

了解 JavaFX Script 编程语言 - 教程概述

By Sun 中国技术社区, 12/19/08

通过 JavaFX Script 编程语言，您可以使用精妙的图形用户界面来创建外观时尚的应用程序。这是一种为轻松编写 GUI 应用程序而全新设计的编程语言；借助其声明语法、数据绑定模型、动画支持及内置视觉效果，您能够使用更少的代码来完成更多的工作，从而缩短开发周期并提高生产力。

此教程是您学习 JavaFX Script 编程语言的起点。它只着重于介绍基础知识，即着重于所有 FX 应用程序通用的基本非可视核心结构。学完本教程后，您就可以学习使用 JavaFX 构建 GUI 应用程序了，它是该系列的第二个教程。学完第二个教程之后，媒体浏览器教程将引导您完成一个实际应用程序的完整端到端开发。

此外，高级开发者会对《JavaFX Language Reference》和应用编程接口 (Application Programming Interface, API) 文档感兴趣。这些参考文档对 JavaFX Script 编程语言的语法、语义以及支持的库进行了较低层面的讨论，此外还对 SDK 进行了讨论。

本教程中的课程包括：

- 第 1 课：JavaFX Script 入门 - 提供软件下载和安装说明，并介绍如何选择合适的开发环境。
- 第 2 课：编写脚本 - 介绍如何编译源代码、运行应用程序、声明脚本变量和调用脚本函数。
- 第 3 课：使用对象 - 介绍对象，说明如何声明对象字面值以及如何调用对象的函数。
- 第 4 课：数据类型 - 介绍内置数据类型 String、Number、Integer、Boolean 和 Duration，以及 Void 和 null 的用法。
- 第 5 课：序列 - 介绍序列数据结构，它用于存储和处理一组对象。本课讨论如何创建、使用和比较序列及其子集。
- 第 6 课：运算符 - 介绍支持的运算符（赋值运算符、算术运算符、一元运算符、关系运算符、条件运算符和类型比较运算符）。
- 第 7 课：表达式 - JavaFX Script 编程语言是一种表达式语言。本课解释这句话的含义并介绍可供您使用的各种类型的表达式。
- 第 8 课：数据绑定和触发器 - JavaFX Script 编程语言最强大的功能之一是能够自动将 GUI 与其底层数据同步。本课探讨数据绑定和触发器结构的基本原理。
- 第 9 课：编写您自己的类 - JavaFX API 提供了大量的类供您应用程序中使用。但是，有时您可能需要编写自己设计的自定义类。本课探讨编写自定义类所涉及的基础知识。
- 第 10 课：软件包 - 将您的源文件放置到命名的软件包中可以更好地组织您的代码（就名称空间管理而言）。本课通过循序渐进的示例（可引导您了解各种注意事项）来探讨如何创建和使用软件包。
- 第 11 课：访问修饰符 - 访问修饰符用来为变量、函数和类指定各种可见性级别。本课探讨可用的访问修饰符，并讨论当未提供访问修饰符时会出现什么情况。

除了概念解释和示例代码之外，某些课程还包含 SDK 演示代码摘录。研究这些代码片段有助于识别 .fx 源文件中的模式。这也有助于您巩固对每个新概念的理解。

第 1 课：JavaFX Script 入门

准备好研究 JavaFX Script 编程语言了吗？太棒了！本课介绍在开始之前必须在系统上安装的软件。在本课结束时，您将可以编写您的第一个脚本了！

目录

- 第 1 步：下载并安装 JDK
- 第 2 步：选择开发环境
- 第 3 步：下载并安装 JavaFX 编译器和运行时

第 1 步：下载并安装 JDK

JavaFX Script 编程语言基于 Java 平台，因此要求在系统上安装 JDK 5 或 JDK 6 (6 更快)。如果您尚未安装，请立即下载并安装 JDK 6 或 JDK 5，然后再继续学习本教程。

第 2 步：选择开发环境

就选择开发环境而言，有两大类供选择：集成开发环境 (Integrated Development Environment, IDE) 和纯文本编辑器。选择哪一种完全取决于个人喜好，但是下面的综述可以帮助您在作决定时更有依据。

一般而言：

IDE 提供一个集各种功能于一身的完整开发环境。您可以下载一个能够提供编译/运行/调试应用程序所需全部内容的软件（或是该软件的一个插件）。IDE 以图形用户界面 (Graphical User Interface, GUI) 元素形式提供最常用的函数，此外还提供许多有用的功能（如代码自动完成）。IDE 还可以立即为您提供有关错误的反馈并突出显示代码以使它们更便于理解。

文本编辑器比较简单而且更为用户所熟悉。有经验的程序员通常依赖他们所选择的文本编辑器，他们喜欢尽可能地在该环境中工作（某些编辑器 - 如 vi - 具有一组丰富的内置击键命令，有些程序员几乎离不开这些命令！）

JavaFX Script 编程语言正式支持的 IDE 是 NetBeans IDE 6.5。NetBeans IDE Web 站点提供了有关下载、安装和配置该 IDE 的说明。

第 3 步：下载并安装 JavaFX 编译器和运行时

您还需要下载并安装 JavaFX Script 编译器和运行时。获取此软件的一种方法是下载整个 JavaFX SDK，该 SDK 提供编译器、运行时和许多其他工具。

另一种方法是仅从 OpenJFX 项目 Web 站点下载最新的编译器二进制文件。编译器本身是用 Java 编程语言编写的；因此，安装预先编译的二进制文件将涉及到提取已下载的文件并将 javafx 和 javafx 工具添加到您的路径中。PlanetJFX Wiki 上提供了此方法的完整说明。

最后，如果您颇具创新意识，则可以加入 OpenJFX Compiler 项目，创建您自己的编译器工作区副本，然后亲自从编译器源文件生成所有内容。（如果您选择此方法，还需要 1.7.0 版本的 Apache Ant 以及最新的 Subversion 副本 - 在编写本教程时最新的版本为 1.5.4。）有关从源代码生成编译器的更多信息，请参见 Planet JFX Wiki。

第 2 课：编写脚本

本课对 JavaFX Script 编程语言进行了实例介绍。您将通过编写一个从命令行运行的简单的计算器来学习有关变量和函数的基础知识。每一节都将介绍一个新的核心概念，对其进行讨论，并提供您可以编译和运行的示例

代码。讨论还包含“实际”代码摘录，说明实际的 SDK 演示中是如何使用特定结构的。单击每个演示的链接可以访问 javafx.com Web 站点，该站点提供了完整的代码列表以及开发者的附加注释。

目录

- 声明脚本变量
- 定义和调用脚本函数
- 向脚本函数传递参数
- 从脚本函数返回值
- 访问命令行参数

声明脚本变量

前一课介绍了如何建立开发环境；这里我们将更深入地研究 [calculator.fx](#) 源代码。下面的红色代码声明了程序的脚本变量。脚本变量是使用 `var` 或 `def` 关键字声明的。二者之间的区别在于：在脚本的整个生命周期内都可以为 `var` 变量赋予新值，而 `def` 变量在被首次赋予新值后将保持不变。这里我们已经为 `numOne` 和 `numTwo` 赋予了特定的值，但未对 `result` 进行初始化，因为此变量将用来存放将来的计算结果：

```
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;
```

```

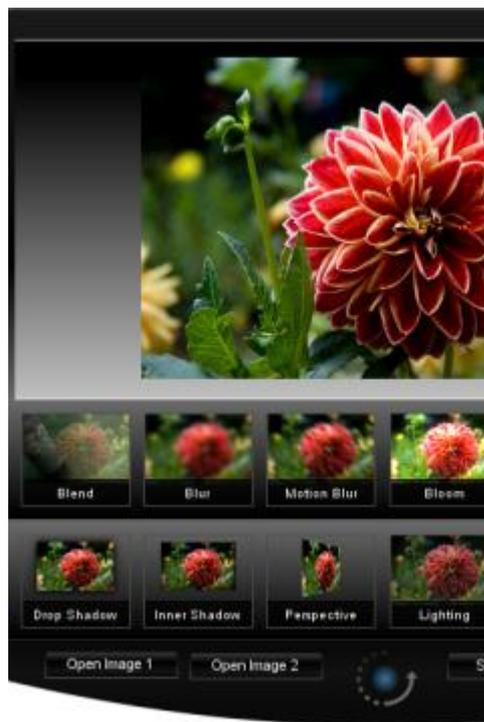
    println("{numOne} * {numTwo} = {result}");
}

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

