```
This is a public script variable, in one.fx
Hello from class one!
Hello from class one! (in function printMessage)
```

The `public-read` Access Modifier

The `public-read` access modifier defines a variable that is publicly readable, but (by default) is writeable only from within the current script. To widen its write access, prepend either the `package` or `protected` modifier (i.e. `package public-read` or `protected public-read`). Doing so will set its write access to the `package` or `protected` level.

Example:

```
// Inside file tutorial/one.fx
package tutorial;
public-read var x = 1;

// Inside tutorial/two.fx
package tutorial;
println(one.x);
```

Compile and run this example:

```
javafx tutorial/one.fx tutorial/two.fx
javafx tutorial/two
```

The output is "1", which proves that `x` can be read from outside the `tutorial/one.fx` script.

Now, let's attempt to modify its value:

```
// Inside tutorial/two.fx
package tutorial;
one.x = 2;
println(one.x);
```

The result is a compile-time error:

```
tutorial/two.fx:3: x has script only (default) write access in tutorial.one
one.x = 2;
  ^
1 error
```

For this to work, we must widen the write access of `x`:

```
// Inside file tutorial/one.fx
package tutorial;
package public-read var x = 1;

// Inside tutorial/two.fx
package tutorial;
one.x = 2;
println(one.x);
```

The example will now compile and print "2" to the screen.

The `public-init` Access Modifier

The `public-init` access modifier defines a variable that can be *publicly initialized* by object literals in any package. Subsequent write access, however, is controlled in the same manner as `public-read` (the default is write access at the script-level, but prepending `package` or `protected` will widen the access accordingly). The value of this variable is always readable from any package.

Example:

```
// Inside file tutorial/one.fx
package tutorial;
public class one {
    public-init var message;
}

// Inside file two.fx
import tutorial.one;
var o = one {
    message: "Initialized this variable from a different package!"
}
println(o.message);
```

Compile and run the example:

```
javafx tutorial/one.fx two.fx
javafx two
```

This prints "Initialized this variable from a different package!", proving that an object literal in a different package can initialize the `message` variable. However, since subsequent write access is script-only, we cannot change its value:

```
// Inside file two.fx
import tutorial.one;
var o = one {
    message: "Initialized this variable from a different package!"
}
o.message = "Changing the message..."; // WON'T COMPILE
println(o.message);
```

The compile-time error is:

```
two.fx:12: message has script only (default) write access in tutorial.one
o.message = "Changing the message..."; // WON'T COMPILE
  ^
1 error
```

This confirms the expected behavior: the variable can be publicly initialized, but subsequent writes are controlled by a different level of access.