```

您可能注意到，我们不需要将这些变量明确指定为存放数值型数据（而不是字符串或任何其他类型的数据）。编译器非常智能，可以根据使用变量的上下文来弄清您的意图。这称为**类型推断**。类型推断可以使脚本程序员的工作有所简化，这是由于它省去了声明变量与之兼容的数据类型的工作。

实际示例：效果场



```

...
def width = (6 * (82 + 10)) + 20;
def canvasWidth = width - 10;
def canvasHeight = 275;

var stage: Stage;

var inBrowser = "true".equals(FX.getArgument
var dragTextVisible =
    bind inBrowser and AppletStageExtension.
var closeButtonVisible = bind (not inBrowser

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

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

```

上面的屏幕抓图显示了“效果场”演示应用程序。右边的代码摘录显示了该应用程序的几个脚本变量。您还不能理解整个列表，但应该可以根据刚学过的内容看懂突出显示的部分。请记住，尽管本教程的重点仅针对非图形核心结构，但您最终要把这些知识用于自己的基于 GUI 的应用程序。

有关变量的较低层面的讨论，请参见《JavaFX Language Reference》中的“Chapter 3. Variables”。

定义和调用脚本函数

在计算器示例中还定义了一些脚本函数，用于对两个数字进行加、减、乘、除。函数是用来执行特定任务的可执行代码块。下面的红色代码定义了四个函数；每个函数执行一种简单的数学计算，然后输出结果。将代码组织成函数是一种常见的做法，这会使程序更易于阅读、使用和调试。函数体通常会缩进以增强可读性。

```
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}");
}
```

此外，除非函数代码被明确调用，否则不会执行。这样就可以在脚本的任何位置运行函数。将函数调用放在函数定义之前还是之后无关紧要（在我们的示例源文件中，函数是在实际定义之前调用的）：

```
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}");
}

```

实际示例：可拖动的 MP3 播放器



```

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

function playCurrentSong():Void {
    playlist.currentPlayingSong = playlist.song;
    mediaPlayer.media = playlist.currentPlayingSong.media;
    mediaPlayer.play();
}
...

```

在“可拖动的 MP3 播放器”演示中，程序员定义了用于停止或播放当前歌曲的函数。尽管我们正在研究的这些代码完全脱离了上下文，但这些函数的命名选择（`stopCurrentSong` 和 `playCurrentSong`）使这些代码是自说明的，因此更易于分析。在对变量和函数进行命名时，请尽量使用这种有意义的字词。约定名称中的第一词全部使用小写字母，后续每个词的首字母都使用大写字母。

向脚本函数传递参数

还可以将脚本函数定义为接受参数。参数是指在调用函数时所传入的特定值。这种方法可使计算器应用程序执行对任何两个数字（而不仅是硬编码到 numOne 和 numTwo 变量中的值）的计算。实际上，在该版本中已完全删除了 numOne 和 numTwo，仅保留了 result 脚本变量。

```
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}");
}
```

此脚本的输出现在为：

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

实际示例：有趣的照片



```

...
// Load image and data specified in given Photo object
function loadImage(photo: Photo, thumbImageView: ThumbnailImageView) {
    thumbImageView.image = Image {
        url: "http://farm{photo.farm}.static.flickr.com/{photo.server}/{photo.id}_{photo.secret}.jpg"
        width: thumbSize
        height: thumbSize
        backgroundLoading: true
        placeholder: thumbImageView.image
    };
    thumbImageView.photo = photo;
}
...

```

在这段来自“有趣的照片”演示的代码摘录中，我们看到一个名为 `loadImage` 的脚本函数，它可以接受一组参数。同样，函数和参数名称的选择使代码更容易理解。理解该函数的完整实现此时并不重要。重要的是识别可以接受两个参数的函数。当您开始编写自己的应用程序时，您很可能会依赖此类样例代码，从中学习正确的语法。

从脚本函数返回值

函数也可能会向调用它的代码返回一个值。例如，可以更改计算器的 `add` 函数，使其返回每次计算的结果：

```

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

```

第一段红色代码指定函数返回一个 `Integer`；第二段红色代码是实际返回值的代码。

现在，可以按如下方式调用 `add` 函数：

```

var total;

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

```

如果未指定返回值，函数会默认返回 Void。

实际示例：来自 Flickr 的动画照片



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

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

在“来自 Flickr 的动画照片”演示的这段代码摘录中，我们可以看到三个不同的函数使用了返回值。这些返回值比您以前见过的要稍微复杂一点，但核心概念是一样的：每个函数执行某种特定的计算，然后返回一个结果。前两个函数随后调用了 `Math` 函数（用于计算平方根）并返回其结果。第三个函数返回了一个新的 `Vector2D` 对象。单从此列表来看，没有足够的信息来确切地了解其含义，但借助完整的源代码您就可以看懂它们的含义了（如果您已经首先花时间学习了该语言！）

有关函数的较低层面的讨论，请参见《JavaFX Language Reference》中的 "Chapter 4. Functions"。

访问命令行参数

最后，脚本也可以接受命令行参数。在计算器示例中，这将使最终用户可以在运行时指定要进行计算的数字。

```
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}");
}

```

此更改引入了 `run` 函数，脚本通过此函数接收命令行参数。与您以前看到的其他函数不同，`run` 是用作脚本主入口点的特殊函数。`run` 函数会将所有命令行参数存储在 `args` (一个 *String* 对象序列) 中。(序列是对象的有序列表，与其他编程语言中的数组相似；[第 5 课：序列](#)中详细介绍了序列。)

为了运行此脚本，用户现在必须在运行时指定第一个数字和第二个数字：

```
javafx calculator 100 50
```

输出现在为：

```

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

```

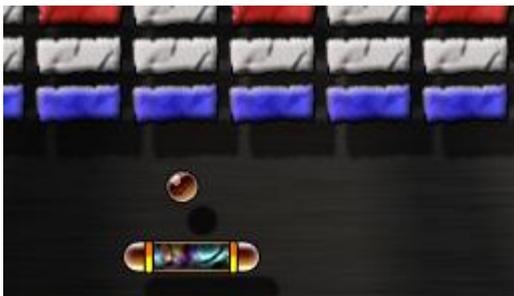
请注意,在所有以前版本的计算器脚本中,我们没有明确提供 run 函数,我们只是在脚本级别键入了代码,然后代码按照预期方式运行。在这些情况下,编译器会在无提示的情况下生成一个不带参数的 run 函数,并将要执行的代码放在该函数中。当指定您自己的 run 函数时,名称 args 可以是您希望的任何字符串;这里使用了 "args",但您很可能会看到程序员使用该名称的其他变体,如 "arg"、"ARGS"、"__ARGS__"、"argv" 等。

同时请注意,在该版本中我们重新使用了变量 numOne 和 numTwo,现在是在 run 函数中(而不是在脚本级别)定义它们。(当变量在函数中定义时,该变量在技术上称作局部变量,因为只有该函数中的其他代码可以看到该变量。)我们的计算器函数需要接受数字,但命令行参数是字符串;因此,我们必须首先将每个命令行参数从 String 转换为 Integer,然后才能将它们传递给函数:

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

为此,我们借助于 Java 编程语言来执行实际的类型转换。根据需要利用现有的 Java 生态系统会给这个原本简单的脚本语言带来强大的功能。

实际示例:撞砖



```
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
        }
    }
}
```

这段摘录来自“撞砖”演示,显示了用作游戏主入口点的 run 函数。尽管这个特殊示例实际上并未使用其命令行参数,但我们可以看到,run 函数对应用程序的主框架进行了初始化(设置其标题、宽度、高度等)。

第 3 课:使用对象

本课介绍如何使用对象。首先在较高层面上介绍什么是对象,然后介绍对象面值、实例变量和实例函数。本课提供您可以编译的 Address 和 Customer 示例,最后给出说明如何在实际应用程序中使用对象的代

码摘录。

目录

- 什么是对象？
- 声明对象字面值
- 对象字面值语法
- 调用实例函数

什么是对象？

JavaFX Script 编程语言是一种 *面向对象* 的语言。但这意味着什么呢？对象究竟是什么？简而言之，对象是独立的软件包，由 *状态* 和 *行为* 构成。为了更好地理解软件对象，不妨暂且跳出软件范围，考虑一下您已经熟悉的概念。

例如，您的电视机就是一个对象。它具有状态（当前频道、当前音量、开机状态、关机状态）和行为（更换频道、调节音量、打开、关闭）。我们往往认为电视机是单个对象，但实际上，电视机是由许多其他对象组成的（电视机外面的按钮和旋钮都是对象，其内部的各个组件也都是对象）。在许多情况下，这些较小的对象也是由...你猜对了...其他对象组成的。

我们可以不断分解电视机的组成部分，直到无法继续分解为止（如螺钉，确实只是单个对象，不是由任何其他对象组成的）。如果您购买过运动装备或其他任何“需要进行某种程度的装配”的东西，您可能在其文档中看到过一张分解图，显示其结构中包含的每一个对象。您一眼就可以看出有多少个对象以及这些对象是如何组合在一起的。对于 .fx 源文件也是如此：您可以很容易地看出脚本的所有对象如何组合在一起以形成完整的应用程序。

声明对象字面值

那么就代码而言，软件对象实际是什么样的呢？在 JavaFX Script 编程语言中，对象是使用 *对象字面值* 创建的：

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

以上代码创建了一个 Address 对象，用于假设的通讯录应用程序。使用该对象的 street、city、state 和 zip 的特定值对该对象进行了初始化。在实际应用程序中，您可以想象该通讯录的 GUI 组件将与其底层 Address 对象同步，这样屏幕上显示的内容将反映程序存储的实际数据。

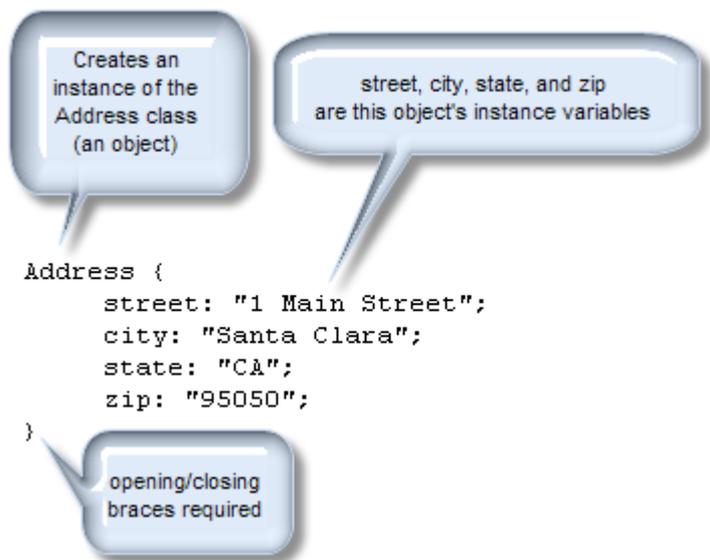
但在您尝试编译此代码之前，您还需要了解这一点：编译器首先需要有一个用于描述 Address 对象究竟包含什么数据的特殊“蓝图”（称为类）。（在该示例中，Address 包含街道、市、省/自治区和邮编，但编译器并不知道这些，除非我们提供一个用于提供这些信息的类文件。）在本教程的最后才会介绍如何编写您自己的类，因为 JavaFX 应用编程接口 (Application Programming Interface, API) 包含一个大型的内置类库，您可以在自己的程序中使用这些类。此外，由于 JavaFX Script 编程语言建立在 Java 平台上，因此您还可以访问 Java 编程语言的 API。

如前所述，您可以通过下载 Address.fx 文件（该文件提供 Address 类定义）并将其放置到与上面的示例代码所在目录相同的目录下来编译 Address 示例。（务必将示例代码保存到具有不同名称 - 如 AddressTest.fx - 的文件中）。这些代码不产生任何输出，但编译事实本身表明已成功创建对象。

注意：该示例中 Address 对象的变量（street、city、state、zip）在技术上称为实例变量。您可以将实例变量看作每个对象中都肯定会包含的一组内置属性。实际上，JavaFX Script 编程语言的早期版本中使用了“属性”一词（您仍会偶尔在旧的文档和演示中看到该词）。在面向对象编程领域中，“实例”和“对象”是同义词，这正是“实例变量”一词的由来。

对象字面值语法

对象字面值语法既简单易学又直观好用。继续看我们的示例，第一个词 (Address) 指定要创建的对象的类型。左花括号和右花括号定义对象的主体。其他内容 (street、city、state 和 zip) 初始化对象的实例变量。



请注意，当声明一个对象字面值时，可以使用逗号、空白或分号分隔实例变量。您可以使用上述任一种分隔符，但在本教程中我们一般使用分号。有关该语法更形式化的描述，请参阅《[JavaFX Language Reference](#)》中的 "Figure 6.38"。

您还可以将新创建的对象赋给变量以供以后访问：

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

或将一个对象嵌套在另一个对象中：

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

在最后一个示例中，`Customer` 对象引入了几个新变量，在名为 `address` 的变量中包含最初的 `Address` 对象。请注意在对象开始嵌套时我们是如何缩进代码的。实际应用程序通常会包含许多嵌套对象；此嵌套模式使您很容易一眼就看出对象之间的所属关系。要编译该示例，需要将 `Customer.fx` 添加到您当前的目录。

实际示例： [视频拼图](#)



```

...
var previewSpotX = 12; var previewSpotY = 12;
var overlaySpotX = 95; var overlaySpotY = 95;
var previewDimOverlay = Group {
    visible: false
    content: [
        Rectangle {
            x: 7 y: 25 width: 685 height: 300
            arcWidth: 10 arcHeight: 10
            fill: Color.rgb(0, 0, 0, 0.4)
            blockMouse: true
        }
    ]
}
...

```

视频拼图演示接收流式视频输入并将其分割成拼图。这里突出显示了该示例中使用对象字面值的一段代码。即使您不熟悉所给出的类，也应该能够看懂该语法。在此代码摘录中，`visible` 和 `content` 实例变量属于 `Group` 对象。`x`、`y`、`width`、`height`、`arcWidth`、`arcHeight`、`fill` 和 `blockMouse` 实例变量属于 `Rectangle` 对象。`Group` 对象已储存到一个名为 `previewDimOverlay` 的变量中，以供以后访问。

调用实例函数

除实例变量之外，对象还可以包含实例函数。您可以通过以下方法来调用对象的实例函数：键入变量的名称（在下面的示例中为 `customer`），后跟一个句点（`.`），句点后跟您要调用的函数：

```

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();

```

这将生成以下输出：

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

如前一课中所述，也可以将实例函数定义为接受参数并返回值。

第 4 课：数据类型

现在您已经了解了变量、函数和对象。本课探讨内置的*数据类型*。JavaFX Script 编程语言支持字符串类型、数值类型、布尔 (true/false) 类型。还支持基于时间 (持续时间) 的类型，以及用于指示函数不返回任何值和指示缺少正常值的特殊类型。

本课重点介绍大多数脚本编写者想了解的基本信息。有关类型系统的较低层面的讨论，请参见《[JavaFX Language Reference](#)》中的 "Chapter 2. Types and Values"。

目录

- String
- Number 和 Integer
- Boolean
- Duration
- Void
- null

String

您已经见过许多 String 示例，但现在让我们更深入地研究一下该数据类型。可以使用单引号或双引号来声明 String：

```

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

单引号和双引号是对称的：您可以在双引号中嵌入单引号，也可以在单引号中嵌入双引号。使用单引号括起来的字符串和使用双引号括起来的字符串没有任何区别。

您还可以使用花括号 "{}" 在字符串中嵌入表达式：

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

嵌入的表达式本身可以包含用引号括住的字符串，这些字符串中又可以进一步嵌入表达式：

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

在运行时，如果 `answer` 的值为 `true`，则编译器会将上面的粗体表达式替换为字符串 `"Yes"`，否则会将其替换为 `"No"`。

要联接（串联）多个字符串，请在引号中使用花括号：

```
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 和 Integer

`Number` 和 `Integer` 类型表示数值型数据，尽管对于大多数脚本任务来说，您通常只需让编译器推断正确的类型：

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

但是，您可以显式声明变量的类型：

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

这两种类型之间的区别是，`Number` 表示浮点数字，而 `Integer` 仅表示整数。只有当您确实需要浮点精度时才应使用 `Number`，否则应首选 `Integer`。

注意：从 SDK 1.1 开始，该语言还包含与 Java 编程语言中的数值类型相一致的数值类型。因此，数值类型的完整列表为：Byte、Short、Number、Integer、Long、Float、Double 和 Character。但以上给出的建议仍是正确的：大多数程序员在其编写的脚本中只需要使用 Integer（或 Number）。如果您在学习该语言时具有 Java 编程语言背景并且需要完成一个必须使用其他数值类型的任务，那么请记住您的脚本现在可以使用这些附加类型。

Boolean

Boolean 类型表示两个值：true 或 false。在以下两种情况下使用此类型：设置某个特定于应用程序的内部状态时：

```
var isAsleep = true;
```

或者计算条件表达式时：

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

如果小括号 "()" 中的表达式为 true，将执行花括号 "{}" 中的代码。有关条件表达式的更多信息，请参见 [表达式](#) 一课。

Duration

Duration 类型表示固定的时间单元（毫秒、秒、分钟或小时）。

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

持续时间用 *时间* 字面值来标记，例如，5m 是一个表示五分钟的时间字面值。时间字面值最常用在动画（您将在使用 [JavaFX 构建 GUI 应用程序中的创建动画对象](#) 一课中学习动画）中。

Void

Void 用来指示函数不返回任何值：

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

这与下面的代码等效，这些代码中省略了函数的返回类型：

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

JavaFX 关键字 `Void` 以大写字母 **V** 开头。如果您熟悉 Java 编程语言中的 `void`，则应当注意这一点。

注：在 JavaFX 中，一切都是表达式。第二个 `printMe` 函数的返回类型也是 `Void`，因为编译器能够推断其类型。[表达式](#) 一课将对此进行深入介绍。

null

`null` 是一个特殊的值，用来指示缺少正常值。`null` 与零或空字符串不同，因此 `null` 比较与零或空字符串比较不同。

允许使用 `null` 关键字进行比较。您通常会看到以下使用 `null` 的情形：

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

此函数接受一个参数，然后执行简单的测试来检查其值是否为 `null`。

第 5 课：序列

除了基本的数据类型外，JavaFX Script 编程语言还提供称为*序列*的特殊数据结构。序列代表按顺序排列的对象列表（但序列本身不是对象）。序列中的对象称为*项*。使用方括号 `[]` 来声明序列，其中的每个项都用一个逗号进行分隔。本课包含创建和使用序列的所有基础知识，此外还介绍如何使用序列子集（序列

的一部分)。有关附加信息，请参见《JavaFX Language Reference》中的 "Sequence Types"。

目录

- 创建序列
- 使用谓词
- 访问序列中的项
- 在序列中插入项
- 从序列中删除项
- 颠倒序列中项的次序
- 比较序列
- 使用序列子集

创建序列

创建序列的一种方法是显式列出其各个项。每个项都用一个逗号进行分隔，列表用方括号 "[" 和 "]" 括起来。例如，下面的代码：

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

声明了一个序列并将其赋给 weekDays。我们在这个例子中使用了 def，因为在创建序列后我们不打算改变它的值。这里，编译器知道我们打算创建一个“字符串序列”，因为每个项都声明为 String 字面值。如果序列是使用 Integer 声明的(例如,def nums = [1,2,3];)，编译器将知道我们打算创建“整数序列”。

您还可以显式指定序列的类型，方法是修改序列的声明，使其包含后跟 "[]" 的类型名称。

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

这会通知编译器 weekDays 将用来存放 String 序列（而不是单个 String）。

您还可以在序列中声明其他序列：

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

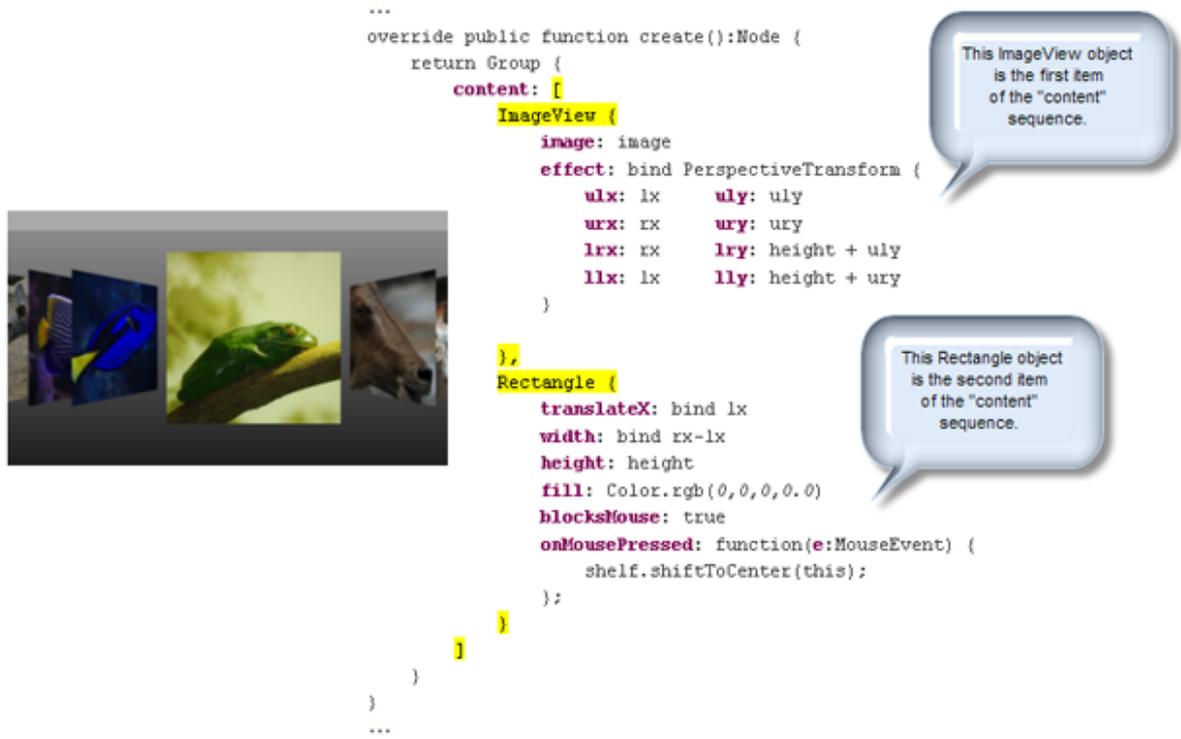
在这种情况下，编译器将自动平展开嵌套的序列以构成单个序列，这样上面的代码就等效于：

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

还可以通过简化表示法来更方便地创建可构成等差数列的序列。要创建一个由数字 1 至 100 构成的序列，请使用以下代码：

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

实际示例：展示架 (序列示例 1)



The screenshot shows a graphical user interface on the left with three items on a shelf: a blue object, a green parrot, and a brown object. To the right is a code snippet for a 'create' function. The code defines a 'content' sequence containing an 'ImageView' and a 'Rectangle'. Two callout boxes provide context: one points to the 'ImageView' in the code, stating it is the first item of the 'content' sequence; the other points to the 'Rectangle' in the code, stating it is the second item of the 'content' sequence. The code is as follows:

```

...
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);
                };
            }
        ]
    }
}
...

```

此屏幕抓图显示了“展示架”演示应用程序。右边的代码摘录显示了一个属于 Group 对象的实例变量(名为“content”)。该变量用来存放已使用两个项(突出显示的 ImageView 和 Rectangle 对象)进行初始化的序列。您可以通过查找序列的左方括号和右方括号(也已突出显示)来识别该序列。序列中的项以逗号分隔，因此 ImageView 和 Rectangle 对象之间有一个“,”。

使用谓词

您可以使用布尔表达式(又称为谓词)声明一个为现有序列的子集的新序列。例如，请考虑以下代码：

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

要创建第二个序列（基于此第一个序列中的项）但仅包含大于 2 的数字，请使用以下代码：

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

可以将上面的代码行用中文表示为：“从 `nums` 序列中选择**项值大于 2** 的所有项并将这些项赋给名为 `numsGreaterThanTwo` 的新序列。”以粗体突出显示的“项值大于 2”子句是谓词。

在这些代码中：

1. 新创建的序列存储在 `numsGreaterThanTwo` 中。
2. 代码 `nums[n | n > 2]`；中标记为粗体的部分指定要从中复制项的原始序列。在我们的示例中，`nums` 是原始序列的名称。
3. 这会选择 `nums` 中的项，并按顺序返回一个由使表达式为 `true` 的项构成的新序列。
4. “|” 字符用来在视觉上将变量 “n” 与代码的其余部分隔开：`nums[n | n > 2]`；
5. 代码 `nums[n | n > 2]`；中标记为粗体的部分定义一个布尔表达式，该表达式指定将当前项复制到新序列中时需要满足的条件。

访问序列中的项

序列中的项是按数字索引（从 0 开始）进行访问的。要访问单个项，请键入序列名称，后跟该项的数字索引（用方括号括起来）：

```
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]);
```

这会将以下内容输出到屏幕上：

```
Mon
Tue
```

```
Wed
Thu
Fri
Sat
Sun
```

您还可以使用与序列名称的 `sizeof` 运算符来确定序列的大小：

```
sizeof days
```

以下代码会将 "7" 输出到屏幕上：

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

在序列中插入项

`insert` 关键字可用于在序列中特定项的*前面或后面*插入一个项。

注：从技术上讲,序列是*不可变的*,即一经创建就永不更改。在修改序列(例如通过插入或删除项)时,会在后台创建一个新序列并重新指定序列变量,这会给人一种序列已被修改的印象。

让我们通过重新创建 `days` 序列来对此进行研究。请注意,我们现在用 `var` 来声明变量 `days`,因为我们会在创建原始序列后改变它的值：

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

此时,该序列仅包含一个项 "Mon"。

我们可以使用 `insert` 和 `into` 关键字在该序列的末尾插入 "Tue"：

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

类似地,我们也可以添加 "Fri"、"Sat" 和 "Sun"：

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

该序列中现在包含："Mon"、"Tue"、"Fri"、"Sat" 和 "Sun"。

我们还可以使用 `insert` 和 `before` 关键字在给定索引处的项前面插入一个项。请记住，索引是从 0 开始的，因此，在当前的序列中，"Fri" 位于索引位置 2。因此，我们可以在 "Fri" 前面插入 "Thu"，如下所示：

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

该序列中现在包含："Mon"、"Tue"、"Thu"、"Fri"、"Sat" 和 "Sun"。

要在 "Tue" 后面插入 "Wed"，我们可以使用 `insert` 和 `after` 关键字：

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

该序列现在包含一周中的每一天："Mon"、"Tue"、"Wed"、"Thu"、"Fri"、"Sat" 和 "Sun"。

从序列中删除项

通过使用 `delete` 和 `from` 关键字可以轻松地从序列中删除项：

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

该序列中现在包含："Mon"、"Tue"、"Wed"、"Thu"、"Fri" 和 "Sat"。

还可以删除位于特定索引位置的项。以下代码从该序列中删除 "Mon"（请记住，"Mon" 是第一个项，因此它的索引位置为 0）。

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

要删除序列中的所有项，请使用后跟序列名称的 `delete` 关键字：

```
delete days;
```

请注意，`delete` 仅从序列中删除项，而不从脚本中删除 `days` 变量。您仍可以像以前那样访问 `days` 变量并向其中添加新项。

颠倒序列中项的次序

使用 `reverse` 运算符可以轻松地颠倒序列中项的次序：

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

比较序列

有时，您可能希望对序列进行比较，看它们是否相等。序列是按值来比较是否相等的：如果它们的长度相同而且各个项相等，则它们相等。

让我们创建两个具有相同内容的序列来对此进行测试：

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

表达式 `seq1 == seq2` 的值为 `true`，因为这两个序列具有相同数量的项，而且这两个序列中每个项的值都相等。因此，这些代码会将 `"true"` 输出到屏幕上。

通过更改其中一个序列中项的数量（而不更改另一个序列），这两个序列现在具有不同的长度：

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

由于第二个序列比第一个序列长，从而这两个序列不相等，因此该脚本的输出为 `"false"`。

我们还可以通过更改项的值来使两个序列不相等（即使这两个序列的长度仍相同）：

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

由于这两个序列不相等，因此这些代码将再次输出 "false"。

使用序列子集

序列子集提供对序列某些部分的访问。

seq[a..b]

此语法提供对位于索引 a 和索引 b 之间 (含 a 和 b) 各个项的访问。以下脚本创建一个 weekend 序列，其中仅包含 "Sat" 和 "Sun" 项。

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

seq[a..<b]

使用 "<" 字符可以访问位于索引 a 和索引 b 之间 (含 a, 不含 b) 的项。我们可以对 days 使用此方法来创建一个 weekdays 序列，其中包含项 "Mon" 至 "Fri"。

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

seq[a..]

通过省略第二个索引，可以访问从索引 a 到序列末尾处的所有项。为了与上一个示例保持一致，我们可以按如下方式创建 weekend 序列：

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

seq[a..<]

最后，您可以使用不带第二个索引的 "<" 来访问从索引 a 到序列末尾处的所有项 (不含最后一项)。

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

此版本会创建一个 days2 序列，其中包含项 "Mon" 至 "Sat"。

实际示例：展示架（序列示例 2）

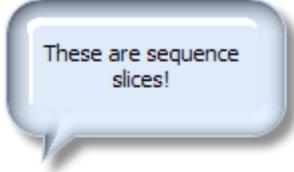
```
...
override public function create():Node {
    var half = content.size()/2-1;

    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
        ]
    }
}
...

```



可以看到，这个来自“展示架”演示应用程序的代码列表中使用了两个序列子集。这两个子集声明的范围分别为 "0..half-2" 和 "half..content.size()-1"。要识别一个序列子集，请在方括号中查找 ".."。

第 6 课：运算符

运算符是一些特殊的符号，用来对一个或两个操作数执行特定的运算，然后返回一个结果。JavaFX Script 编程语言提供赋值运算符、算术运算符、一元运算符、相等和关系运算符、条件运算符和类型比较运算符。有关支持的运算符的形式化描述，请参见《JavaFX Language Reference》中的 "Table 6.2 - Table 6.8"。

目录

- 赋值运算符

- 算术运算符
- 一元运算符
- 相等和关系运算符
- 条件运算符
- 类型比较运算符

赋值运算符

赋值运算符 "=" 是您将遇到的最常用的运算符。使用该运算符可以将其右侧的值赋给其左侧的操作数：

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

在前面的许多课程中，您已经使用过该运算符。

算术运算符

使用*算术运算符*可以执行加、减、乘和除运算。mod 运算符用一个操作数除以另一个操作数并将余数作为结果返回。

```
+ (加运算符)
- (减运算符)
* (乘运算符)
/ (除运算符)
mod (求余运算符)
```

以下脚本提供了一些示例：

```
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);
```

您还可以结合使用算术运算符与赋值运算符来创建复合赋值。例如 `result += 1;` 和 `result = result+1;` 都会将 `result` 的值加 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);
```

唯一不能按照此方式使用的算术运算符是 `mod`。例如，如果您希望将 `result` 除以 2，然后将余数重新赋给其自身，则需要编写：`result = result mod 2;`

一元运算符

大多数运算符都需要两个操作数，而一元运算符仅使用一个操作数来执行诸如按 1 递增/递减某个值、对某个数字求反或对布尔值求反之类的操作。

-	一元减运算符；对某个数字求反
++	递增运算符；按 1 递增某个值
--	递减运算符；按 1 递减某个值
not	逻辑求补运算符；对布尔值求反

以下脚本用于测试一元运算符：

```

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

```

递增/递减运算符可以在操作数之前（前缀）或之后（后缀）应用。代码 `result++;` 和 `++result;` 都将导致 `result` 的值加 1。二者之间的唯一区别就是前缀版本 (`++result`) 得到的是递增后的值，而后缀版本 (`result++`) 得到的是原始值。（您可以通过以下方法来记忆：`++result` 先执行递增再获得值，而 `result++` 先获得值再执行递增。）如果您只是执行简单的递增/递减，则选择哪个版本都一样。但是，如果您将该运算符作为较大表达式的一部分进行使用，则选择不同的版本会对结果产生很大的影响。

以下脚本说明了这种区别：

```

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

```

相等和关系运算符

相等和关系运算符确定一个操作数是大于、小于、等于还是不等于另一个操作数。

<code>==</code>	等于
<code>!=</code>	不等于
<code>></code>	大于
<code>>=</code>	大于或等于

```
<      小于
<=     小于或等于
```

以下脚本用于测试这些操作数：

```
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
```

条件运算符

条件与 (and) 和 *条件或 (or)* 运算符用于对两个布尔表达式执行条件运算。这些运算符会表现出“短路”行为,也就是说,仅在必要时才计算第二个操作数:例如,对于 *and* 运算,如果第一个表达式的结果为 *false*,将不计算第二个表达式。对于 *or* 运算,如果第一个表达式的结果为 *true*,将不计算第二个表达式。

```
and
or
```

以下脚本定义了 *username* 和 *password* 变量,然后输出各个条件的匹配项,从而说明了这些运算符的用法:

```
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");
}

```

输出为：

```

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

```

类型比较运算符

`instanceof` 运算符将对象与指定的类型相比较。您可以使用该运算符来确定某个对象是否为特定类的实例：

```

def str1="Hello";
println(str1 instanceof String); // prints true

def num = 1031;
println(num instanceof java.lang.Integer); // prints true

```

在本教程的最后一课中深入了解类和继承后，您将发现该运算符非常有用。

第 7 课：表达式

表达式是可以生成某个结果值的代码段，可以结合使用表达式来生成“更大的”表达式。JavaFX Script 编程语言是表达式语言，这意味着一切（包括循环、条件甚至块）都是表达式。在某些情况下（如 `while` 表达式），表达式具有 `Void` 类型，这意味着它们不返回结果值。有关表达式的较低层面的讨论，请参见《[JavaFX Language Reference](#)》中的“[Chapter 6. Expressions](#)”。

目录

- 块表达式
- if 表达式

- 范围表达式
- for 表达式
- while 表达式
- break 和 continue 表达式
- throw、try、catch 和 finally 表达式

块表达式

块表达式由一系列声明或表达式组成，它们括在花括号中并用分号进行分隔。块表达式的值是最后一个表达式的值。如果块表达式中不包含表达式，则其类型为 Void。请注意，var 和 def 是表达式。

下面的块表达式对几个数字进行相加并将结果存储在一个名为 total 的变量中：

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

运行此脚本将生成以下输出：

```
Total is 24.
```

第一行 (var nums = [5, 7, 3, 9];) 声明一个整数序列。

第二行声明一个名为 total 的变量，该变量将用来存放这些整数的和。

随后的块表达式由左花括号和右花括号之间的所有内容构成：

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

在该块内部，第一行代码声明一个名为 `sum` 的变量，该变量将用来存放此序列中各个数字之和。第二行（一个 `for` 表达式）遍历该序列，将每个数字与 `sum` 相加。最后一行设置该块表达式的返回值（在本例中为 24）。

if 表达式

使用 `if` 表达式后，仅当特定条件为真时才执行某些代码块，从而对程序流进行定向。

例如，以下脚本基于年龄来设置票价。12 岁到 65 岁的人支付正常价 10 美元。老人和儿童支付 5 美元；5 岁以下的儿童免费。

```
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.");
```

如果将 `age` 设置为 8，该脚本将生成以下输出：

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

该示例的程序流如下所示：

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

如果 `age` 小于 5，则票价将设置为 0。

程序随后将跳过其余条件测试并输出结果。

如果 age 不小于 5 ,程序将继续执行下一个条件测试(由后跟另一个 if 表达式的 else 关键字来指示) :

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

如果人的年龄在 5 到 12 岁之间或者大于 65 岁 , 该程序会将票价设置为 5 美元。

如果人的年龄在 12 到 65 岁之间 , 程序会流至最后一个代码块 (用 else 关键字进行标记) :

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

只有当前面的所有条件均不满足时 , 才会执行此块。它会针对 12 到 65 岁之间的人将票价设置为 10 美元。

注 : 可以将上面的代码缩减成一个非常简洁的条件表达式 :

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

这是一个需要掌握的有用方法 , 在本教程的后面部分中还会使用它。

范围表达式

“序列” 一课讲授了一种用来声明形成等差数列的数字序列的简化表示法。

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

从技术上讲，`[0..5]` 是一个范围表达式。默认情况下，相邻值之间的间隔为 1，但是您可以使用 `step` 关键字来指定一个不同的间隔。例如，定义一个由 1 到 10 之间的奇数构成的序列：

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

此脚本的输出如下所示：

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

要创建降序范围，请确保第二个值小于第一个值，并指定一个负的 `step` 值：

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

输出为：

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

如果您在创建降序范围时没有提供负的 `step` 值，则会生成一个空序列。

以下代码：

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

将生成下面的编译时警告：

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

```
    ^  
1 warning
```

如果您完全忽略 `step` 值，也会生成一个空序列。

for 表达式

另一个与序列有关的表达式是 `for` 表达式。`for` 表达式为遍历序列中的各个项提供了一种方便的机制。

以下代码提供了一个示例：

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

此脚本的输出如下所示：

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

让我们将该示例分成几个部分。`for` 表达式以 `"for"` 关键字开头：

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

`days` 变量是要由 `for` 表达式处理的输入序列的名称：

```
for (day in days) {
```

```
println(day);
}
```

当 `for` 表达式遍历该序列时，`day` 变量用来存放当前项：

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

请注意，不需要在脚本中的其他位置声明 `day` 变量即可将其用在 `for` 表达式中。此外，在完成整个循环之后，将无法访问 `day`。程序员通常会赋予临时变量（如该变量）非常短的名称（或由一个字母构成的名称）。

在上例中，未显示 `for` 返回值；但 `for` 也是一个返回序列的表达式。以下代码显示了两个使用 `for` 表达式从另一个序列创建序列的示例：

```
// 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();
```

请注意，`toUpperCase` 函数由 `String` 对象提供。您可以通过查阅 API 文档来查看完整的可用函数列表。

while 表达式

另一个循环结构是 `while` 表达式。与作用于序列中项的 `for` 表达式不同，`while` 表达式会一直循环，直到给定的表达式为 `false` 为止。尽管 `while` 在语法上是表达式，但是它的类型为 `Void`，不返回值。

下面提供了一个示例：

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

此脚本的输出如下所示：

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

第一行声明一个名为 `count` 的变量并将其初始化为 0：

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

第二行以 `while` 表达式开头。此表达式创建了一个循环（在左花括号和右花括号之间），该循环会一直进行，直到 `count < 10` 的值为 `false` 为止：

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

`While` 表达式的主体会输出 `count` 的当前值，然后将 `count` 的值加 1：

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

当 count 等于 10 时，循环退出。要创建一个无限循环，请将 true 关键字放在左小括号和右小括号之间，如 while(true){} 中所示。

break 和 continue 表达式

break 和 continue 表达式与循环表达式有关。这两个表达式会影响循环迭代：break 完全放弃循环，而 continue 仅放弃当前迭代。

尽管 break 和 continue 在语法上是表达式，但它们的类型为 Void，不返回值。

示例：

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

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

    println(i);
}
```

输出：

```
1
3
5
```

如果没有 if 表达式，该程序将只是输出 0 到 10 之间的数字。

如果只有第一个 if 表达式，程序将在 i 的值大于 5 时中断循环：

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

因此，程序将仅输出 1 到 5 之间的数字。

通过添加第二个 if 表达式，程序将仅放弃循环的当前迭代而继续执行下一个迭代：

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

在这种情况下，只有当 i 为偶数（即 i 能被 2 整除，没有余数）时才执行 `continue`。出现这种情况时，将永远不会调用 `println()`，因此输出中将不包含该数字。

throw、try、catch 和 finally 表达式

在实际的应用程序中，正常的脚本执行流有时会被某个事件中止。例如，如果脚本从某个文件中读取输入，但是找不到该文件，该脚本将无法继续。我们将这种情况称为“异常”。

注意：异常是对象。它们的类型通常以它们所表示的情况命名（例如，`FileNotFoundException` 表示找不到文件的情况）。但是，定义一组特定于即将给出的示例的异常不在本节的讨论范围之内。因此，我们将使用一个通用的 `Exception` 对象（从 Java 编程语言借用而来）来说明 `throw`、`try`、`catch` 和 `finally` 表达式。

以下脚本定义（和调用）一个会抛出异常的函数：

```
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.");
    }
}
```

按原样运行此脚本（将 `somethingWeird` 设置为 `false`）将输出以下消息：

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

但是，如果将该变量更改为 `true`，则会抛出异常。在运行时，该脚本将崩溃并显示以下消息：

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

为了防止崩溃，我们将需要使用 `try/catch` 表达式来包装 `foo()` 调用。顾名思义，这些表达式尝试执行某些代码，但会在出现问题时捕获异常：

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

现在，程序不会崩溃，而只是输出：

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

还有一个 `finally` 块（它在技术上不是表达式），无论是否抛出了异常，该块始终在 `try` 表达式退出之后的某个时间执行。`finally` 块用来执行无论 `try` 主体是成功还是抛出异常都需要执行的清除操作。

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

```
}

```

程序输出现在为：

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

第 8 课：数据绑定和触发器

数据绑定（即在两个变量之间创建直接关系的功能）是 JavaFX Script 编程语言最为强大的功能之一。本课首先介绍如何绑定两个简单的变量，然后介绍如何在一个变量和一个函数/表达式的结果之间建立更复杂的绑定。在了解了这一概念之后，可以参阅[对 UI 对象应用数据绑定（使用 JavaFX 构建 GUI 应用程序中的一课）](#)，该课程举例说明了在构建 JavaFX 应用程序时数据绑定是一个多么强大的工具。

替换触发器是附加到变量的代码块 - 当变量发生变化时，代码会自动执行。本课提供了替换触发器的实际应用。

有关数据绑定的更多信息，请参见《[JavaFX Language Reference](#)》中的 "[Chapter 7. Binding](#)"。

目录

- [绑定概述](#)
- [绑定和对象](#)
- [绑定和函数](#)
- [对序列进行绑定](#)
- [替换触发器](#)

绑定概述

bind 关键字将目标变量的值与绑定表达式的值相关联。绑定表达式可以是某个基本类型的简单值、对象、函数的结果或表达式的结果。以下几节分别提供每种绑定表达式的示例。

绑定和对象

在大多数实际的编程情况下，需要通过数据绑定使应用程序的图形用户界面（Graphical User Interface, GUI）与其底层数据同步。（GUI 编程是[使用 JavaFX 构建 GUI 应用程序](#)的主题；下面我们将用一些非可视的示例来说明基本的底层结构。）

让我们先看一个简单的示例：下面的脚本将变量 `x` 绑定到变量 `y`，更改 `x` 的值，然后输出 `y` 的值。由于这两个变量绑定在一起，因此 `y` 值会自动更新为新值。

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

请注意，我们已将变量 `y` 声明为 `def`。这可防止任何代码直接为该变量赋值（尽管允许该变量的值因绑定（`bind`）而更改）。在绑定到对象时，此约定仍适用（请回顾[使用对象](#)中介绍的 `Address`）：

```
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}");
```

如果更改 `myStreet` 的值，`address` 对象内部的 `street` 变量将受到影响：

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

请注意，更改 `myStreet` 的值实际上会导致创建一个新的 `Address` 对象，然后将该对象重新赋给 `address` 变量。为了跟踪所做的更改而不创建新的 `Address` 对象，请改为直接绑定（`bind`）到该对象的实例变量：

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

如果要显式绑定到实例变量，还可以省略第一个 `bind` (`Address` 前面的那个)：

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

绑定和函数

前面的课程已讲授了函数，但是您还必须了解*绑定函数*与*非绑定函数*之间的区别。

请考虑下面的函数，该函数创建和返回一个 `Point` 对象：

```
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;  
}
```

这就是所谓的*绑定函数*，因为它前面有 `bound` 关键字。

注意：bound 关键字不能替换 bind 关键字；这两个关键字按如下所示方式结合使用。

接下来，让我们添加一些代码来调用此函数并测试绑定：

```
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);
```

此脚本的输出如下所示：

```
3.0
10.0
20.0
```

让我们分析一下此脚本（一次分析一部分）。

代码：

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

将脚本变量 `myX` 和 `myY` 初始化为 3.0。这些值随后作为参数传递给 `makePoint` 函数，该函数会创建并返回一个新的 `Point` 对象。`bind` 关键字(位于 `makePoint` 调用前面)将新创建的 `Point` 对象 (`pt`) 绑定到 `makePoint` 函数的结果。

接下来，代码：

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

将 `myX` 的值更改为 10.0 并输出 `pt.x` 的值。输出表明 `pt.x` 现在也为 10.0。

最后，代码：

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

更改 `scale` 的值并再次输出 `pt.x` 的值。`pt.x` 的值现在为 20.0。但是，如果我们从该函数中删除 `bound` 关键字（从而使其成为非绑定函数），则输出应为：

```
3.0
10.0
10.0
```

这是因为，非绑定函数只是在其某个参数发生变化时才被重新调用。由于 `scale` 不是函数的参数，因此更改它的值将不会导致另一个函数调用。

对序列进行绑定

您还可以将 `bind` 与 `for` 表达式结合使用。为了对此进行研究，让我们首先定义两个序列并输出这两个序列中各个项的值：

```
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 包含十个项 (数字 1 至 10)。seq2 也包含十个项；这些项本来会与 seq1 具有相同的值，但是我们已经对其中的每个项都应用了表达式 `item*2`，因此它们的值将加倍。

因此，输出为：

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

我们可以通过将 `bind` 关键字放在 `for` 关键字前面来绑定这两个序列。

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

问题现在变成：“如果 seq1 发生了某些变化，那么是 seq2 中的*所有项*都受到影响还是*部分项*受到影响？”我们可以通过以下方法来对此进行测试：将一个项（值 11）插入 seq1 的末尾处，然后输出这两个序列的值，看有什么变化：

```
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);}
}
```

输出：

```
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
```

输出表明，将 11 插入 seq1 的末尾处不会影响 seq2 中的前 10 项；新项会自动添加到 seq2 的末尾处，其值为 22。

替换触发器

*替换触发器*是附加到变量的任意代码块，一旦变量的值发生变化，它们就会执行。以下示例显示了基本语法：它定义一个 password 变量并向其附加一个替换触发器；当密码发生变化时，该触发器会输出一则消息来报告此变量的新值：

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

password = "bar";
```

此示例的输出如下所示：

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

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

此示例中的触发器引发两次：当 password 初始化为 "foo" 时引发一次，当其值变成 "bar" 时又引发一次。请注意，oldValue 变量存储在调用触发器之前变量的值。您可以将 oldValue 变量命名为任何所需的名称，我们是由于该名称具有描述性才恰好使用它。

第 9 课：编写您自己的类

JavaFX Script 编程语言 API 包含大量可在您的应用程序中立即使用的类。但在某些时候，您可能会发现需要自己编写一些特定于应用程序的类。本课对与此相关的内容进行较高层面的介绍。有关较低层面的介绍，请参见《JavaFX Language Reference》中的 "Chapter 5. Classes"。

目录

- Customer 示例
- 从其他类继承

Customer 示例

在使用对象中，您学习了如何使用对象。但是请回忆一下，我们那时提供了必需的 Address.fx 和 Customer.fx 文件。这些源文件包含了类定义，但我们对它们的内容未进行任何讨论。本节将回顾这个示例，但这次会将您需要的所有内容放入同一文件中：

```
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}");
}
}

```

由于您已经了解了变量和函数，因此应该对该示例比较熟悉。Address 类声明了 street、city、state 和 zip 实例变量，它们均为 String 类型。Customer 类也声明了几个实例对象，以及用于输出这些对象值的函数。由于这些变量和函数是在类中声明的，因此您创建的任何 Address 和 Customer 对象都可以使用它们。

从其他类继承

您还可以编写从其他类继承变量和函数的类。例如，设想您在银行有一个储蓄账户和一个支票账户。每个账户都有账号和余额。您可以查询余额、存款或取款。我们可以通过创建一个用来提供最常用的变量和函数的 Account 基类来对以上情况进行建模：

```

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;
    }
}

```

我们已经将该类标记为 `abstract`，这意味着不能直接创建 `Account` 对象（此设计的目的在于使您只能创建储蓄账户或支票账户）。

`accountNum` 和 `balance` 变量用来存放账号和当前余额。其余函数提供基本行为：获得余额、存款或取款。

然后，我们可以定义一个使用 `extends` 关键字来继承这些变量和函数的 `SavingsAccount` 类：

```

class SavingsAccount extends Account {

    var minBalance = 100.00;
    var penalty = 5.00;

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

```

由于 `SavingsAccount` 是 `Account` 的子类，因此它将自动包含 `Account` 的所有实例变量和函数。这样就可以使 `SavingsAccount` 源代码将重点放在它与其父类之间的区别（要求储蓄账户所有者必须至少保持 100 美元的余额才能避免受罚）上。

类似地，我们可以定义一个也对 `Account` 进行扩展的 `CheckingAccount` 类：

```

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!
    }
}
}
}

```

该类与 Account 的区别在于，它定义了一个用来跟踪账户所有者是否有透支保护（如果取款（例如写支票）导致余额小于零，则透支保护将激活以确保支票不会被退回）的变量。请注意，在这种情况下，我们将更改所继承的 withdraw 函数的行为。这就是所谓的**函数覆盖**（由 override 关键字指示）。

第 10 课：软件包

使用软件包可以组织类以及它们之间的相互关系。本课指导您如何设置和使用软件包。

目录

- 第 1 步：选择软件包名称
- 第 2 步：创建目录
- 第 3 步：添加软件包声明
- 第 4 步：添加访问修饰符
- 第 5 步：编译源文件
- 第 6 步：使用类

到现在为止，您已经熟练掌握了 JavaFX Script 编程语言的基础知识。但是，源文件的位置可能有点混乱（到现在为止，您可能只有一个目录，其中包含大量不相关的示例）。我们可以通过将代码放在**软件包**中来改进整体组织结构。

使用软件包可以按照功能对代码进行分组。软件包还为类提供一个唯一的名称空间。下面我们将研究一个分步示例，该示例将 Address 类放进一个特定的软件包中。

第 1 步：选择软件包名称

在修改任何代码之前，必须首先为将要创建的软件包选择一个名称。由于我们的 Address 类将要用在（假设的）通讯录应用程序中，因此我们将使用 "addressbook" 作为软件包名称。

第 2 步：创建目录

接下来，我们必须在文件系统本身上创建一个 `addressbook` 目录。此目录中将包含被指定为属于 `addressbook` 软件包的任何类的 `.fx` 源文件。您可以在任何所需的位置创建此目录；在本例中我们将使用 `/home/demo/addressbook`，但是，这些脚本必须位于与软件包名称（在本例中为 `addressbook`）相匹配的目录中。

第 3 步：添加软件包声明

现在，请转到 `addressbook` 目录并创建 `Address.fx` 源文件。将下面的源代码粘贴到该文件中。第一行提供*软件包声明*，它声明此类属于 `addressbook` 软件包：

```
package addressbook;

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

请注意，如果存在软件包声明，它必须单独出现在源文件的第一个代码行中。每个源文件中只允许有一个软件包声明。

第 4 步：添加访问修饰符

接下来，我们必须为 `Address` 类及其变量添加 `public` 关键字：

```
package addressbook;

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

此关键字是五个可用的*访问修饰符*之一。我们将在[下一课](#)研究访问修饰符。现在，您只需知道 `public` 使这些代码可供其他类和脚本访问即可。

第 5 步：编译源文件

仍在 `addressbook` 目录中，使用 `javafx Address.fx` 命令按照通常的方式编译此源文件。（在较大的软件项目中，可通过更复杂的方法从多个软件包生成代码，但在本例中编译此目录中的源代码即可。）在编译之后，此目录中将包含所生成的 `.class` 文件。

第 6 步：使用类

现在我们可以测试经过修改的 `Address` 类。但是，我们将首先回到父目录 `/home/demo`。在这里，我们将创建一个名为 `packagetest.fx` 的简单脚本，该脚本用来测试 `addressbook` 软件包的用法。

我们可以通过两种方法来访问此类：

```
// Approach #1

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

方法 1 使用完全限定类名（现在为 `addressbook.Address`）创建一个对象。与其他方法相比，此方法可能显得过于繁琐（尤其是在大型脚本中），但是您仍应当知道有这么一种方法。

```
// Approach #2
import addressbook.Address;

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

方法 2 使用 `import` 关键字，该关键字允许现在在脚本中的任何位置使用简称 (`Address`)。建议在较大的程序中使用此方法，因为它是自说明的。一看就知道每个类属于哪个软件包。

```
// Approach #3

import addressbook.*;
```

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

方法 3 使用通配符 "*" 来导入 addressbook 软件包中定义的所有公共类。在这个例子中这样做没有太多优势，但如果一个软件包中包含大量类，则可以使用此技术一次导入所有这些类。一些程序员发现这样做很方便；但这种做法的缺点是更加难以辨别一个给定的类位于哪个软件包中，特别是当涉及到多个软件包时更是如此。

第 11 课：访问修饰符

既然您已经了解了软件包，我们可以讨论 JavaFX Script 编程语言提供的各种 *访问修饰符*了。通过这些特殊的关键字可以为变量、函数和类设置各种可见性级别。有关更多信息，请参见《[JavaFX Language Reference](#)》中的 "Chapter 8. Access Modifiers"。

目录

- 默认访问
- package 访问修饰符
- protected 访问修饰符
- public 访问修饰符
- public-read 访问修饰符
- public-init 访问修饰符

默认访问

默认访问称为“仅脚本”，是指在未提供访问修饰符的情况下获得的访问。我们在本教程中大多数都在使用这种访问。

下面是一些示例：

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

使用此访问级别，只能从脚本内部初始化、覆盖、读取、指定或绑定变量。其他任何源文件都不能读取或访问这些信息。

package 访问修饰符

要使变量、函数或类可供同一个软件包中的其他代码访问，请使用 package 访问修饰符：

```
package var x;
```

一定不要将该访问修饰符与前一课中介绍的软件包声明相混淆！

示例：

```
// 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();
```

可以使用以下命令（从 tutorial 父目录中）编译并运行此示例：

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

输出为：

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

protected 访问修饰符

protected 访问修饰符使变量或函数可供同一个软件包中的其他代码以及任何软件包中的子类访问。

示例：

```
// 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();
```

编译并运行以下演示代码：

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

输出为：

```
Class two says Hello!
```

注意：不能对类应用此访问修饰符，这就是我们为什么将类 one 标记为 public 的原因。

public 访问修饰符

public 类、变量或函数的可见性最高。可以从任何软件包中的任何类或脚本对其进行访问。

示例：

```
// 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();
```

编译并运行以下示例：

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

输出：

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

public-read 访问修饰符

`public-read` 访问修饰符定义可公开读取但（在默认情况下）只能从当前脚本内部写入的变量。为了扩大该变量的写入访问级别，可以在该修饰符前面添加 `package` 或 `protected` 修饰符（即 `package public-read` 或 `protected public-read`）。这会将该变量的写入访问级别设置为 `package` 或 `protected`。

示例：

```
// Inside file tutorial/one.fx
package tutorial;
```

```
public-read var x = 1;

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

编译并运行以下示例：

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

输出为 "1"，这证明了可以从 tutorial/one.fx 脚本外部读取 x。

现在，让我们尝试修改其值：

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

结果为编译时错误：

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

为了能够修改该变量的值，我们必须扩大 x 的写入访问级别：

```
// 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);
```

此示例现在将进行编译并将 "2" 输出到屏幕上。

public-init 访问修饰符

public-init 访问修饰符定义可以由任何软件包中的对象字面值公开初始化的变量。但是，后续的写入访问将按照与 public-read 相同的方式进行控制（默认情况下写入访问级别是脚本级别，但是该修饰符前面的 package 或 protected 将相应地扩大访问级别）。始终可以从任何软件包读取此变量的值。

示例：

```
// 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);
```

编译并运行以下示例：

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

这将输出 "Initialized this variable from a different package!"，从而证明了其他软件包中的对象字面值可以初始化 message 变量。但是，由于后续的写入访问级别是“仅脚本”级别，因此我们无法更改此变量的值：

```
// Inside file two.fx
import tutorial.one;
var o = one {
    message: "Initialge!"
```

```
}  
o.message = "Changing the message..."; // WON'T COMPILE  
println(o.message);
```

编译时错误为：

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

这就证实了预期的行为：该变量可以公开初始化，但是后续的写入将由其他访问级别来控制。